|  |  |
| --- | --- |
| Описание: Описание: Описание: http://almetpt.ru/img/emblema.png | МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ ТАТАРСТАН  Государственное автономное профессиональное образовательное учреждение  **«АЛЬМЕТЬЕВСКИЙ ПОЛИТЕХНИЧЕСКИЙ ТЕХНИКУМ»** |

**Курс лекций по МДК 02.01. Технология разработки программного обеспечения**

Филимонова А.Ю.

г. Альметьевск, 2021

Содержание

[Лекция 1. Понятия требований, классификация, уровни требований. Методологии и стандарты, регламентирующие работу с требованиями. 4](#_Toc95731889)

[1.1 Определение понятия требования 4](#_Toc95731890)

[1.2 Классификация требований. 5](#_Toc95731891)

[1.3 Уровни требований 6](#_Toc95731892)

[1.4 Методологии и стандарты, регламентирующие работу с требованиями 8](#_Toc95731893)

[Лекция 2. Современные принципы и методы разработки программных приложений. 10](#_Toc95731894)

[2.1 Метод нисходящего проектирования (метод пошаговой детализации, метод иерархического проектирования, top-down-подход) 10](#_Toc95731895)

[2.2 Модульное проектирование 11](#_Toc95731896)

[2.3 Структурное программирование 12](#_Toc95731897)

[2.4 CASE-технологии 13](#_Toc95731898)

[2.5 Технологии RAD 13](#_Toc95731899)

[2.6 Data Warehouse 14](#_Toc95731900)

[2.7 Система OLAP (On-Line Analytical Process) 15](#_Toc95731901)

[Лекция 3. Методы организации работы в команде разработчиков. Системы контроля версий 15](#_Toc95731902)

[3.1 Авторская разработка 16](#_Toc95731903)

[3.2 Коллективная разработка 17](#_Toc95731904)

[3.3 Общинная модель разработки 18](#_Toc95731905)

[3.4 Организация работы команды в системе контроля версий. 19](#_Toc95731906)

[Лекция 4. Основные подходы к интегрированию программных модулей. 26](#_Toc95731907)

[4.1 Интеграция на уровне данных 27](#_Toc95731908)

[4.2 Интеграция на уровне физических, программных и пользовательских интерфейсов 27](#_Toc95731909)

[4.3 Интеграция на функционально-прикладном и организационном уровнях 29](#_Toc95731910)

[4.4 Интеграция на уровне корпоративных программных приложений 30](#_Toc95731911)

[4.5 Интеграция при помощи Web-сервисов 31](#_Toc95731912)

[Лекция 5. Стандарты кодирования. 33](#_Toc95731913)

[Лекция 6. Описание требований: унифицированный язык моделирования- UML. 45](#_Toc95731914)

[Лекция 7. Диаграммы UML. 45](#_Toc95731915)

[Лекция 8. Описание и оформление требований (спецификация). 45](#_Toc95731916)

[8.1 Шаблон спецификации требований 46](#_Toc95731917)

[8.2 Структура спецификации требований 48](#_Toc95731918)

[Лекция 9. Анализ требований и стратегии выбора решения. 60](#_Toc95731919)

[9.1 Анализ требований 61](#_Toc95731920)

[9.2 Анализ несоответствия 62](#_Toc95731921)

[9.3 Подход на основе лучших практик 63](#_Toc95731922)

[9.4 Процесс выбора решения 64](#_Toc95731923)

[Лекция 10. Цели, задачи и виды тестирования. Стандарты качества программной документации. Меры и метрики. 67](#_Toc95731924)

[10.1 Виды тестирования 70](#_Toc95731925)

[10.2 Автономная отладка программного средства 71](#_Toc95731926)

[10.3 Комплексная отладка ПС 74](#_Toc95731927)

[10.4 Работа с ошибками 76](#_Toc95731928)

[Лекция 11. Тестовое покрытие. 77](#_Toc95731929)

[11.1 Покрытие требований (Requirements Coverage) 79](#_Toc95731930)

[11.2 Покрытие кода (Code Coverage) 79](#_Toc95731931)

[11.3 Тестовое покрытие на базе анализа потока управления 80](#_Toc95731932)

[Лекция 12. Тестовый сценарий, тестовый пакет. 81](#_Toc95731933)

[Лекция 13. Анализ спецификаций. Верификация и аттестация программного обеспечения. 85](#_Toc95731934)

# **Лекция 1. Понятия требований, классификация, уровни требований. Методологии и стандарты, регламентирующие работу с требованиями.**

# **1.1 Определение понятия требования**

Л.Новиков в русской редакции нотации RUP приводит следующее определение: «Требование – это условие или возможность, которой должна соответствовать система».

В IEEE Standard Glossary of Software Engineering Terminology (1990) данное понятие трактуется шире. Требование – это:

1. условия или возможности, необходимые пользователю для решения проблем или достижения целей;

2. условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;

3. документированное представление условий или возможностей для пунктов 1 и 2 (конец цитаты).

Введём ещё одно определение. Требования – это исходные данные, на основании которых проектируются и создаются автоматизированные информационные системы. Первичные данные поступают из различных источников, характеризуются противоречивостью, неполнотой, нечёткостью, изменчивостью. Требования нужны в частности для того, чтобы Разработчик мог определить и согласовать с Заказчиком временные и финансовые перспективы проекта автоматизации. Поэтому значительная часть требований должна быть собрана и обработана на ранних этапах создания АИС. Однако собрать на ранних стадиях все данные, необходимые для реализации АИС, удаётся только в исключительных случаях. На практике процесс сбора, анализа и обработки растянут во времени на протяжении всего жизненного цикла АИС, зачастую нетривиален и содержит множество подводных камней.

# **1.2 Классификация требований.**

Требования к продукту. В своей основе требования – это то, что формулирует заказчик. Цель, которую он преследует – получить хороший конечный продукт: функциональный и удобный в использовании. Поэтому требования к продукту являются основополагающим классом требований. Более подробно требования к продукту детализируются в следующих ниже классификациях.

Требования к проекту. Вопросы формулирования требований к проекту, т.е. к тому, как Разработчик будет выполнять работы по созданию целевой системы, казалось бы, не лежат в компетенции Заказчика. Без регламентации процесса Заказчиком легко можно было бы обойтись, если бы все проекты всегда выполнялись точно и в срок. Однако, к сожалению, мировая статистика результатов программных проектов говорит об обратном. Заказчик, вступая в договорные отношения с Разработчиком, несёт различные риски, главными из которых является риск получить продукт с опозданием, либо ненадлежащего качества. Основные мероприятия по контролю и снижению риска – регламентация процесса создания программного обеспечения и его аудит.

Насколько подробно Заказчику следует регламентировать требования к проекту – вопрос риторический. Ответ на него зависит о множества факторов, таких, как ценность конечного продукта для Заказчика, степень доверия Заказчика к Разработчику, сумма подписанного контракта, увязка срока сдачи продукта в эксплуатацию с бизнес-планами Заказчика и т.д. Однако, со всей определённостью можно сказать следующее: 1) регламентация процесса Заказчиком позволяет снизить его риски; 2) мероприятия Заказчика по регламентации процесса приводят к дополнительным накладным расходам. Требуется найти разумный компромисс между степенью контроля рисков и величиной расходов.

В качестве требований к проекту могут быть внесён регламент отчётов Разработчика, совместных семинаров по оценке промежуточных результатов, определены характеристики компетенций участников рабочей группы, исполняющих проект, их количество, указана методология управления проектом. Ниже сформулирован пример формулировки требования к оффшорному проекту (Заказчик и Разработчик физически находятся в разных государствах) – в этой ситуации Заказчику требуется жёсткий контроль над Разработчиком.

1) Разработчик представляет Заказчику согласованный план работ c детализацией (WorkBreakdownStructure - WBS) с точностью до конкретных исполнителей.

2) Разработчик осуществляет ежедневные сборки, регрессионное тестирование компонент разрабатываемого продукта и тестирование продукта в целом.

3) Все управленческие и проектные артефакты, исходные коды и тестовые примеры размещаются в режиме online в интегрированной среде разработки Rational ClearCase с возможностью для Заказчика осуществления online-мониторинга на базе web-технологий.

# **1.3 Уровни требований**

Внедрение ИС на предприятии всегда преследует конкретные бизнес-цели – такие, как, например, повышение прозрачности бизнеса, сокращение сроков обработки информации, экономия накладных расходов и т.д. Современные информационные системы – это крупные программные системы, содержащие в себе множество модулей, функциональных, интерфейсных элементов, отчётов и т.д. Как охватить единым взором такие разнородные вещи, как цели, преследуемые топ-менеджером предприятия Заказчика, описание интерфейса пользователя и характеристики модуля, осуществляющего расчёт себестоимости изделия?

К счастью, человечество уже давно изобрело приёмы борьбы со сложностью, широко применяемые в моделировании сложных объектов – абстракцию и декомпозицию. Применительно к дисциплине анализа требований к программным системам эти принципы работают следующим образом. Требования разделяются по уровням. Уровни требований связаны, с одной стороны, с уровнем абстракции системы, с другой – с уровнем управления на предприятии.

Обычно выделяют три уровня требований.

На верхнем уровне представлены так называемые бизнес-требования (business requirements). Примеры бизнес-требования: система должна сократить срок оборачиваемости обрабатываемых на предприятии заказов в три раза. Бизнес-требования обычно формулируются топ-менеджерами, либо акционерами предприятия.

Следующий уровень – уровень требований пользователей (user requirements). Пример требования пользователя: система должна представлять диалоговые средства для ввода исчерпывающей информации о заказе, последующей фиксации информации в базе данных и маршрутизации информации о заказе к сотруднику, отвечающему за его планирование и исполнение. Требования пользователей часто бывают плохо структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий уровень, в котором осуществляется формализация требований.

Третий уровень – функциональный (functional requirements). Пример функциональных требований (или просто функций) по работе с электронным заказом: заказ может быть создан, отредактирован, удалён и перемещён с участка на участок.

Существуют объективные противоречия между требованиями различных уровней. Так, очевидным бизнес-требованием является требование о полноте информации, собираемой на рабочих местах пользователей в единую базу данных. Чем полнее информация – тем глубже база для анализа деятельности и принятия решений. С другой стороны, конкретному пользователю системы вполне может быть достаточно использования только той части информации, которая влияет на выполнение его основных функций.

Важные правила внедрения и использования АИС на предприятии – «Одна точка сбора», «Данные собираются там, где они появляются». Использование этих правил позволяет избежать затрат на необоснованное дублирование информации и, что важнее – потерь от ошибок учёта, неизбежно возникающих при дублировании точек ввода.

Внедрение АИС на предприятии приводит к необходимости оснащения всех точек ввода информации автоматизированными рабочими местами (АИС), обучению персонала и, зачастую, оптимизации и повышению уровня формализации рабочих процессов, выполняемых персоналом. Поэтому внедрение АИС – непростой процесс, часто требующий «перекройки человеческого материала» и встречающий сопротивление со стороны пользователей, которые не готовы, либо не хотят работать по-новому.

# **1.4 Методологии и стандарты, регламентирующие работу с требованиями**

Среди основополагающих нормативных документов в области работы с требованиями можно выделить следующие.

1. Разработки IEEE:

* IEEE 1362 “Concept of Operations Document”.
* IEEE 1233 «Guide for Developing System Requirements Specifications».
* IEEE Standard 830-1998, «IEEE Recommended Practice for Software Requirements Specifications»
* IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.121990
* IEEE Guide to the Software Engineering Body of Knowledge (1) - SWEBOK, 2004.

1. Отечественные ГОСТ:

* ГОСТ 34.601-90. Информационная технология. Автоматизированные системы. Стадии создания.
* ГОСТ 34.602-89. Информационная технология. Техническое задание на создание автоматизированной системы
* ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению.

# **Лекция 2. Современные принципы и методы разработки программных приложений.**

# **2.1 Метод нисходящего проектирования (метод пошаговой детализации, метод иерархического проектирования, top-down-подход)**

Суть метода заключается в определении спецификаций компонентов системы путем последовательного выделения в ее составе отдельных составляющих и их постепенной детализации до уровня, обеспечивающего однозначное понимание того, что и как необходимо разрабатывать и реализовывать.

Этот метод является незаменимым при разработке сложных по характеру и больших по объему программ, когда к их разработке необходимо привлекать большое число программистов, работающих параллельно. Он позволяет концентрировать внимание разработчиков на наиболее ответственных частях программы, а также облегчает возможность постоянного контроля за ее работоспособностью по мере разработки, отладки и объединения отдельных составляющих программ за счет организации непрерывности этого процесса в течение всей разработки.

Для ускорения разработки программного комплекса часто вместо некоторых программ нижнего уровня, находящихся в процессе разработки, могут применяться специальные "программы-заглушки" Программы-заглушки требуются только на ранних стадиях разработки для того, чтобы не сдерживать общий ход создания программного комплекса. Суть программы-заглушки заключается в том, что при обращении к ней в соответствии с заданным набором исходных тестовых данных она не формирует, а выбирает результат "решения" из заранее подготовленного набора. Благодаря этому обеспечивается возможность имитировать работу на ЭВМ реально создаваемой программы, а следовательно, осуществлять проверку работоспособности программ верхнего уровня еще до того, как будут разработаны и отлажены все составляющие программы нижнего уровня.

# **2.2 Модульное проектирование**

Реализация метода *нисходящего проектирования* тесно связана с другим понятием программирования - *модульным проектированием*, так как на практике при декомпозиции сложной программы возникает вопрос о разумном пределе ее дробления на составные части. Вместе с тем понятие модульности нельзя сводить только к представлению сложных программных комплексов в виде набора отдельных функциональных блоков.

*Модуль* - это последовательность логически взаимосвязанных фрагментов задачи, оформленных как отдельная часть программы. При этом программные модули должны обладать следующими свойствами:

* на модуль можно ссылаться (т.е. обращаться к нему) по имени, в том числе и из других модулей;
* по завершении работы модуль должен возвращать управление тому модулю, который его вызывал;
* модуль должен иметь один вход и выход;
* модуль должен иметь небольшой размер, обеспечивающий его обозримость.

При разработке сложных программ в них выделяют головной управляющий модуль, подчиненные ему модули, обеспечивающие реализацию отдельных функций управления, функциональную обработку (т.е. непосредственную реализацию основного назначения программного комплекса), а также вспомогательные модули, обеспечивающие сервисное обслуживание пакета (например, сбор и анализ статистики работы программы, обработка различного рода ошибочных ситуаций, обучение и выдача подсказок и т.п.).

Модульный принцип разработки программ обладает следующими преимуществами:

* большую программу могут разрабатывать одновременно несколько исполнителей, и это позволяет сократить сроки ее разработки;
* появляется возможность создавать и многократно использовать в дальнейшем библиотеки наиболее употребимых программ;
* упрощается процедура загрузки больших программ в оперативную память, когда требуется ее сегментация;
* возникает много естественных контрольных точек для наблюдения за осуществлением хода разработки программ, а в последующем для контроля за ходом исполнения программ;
* обеспечивается более эффективное тестирование программ, проще осуществляются проектирование и последующая отладка.

Преимущества модульного принципа построения программ особенно наглядно проявляются на этапе сопровождения и модификации программных продуктов, позволяя значительно сократить затраты сил и средств на реализацию этого этапа.

# **2.3 Структурное программирование**

Актуальная для начального периода развития и использования ЭВМ проблема разработки программ, занимающих минимум основной памяти и выполняющихся за кратчайшее время, в последующем в связи c резким падением стоимости аппаратной части ЭВМ, значительным возрастанием их быстродействия и объемов памяти сменилась необходимостью разработки и применения принципиально новых методов составления программ. Все это нашло свое воплощение в разработке принципа *структурного программирования.* Одной из целей структурного программирования было стремление облегчить разработку и отладку программных модулей, а главное - их последующее сопровождение и модификацию.

В настоящее время структурное программирование - это целая дисциплина, объединяющая несколько взаимосвязанных способов создания ясных, легких для понимания программ. Эффективность применения современных универсальных языков программирования во многом определяется удобством написания с их помощью структурных программ.

# **2.4 CASE-технологии**

За последнее десятилетие в области средств автоматизации программирования сформировалось новое направление под общим названием CASE-технологии (Computer Aided Software Engineering).

*CASE-технология* представляет собой совокупность средств системного анализа, проектирования, разработки и сопровождения сложных программных систем, поддерживаемых комплексом взаимоувязанных инструментальных средств автоматизации всех этапов разработки программ. Благодаря структурным методам CASE-технология на стадиях анализа и проектирования обеспечивает разработчиков широкими возможностями для различного рода моделирования, а централизованное хранение всей необходимой для проектирования информации и контроль за целостностью данных гарантируют согласованность взаимодействия всех специалистов, занятых в разработке ПО.

# **2.5 Технологии RAD**

В начале 80-х годов появилась методология, по которой разработка программы начиналась не после завершения процесса выработки окончательных требований к ней, а как только устанавливались требования на первый, “стартовый” (пилотный) вариант прикладной программы, позволяющий начать содержательную работу по ее реализации на компьютере.

Это дало пользователю возможность, получая уже с первых шагов конкретное представление о характере реализации задачи, уточнять ее постановку. Тем самым облегчался процесс экспериментального поиска нужного решения автоматизации задачи. Благодаря тесному взаимодействию разработчика с заказчиком (пользователем) на самом ответственном этапе создания прикладных программ между ними достигалось быстрое взаимопонимание цели поставленной задачи и возможности ее автоматизации в данных конкретных условиях. Это повышало скорость разработки программ и послужило основанием для названия такой технологии *RAD (Rapid Application Developmen* - быстрая разработка программ), которая получила широкое распространение.

# **2.6 Data Warehouse**

Другое направление разработки прикладных программных средств, олицетворяющее собой современный подход к реализации широкого круга задач для принятия управленческих решений, базируется на концепции создания специального хранилища данных (Data Warehouse). Основное отличие концепции Data Warehouse от традиционного представления баз данных заключается в следующем:

* во-первых, в том, что актуализация данных в Data Warehouse означает не обновление элементов информации, а добавление новых элементов к уже имеющимся (что расширяет возможности проведения различного рода сравнительного анализа);
* во-вторых, в том, что наряду с информацией, непосредственно отражающей состояние системы управления, в Data Warehouse аккумулируются и метаданные.

*Метаданные* (данные о данных) облегчают возможность визуального представления содержимого Data Warehouse, позволяют, "перемещаясь" по хранилищу, быстро отбирать необходимые данные для последующей обработки.

Основные типы метаданных Data Warehouse отражают:

* структуру и содержимое хранилища;
* соответствие между исходными и выходными данными;
* объемные характеристики данных;
* критерии архивирования;
* отношения между данными;
* информацию по кодированию;
* интервал жизни данных и т.п.

Концепция *Data Warehouse* поддерживается RAD средствами разработки прикладного ПО.

Концепция Data Warehouse обеспечивает возможность разработки программных приложений для поддержки процессов принятия решений с использованием OLAP-систем.

# **2.7 Система OLAP (On-Line Analytical Process)**

Предоставляет возможность разработки информационных систем, ориентированных на yна организацию многомерных баз данных и создание корпоративных сетей, а также обеспечивает поддержку Web-технологий в сетях Internet/Intranet

+Успешное применение инструментальных средств OLAP-систем объясняется быстротой разработки приложений, гибкостью и широкими возможностями в области доступа к данным и их преобразования. В настоящее время на рынке ПО предлагается большое число OLAP-стем, разработчиками которых являются различные фирмы, например: IBM, Informix, Microsoft, Oracle, Sybase и др.

# **Лекция 3. Методы организации работы в команде разработчиков. Системы контроля версий**

Все множество разработок в зависимости от количества участников и типов [взаимоотношений](https://pandia.ru/text/category/vzaimootnoshenie/) между ними может быть сведено к [триаде](https://pandia.ru/text/category/triadi/) разработок приведенной на рисунке 3.1.

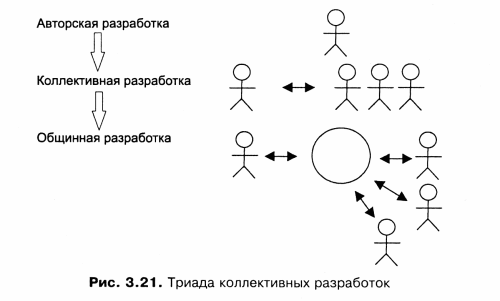


Рисунок 3.1 – Виды разработок в зависимости от количества участников

# **3.1 Авторская разработка**

Авторская разработка – принцип создания программных продуктов, при котором весь жизненный цикл разработки поддерживается одним единственным человеком.

Этот принцип был распространен в 70 – 80-е годы ХХ века. Сейчас он применяется редко. Примеры авторских разработок: операционная система Диспак, текстовый редактор Лексикон, трансляторы с языков Algol-68 и Pascal.

Принцип авторской разработки неприменим для многих современных разработок из-за их сложности, объема и требований к качеству и сопровождению. С другой стороны, программное обеспечение, начиная с момента появления персональных компьютеров, стало продуктом массового потребления, приносящим огромный доход. В этой области быстро выросли и стали доминировать крупные компьютерные компании с развитой структурой менеджмента и мощной рекламой. Авторская разработка может выиграть по производительности в тридцать и более раз у коллективной разработки, что достигается за счет:

-  исключения межличностных коммуникаций, связанных с необходимостью порождения и изучения большого количества технологической документации;

-  исключения работ по разбиению проекта на составляющие, по распределению их между исполнителями, по координации деятельности исполнителей и контролю за их работой.

Объем программного продукта, выполненного методом авторской разработки, в 5-20 раз меньше по сравнению с индустриальными аналогами.

Авторская разработка предполагает достижение профессионального успеха, известности и славы в одиночку. Это реально, если правильно выбрать профессиональную «нишу», область ведения разработки.

# **3.2 Коллективная разработка**

Одним из основных вопросов коллективной разработки является разделение труда – от равноправных соисполнителей до организации в виде жесткой иерархии (например, бригада главного программиста).

**Равноправные соисполнители**

Бригада равноправных соисполнителей обычно состоит из специалистов, занимающихся примерно подобными задачами в рамках одного проекта. Специализаций в рамках одной бригады может быть несколько.

Состав такой бригады:

-  инженеры-разработчики;

-  технические писатели;

-  инженеры тестирования;

-  инженеры качества;

  специалисты по сопровождению продукта;

-  специалисты по продажам продукта.

**Бригада главного программиста**

Один участник команды занимается основной работой, остальные оказывают ему всевозможную поддержку. Бригада главного программиста включает десять человек, выполняющих специализированные роли в команде.

Состав бригады главного программиста:

-  главный программист;

-  дублер;

-  администратор;

-  секретарь;

-  редактор;

-  языковед;

-  инструментальщик;

- отладчик;

-  делопроизводитель.

**Типы совместной деятельности**

Коллективная разработка предполагает большое количество различных действий, причем степень совместной деятельности может существенно изменяться от одного действия к другому. Выделяют четыре типа совместной деятельности:

-  **мандатная** деятельность (формальные собрания, проводимые на регулярной основе, 4% рабочего времени программисты проводят на собраниях);

-  **созываемая**деятельность (решение двух или более программистов собираются вместе для решения некоторого технического вопроса, такие собрания не планируются, участвуют только заинтересованные в решении проблемы, на это уходит 14% рабочего времени);

-  **естественная совместная** деятельность (минимум двое программистов работают над одной и той же задачей одновременно и обмениваются информацией о работе, занимает около 41% рабочего времени);

-  **индивидуальная**деятельность (программист работает над задачей, другие программисты над задачей не работают, деятельность занимает 41% рабочего времени).

# **3.3 Общинная модель разработки**

Идеология общинной («базарной») разработки сформулирована в статье Эрика Раймонда «Собор и Базар».

Общинная модель характеризуется тремя основными факторами.

- *Децентрализованность разработки.* Не существует ограничения сверху на количество людей, принимающих участие в проекте. Разработки такого типа ведутся на базе сети Интернет и могут включать любого заинтересованного разработчика Сети.

- Разработка ведется *на базе открытых исходных текстов.* По ним можно разобраться с сутью задачи и в любой момент подключиться к разработке.

- *Большое количество внешних тестеров* (бета-тестеров), позволяющих быстро обнаруживать ошибки и проблемы в программе.

***Эрик Рэймонд сформулировал несколько уроков, которые позволяют понять особенности общинной разработки:***

-  Каждая хорошая программа начинается с энтузиазма разработчика.

-  Хорошие программисты знают, что можно написать, а великие – что можно переписать.

-  При правильном отношении интересная проблема найдет вас сама.

-  Когда вы теряете интерес к программе, ваша последняя обязанность передать ее компетентному преемнику.

-  Следует выпускать ранние и частые версии программ.

-  Обнаружить проблему и исправить ее могут разные люди.

-  Иногда использовать идеи пользователей лучше, чем свои идеи.

В сети Интернет можно найти большое количество сайтов с проектами, разрабатываемыми по общинной модели.

# **Организация работы команды в системе контроля версий.**

Системы контроля версий (их ещё называют системами управления версиями) –  один из инструментов, который использует в своей работе любой программист от первокурсника до опытного тим-лида с сотнями успешных проектов.

— Ссылки на примеры кода в репозиториях помещают в своё портфолио.

— Большие проекты с открытым программным кодом получают тысячи улучшений, благодаря размещению на специализированном хостинге.

— От 30 до 70% кода, использованного в программном продукте, профессиональные разработчики могут скопировать с проектов, представленных в открытых репозиториях.

Незаменимы системы контроля версий в командной разработке, где синхронизация процесса играет важную роль.

Но даже если Вы принципиально не работаете в команде, система контроля версий может помочь Вам справиться со многими сложными ситуациями.

С необходимостью отката программы к предыдущей версии рано или поздно сталкивается любой программист.

*В точке А программа отлично работает, а попытка расширить функционал, ввести дополнительные переменные, классы и пр. приводит в появлению ошибки.*

Распознать ошибку удаётся не всегда. Поэтому многие программисты предпочитают «пройти перезагрузку» — откатиться к стабильной рабочей версии и переписать дополнения заново, построчно анализируя весь добавленный код.

Реализовать такую процедуру помогает использование системы контроля версий (СКВ), которое позволяет четко отследить историю разработки, найти стабильную версию и продолжить работу.

В системе контроля версий может содержаться, как программный код, так и любая другая информация (включая текстовые файлы, изображения и т.п.)

**Примитивная модель хранения версий**

В примитивной модели актуальные копии проекта перезаписываются в отдельную директорию через определённый промежуток времени.

***Достоинства:***

— возможность восстановления данных одной из записанных версий.

***Недостатки:***

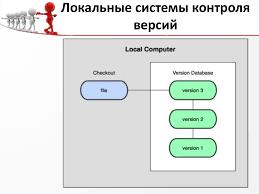
— сложности с поиском необходимой версии в обширной и плохо структурированной базе данных;

— возможность потери данных вследствие возникновения физических поломок оборудования;

— отсутствие возможности совместной разработки.

**Локальные системы контроля версий**

Локальные  СКВ обычно хранят на компьютере список изменений, внесенных в файлы. Основываясь на этих данных, система контроля версий воссоздает нужную версию файла (актуальную на определенный момент времени).



Локальные системы контроля версий

***Достоинства:***

— возможность восстановления данных  из определенной версии (точно определяется по времени записи);

— высокая скорость выполнения восстановления (база данных четко структурирована, поэтому сложностей при поиске не возникает, сетевая задержка отсутствует, поскольку данные хранятся непосредственно на рабочем компьютере).

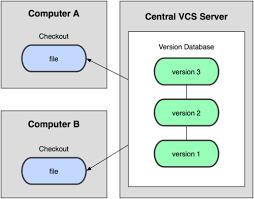
***Недостатки:***

— возможность потери данных вследствие возникновения физических поломок оборудования;

— отсутствие возможности совместной разработки.

**Централизованные системы контроля версий**

Централизованные  системы контроля версий предполагают сохранение версий проектов на  общий сервер, с которого потом получают нужные версии клиенты.



Централизованные системы контроля версий

***Достоинства:***

— возможность восстановления данных  из определенной версии (точно определяется по времени записи);

— возможность ведения командной разработки проекта;

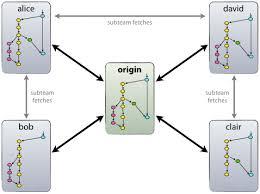
***Недостатки:***

—  отсутствие доступа к данным при сбое работы сервера;

— довольно низкая скорость работы (из-за возникновения сетевых задержек).

**Децентрализованные системы контроля версий**

В децентрализованных системах контроля версий при каждом копировании удалённого репозитория (расположенного на сервере) происходит полное копирование данных в локальный репозиторий (установленный на рабочем компьютере). Каждая копия содержит все данные, хранящиеся в удалённом репозитории. В случае, возникновения технической неисправности на стороне сервера, удаленный репозиторий можно перезаписать с любой сохраненной копии.



Децентрализованные системы контроля версий

***Достоинства:***

— возможность восстановления данных  из определенной версии (точно определяется по времени записи);

— возможность ведения командной разработки проекта;

— при сбое работы сервера система сохраняет данные в локальном репозитории, что позволяет эффективно вести процесс разработки, а после восстановления работы сервера, передать все изменения в удаленный репозиторий;

— при физической поломке сервера данные можно легко перенести в новый удалённый репозиторий с любого локального репозитория;

— высокая скорость работы (в ходе работы данные записываются и получаются из локального репозитория, поэтому сетевые задержки отсутствуют).

**Современные системы контроля версий**

Существует много систем контроля версий (Git, Darcs, Mercurial, Bazaar, Monotone и т.д), сходных по принципу работы и конечным задачам. Отличаются они друг от друга архитектурой, использованными решениями и удобством работы.

Самая популярная на сегодняшний день система контроля версий – Git.

**Git**



Git

Умение работать с git’ом — обязательный навык для программиста любого профиля. Можно долго обсуждать преимущества и недостатки разных систем контроля версий, но большинство компаний используют git, поэтому уметь работать с  git’ом нужно всем.

Git – распределённая система контроля версий. Что даёт ей все преимущества децентрализованной СКВ:

— высокую скорость проведения всех операций (за счет отсутствия сетевой задержки);

— идеальные условия для командной разработки;

— страховку от потери информации при возникновении проблем с центральным сервером.

Для контроля версий в git используются 2 репозитория: локальный и удаленный. Локальный репозиторий (полноценный репозиторий, а не ссылки или копии отдельных ветвей) находится на компьютере разработчика, а удаленный на удалённом сервере. Доступ к удаленному репозиторию обеспечивается  благодаря гит-хостингу Github, Google Code, GitLab и т.д.

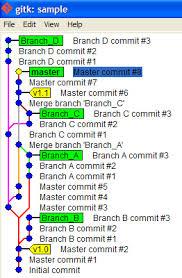
**Как работает git**

Взаимодействие с удаленным репозиторием происходит при наличии интернета и, по сути, представляет собой синхронизацию двух репозиториев.

Команда push копирует новые данные, содержащиеся в локальном репозитории, в удалённый репозиторий, а команда pull передает данные из удаленного репозитория в локальный.

Каждая версия документа, внесенные обновления и т.д записываются в локальный репозиторий.

**Дерево проекта**

Дерево файлов в системе контроля версий

В репозитории содержится «дерево» проекта, то есть все сохраненные версии файлов.

Дерево может быть прямым (в этом случае каждое последующее сохранение файлов производилось после предыдущего без возвращения к более ранним версиям) и разветвленным.

К появлению «веток» приводит работа с более ранними версиями и сохранение внесённых изменений.

На различных ветках дерева  содержатся сохранения, основой которых был один исходный файл. В ходе работы в файлы на разных ветках были внесены разные изменения. В системе управления версиями можно работать со всеми ветками дерева проекта, пошагово, изменяя и дополняя содержащиеся в них данные. После проведения ряда изменений 2 ветки могут «срастись», в новой версии файла будут учтены все внесенные изменения.

**Git-хостинг**

Для комфортной работы с git нужно зарегистрироваться на любом git-хостинге. Их довольно много: [Github](https://github.com/), [Sourceforge](https://sourceforge.net/), [Google Code](https://code.google.com/),  [GitLab](https://gitlab.com/), [Codebase](https://www.codebasehq.com/) и т.д.

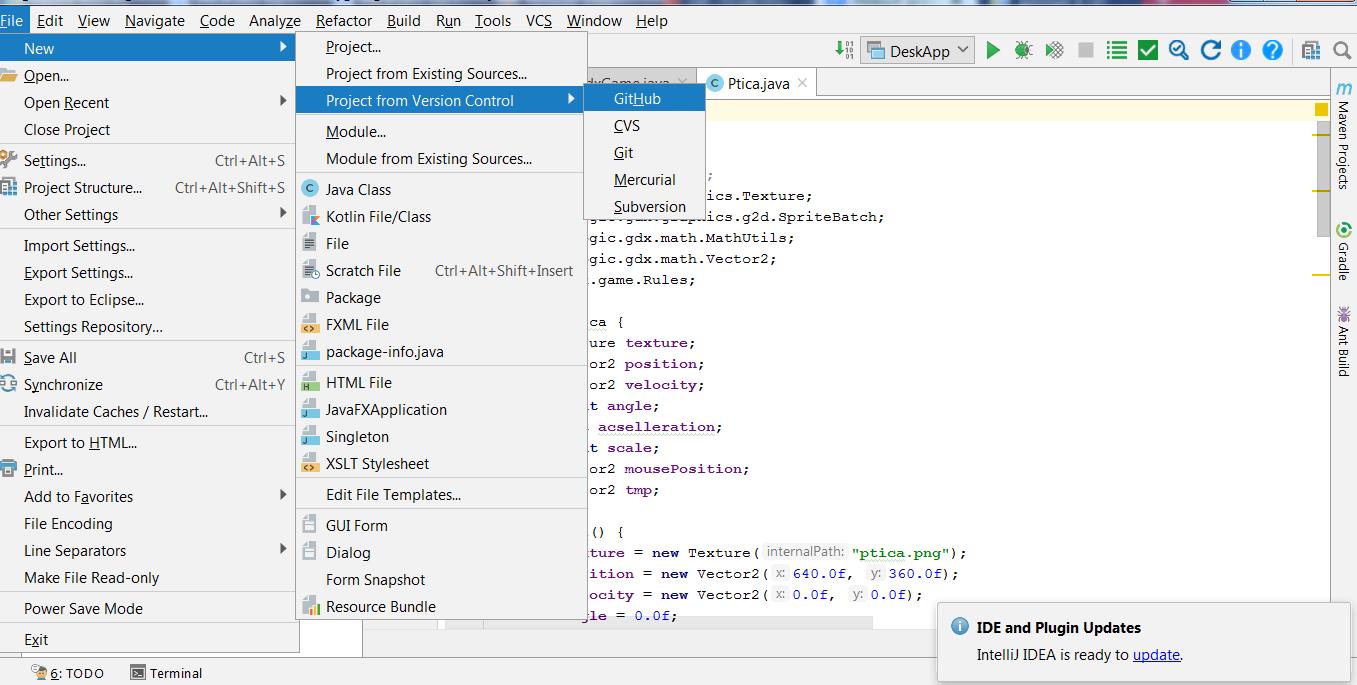
Самый популярный на данный момент git-хостинг  – это [Github](https://github.com/).

Популярность его обоснована простым интуитивно понятным интерфейсом, поддержкой проектов с открытым кодом (возможностью бесплатно размещать такие проекты на хостинге) и обширным функционалом.

**Git-клиент**

Для удобства работы с системой контроля версий git разработан целый ряд графических git-клиентов. Это программы, позволяющие эффективно работать с системой контроля версий, используя графический интерфейс.

Многие IDE предполагают возможность работы с git.



Работа с Git через IDE

Работа с системами контроля версий  — важный навык, нужный каждому программисту.

# **Лекция 4. Основные подходы к интегрированию программных модулей.**

Интеграция — это не просто механическое объединение модулей информационной системы. При разработке плана интеграции исходят прежде всего из стратегических целей развития предприятия, возможного изменения бизнес-логики, в соответствии с которой выстраиваются бизнес-процессы и осуществляется их информационное сопровождение. Интеграция может производиться на уровне форматов и баз данных, программно-аппаратных и сетевых устройств, пользовательских интерфейсов, форм и шаблонов документооборота, программных приложений и т.д.

# **4.1 Интеграция на уровне данных**

Одной из главных проблем интеграции данных является обилие форматов и типов (неструктурированные, частично-структурированные, жёстко-структурированные) данных, а также лавинообразное нарастание их объёмов. Циркулирование разнородных массивов данных и информации в сетях различных служб предприятия создает множество проблем с их сбором, структурированием, обработкой, анализом, хранением, архивированием и передачей пользователю для принятия делового решения.

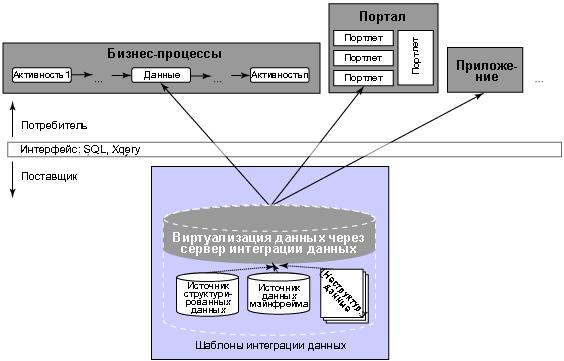


Рисунок 4.1 – Традиционная схема интеграции данных

# **4.2 Интеграция на уровне физических, программных и пользовательских интерфейсов**

Этот вид интеграции начинался как один из видов "лоскутной интеграции", когда предпринимались попытки объединить разрозненные программные приложения, написанные в разное время разными разработчиками, в подобие единого целого. Приложения объединялись по принципу "каждый с каждым", что, в конечном счёте, усложняло их взаимодействие и создавало массу проблем. Кроме того, всё сложнее становилось использовать унаследованные (Legacy Software) и встроенные (Embedded System) системы.

Такой подход хорош для небольшого количества приложений. При большом их числе он практически не работает и не позволяет строить качественно новые запросы к агрегированным данным, т.е. существенного выигрыша от объединения данных нет. В настоящее время проблема интеграции на уровне интерфейсов решается на базе использования информационных подсистем, реализованных стандартными программными приложениями с открытыми интерфейсами (Open Application Programming Interface).

Подобные унифицированные интерфейсы разрабатываются, например, на базе семейства международных стандартов POSIX. В этом случае степень интегрируемости можно характеризовать некоторым числовым показателем (метрикой) который можно, условно говоря, вычислить, перемножив показатель "качества" и "показатель открытости" программного интерфейса. Показателем качества могут выступать такие характеристики, как "совместимость", "надёжность", "переносимость", "понятность", "удобство использования" и пр. В результате мы получим индекс, который (в известной степени) характеризует способность приложения быть частью какого-то другого, глобального композитного приложения.

В настоящее время всё чаще применяется следующий алгоритм: отделяют слой обработки данных от привязанных к ним форм визуализации и реализуют прикладную бизнес-логику на одном из языков третьего поколения (3GL), оформив программный доступ к прикладным функциям в виде хорошо документированного программного интерфейса.

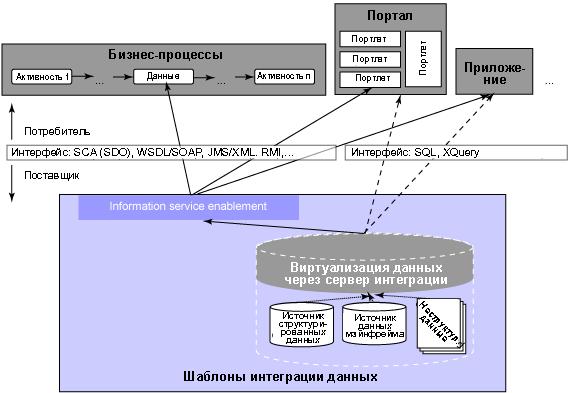


Рисунок 4.2 - Организация доступа к интегрированным данным через открытые интерфейсы

# **4.3 Интеграция на функционально-прикладном и организационном уровнях**

Этот вид интеграции предполагает объединение ряда однотипных или схожих функций в макрофункции с перераспределением потоков данных и управления, а также ресурсов и механизмов для исполнения. Это часто влечёт за собой перестройку организационных структур, бизнес-процессов и, соответственно, схему их информационного и документационного обеспечения.

Выгоды от такой интеграции очевидны — процессы становятся более прозрачными, управляемыми, менее затратными, уменьшается количество обслуживающего персонала, число ошибок при формировании документов и т.д. Однако интеграция такого вида влечёт за собой существенную перестройку или полный реинжиниринг сети процессов, что связано с крупными рисками. Чаще всего такая интеграция проводится в том случае, когда предприятие готовится к внедрению КИС на базе известного решения, которое требует привести бизнес-процессы к требуемому стандарту, или перестраивает свою деятельность в связи со сменой устремлений, открытием филиалов в других странах, освоением новых сегментов рынка и т.д.

# **4.4 Интеграция на уровне корпоративных программных приложений**

Интеграция на уровне приложений (Enterprise Application Integration — EAI,) подразумевает совместное использование исполняемого кода, а не только внутренних данных интегрируемых приложений. Программы разбиваются на компоненты, которые интегрируются с помощью стандартизованных программных интерфейсов и специального связующего ПО.

При таком подходе из этих компонентов создается универсальное программное ядро или платформа, с помощью которых используют все приложения. Для каждого приложения создается только один интерфейс для связи с этим ядром, что существенно облегчает задачу интеграции. Полученную в результате систему легче поддерживать и расширять. Повторное использование функций в рамках имеющейся среды позволяет значительно снизить время и стоимость разработки приложений. В этом случае анализ внутренней конструкции приложений — обязательный этап в оценке степени интегрируемости тех приложений, которые предполагается связывать в рамках того или иного проекта. Этот анализ усложняется тем, что обычно разработчики приложений, являющихся законченными программными продуктами, как правило, не показывают деталей внутренней конструкции приложений.

В связи с этим технология интеграции в настоящее время рассматривает не просто интеграцию приложений, но их интеграцию на базе интеграции бизнес-процессов – в этом случае следует говорить об интеграции на уровне всего предприятия (Enterprise Integration Metodology — EIM).

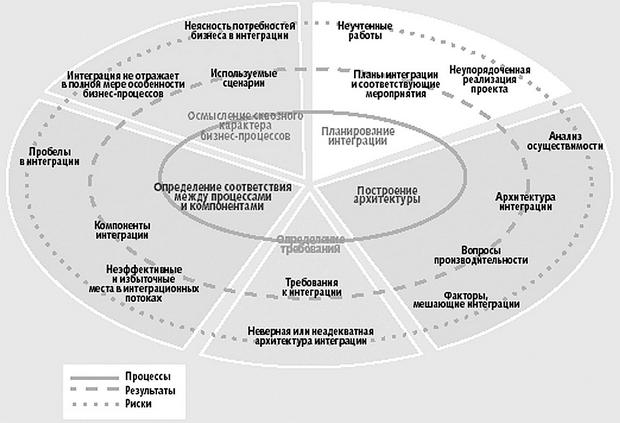


Рисунок 4.3 - Схема применения методологии EIM

# **4.5 Интеграция при помощи Web-сервисов**

Самый современный и быстро развивающийся подход к интеграции приложений. Он основан на обеспечении стандартного для Web-служб интерфейса доступа к приложениям и данным

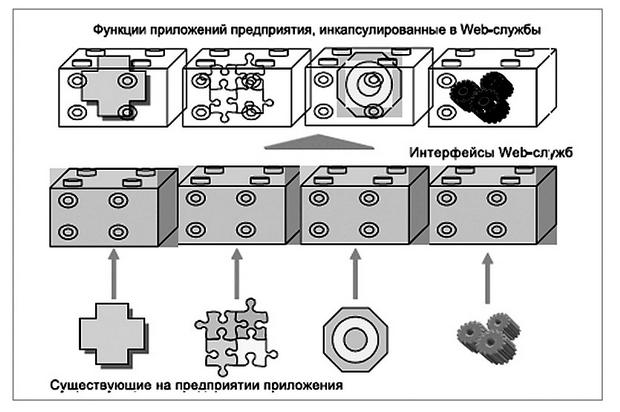


Рисунок 4.4 - Схема доступа с использованием Web-служб

* возможность осуществлять оперативное управление распределенной компанией и ведение консолидированного управленческого учета по нескольким филиалам;
* возможность осуществлять планомерное развитие общекорпоративной информационной системы, интегрируя в нее функциональные компоненты, исходя из приоритетов развития бизнеса компании и потребностей функциональных подразделений, т.е. возможность синхронизировать развитие системы с развитием бизнеса;
* возможность при необходимости заменить любой функциональный компонент другим, более соответствующим текущим бизнес-потребностям;
* возможность инвестировать в развитие информационных технологий не сразу, а поэтапно, на каждом этапе соотнося вложенные средства с полученным бизнес-эффектом, а также снижать общую стоимость автоматизированного рабочего места, включая затраты на создание системы, поддержку рабочих мест и обучение пользователей;
* резкое снижение времени сбора информации, необходимой для принятия управленческих и деловых решений, сокращение времени и трудозатрат на ведение учетных операций, на формирование промежуточных отчетов, на сверку информации между подразделениями и ликвидация противоречивости и несовместимости данных от различных служб;
* cохранение инвестиций в имеющиеся системы и оборудование, в обучение персонала.

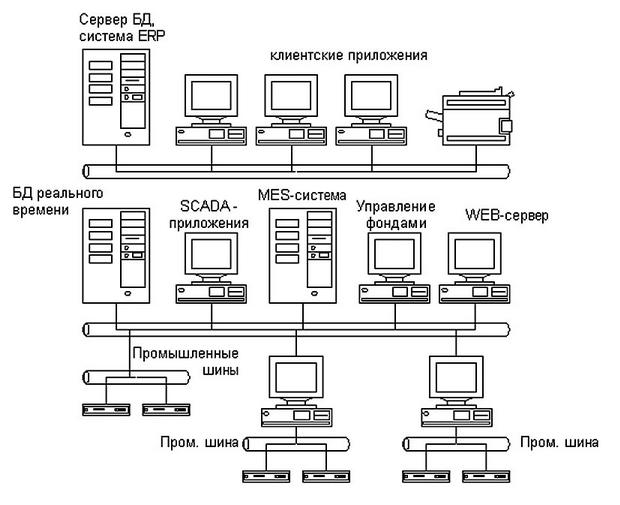


Рисунок 4.5 - Архитектурная модель WebSphere Application Server

# **Лекция 5. Стандарты кодирования.**

**Стандарт кодирования** — набор правил и соглашений, которые описывают базовые принципы оформления программного кода, используемого совместно группой разработчиков.

Целью стандартов кодирования ПО является определение языков программирования, методов, правил и инструментальных средств, которые будут использованы для кодирования ПО. Стандарты кодирования должны включать в себя:

— используемые языки программирования и/или какое-либо их заданное подмножество; должна быть указана ссылка на документы, которые однозначно определяют синтаксис, режим контроля, характер данных и побочные эффекты языка программирования; стандарты могут требовать ограничений на использование некоторых возможностей языка;

— стандарты представления исходного текста (например, ограничение на длину строки, структурное расположение текста, использование пустых строк) и стандарты документирования исходного кода (например, имя автора, история изменений, входные и выходные данные, а также наиболее значимые глобальные данные);

— соглашения по наименованию для компонентов, подпрограмм, переменных, констант;

— условия и ограничения, налагаемые на установленные соглашения кодирования, такие как информационная связность между компонентами ПО, сложность логических или числовых выражений, а также обоснования для их использования;

**Правило 1. Следуйте стандартам оформления кода.**

У каждого языка программирования есть свой стандарт оформления кода, который говорит, как надо делать отступы, где ставить пробелы и скобки, как называть объекты, как комментировать код и т.д.

Например, в этом куске кода в соответствии со стандартом есть 12 ошибок:

for(i=0 ;i

Изучайте стандарт внимательно, учите основы наизусть, следуйте правилам как заповедям, и ваши программы станут лучше, чем большинство, написанные выпускниками вузов.

Многие организации подстраивают стандарты под свои специфические нужды. Например, Google разработал стандарты для более чем 12 языков программирования. Они хорошо продуманы, так что изучите их, если вам нужна помощь в программировании под Google. Стандарты даже включают в себя настройки редактора, которые помогут вам соблюдать стиль, и специальные инструменты, верифицирующие ваш код на соответствие этому стилю. Используйте их.

**Правило 2. Давайте наглядные имена.**

Ограниченные медленными, неуклюжими телетайпами, программисты в древности использовали контракты для имён переменных и процедур, чтобы сэкономить время, стуки по клавишам, чернила и бумагу. Эта культура присутствует в некоторых сообществах ради сохранения обратной совместимости. Возьмите, например, ломающую язык функцию C wcscspn (wide character string complement span). Но такой подход неприменим в современном коде.

Используйте длинные наглядные имена наподобие complementSpanLength, чтобы помочь себе и коллегам понять свой код в будущем. Исключения составляют несколько важных переменных, используемых в теле метода, наподобие итераторов циклов, параметров, временных значений или результатов исполнения.

Гораздо важнее, чтобы вы долго и хорошо думали перед тем, как что-то назвать. Является ли имя точным? Имели ли вы в виду highestPrice или bestPrice? Достаточно ли специфично имя, дабы избежать его использования в других контекстах для схожих по смыслу объектов? Не лучше ли назвать метод getBestPrice заместо getBest? Подходит ли оно лучше других схожих имён? Если у вас есть метод ReadEventLog, вам не стоит называть другой NetErrorLogRead. Если вы называете функцию, описывает ли её название возвращаемое значение?

В заключение, несколько простых правил именования. Имена классов и типов должны быть существительными. Название метода должно содержать глагол. Если метод определяет, является ли какая-то информация об объекте истинной или ложной, его имя должно начинаться с «is». Методы, которые возвращают свойства объектов, должны начинаться с «get», а устанавливающие значения свойств — «set».

**Правило 3. Комментируйте и документируйте.**

Начинайте каждый метод и процедуру с описания в комментарии того, что данный метод или процедура делает, параметров, возвращаемого значения и возможных ошибок и исключений. Опишите в комментариях роль каждого файла и класса, содержимое каждого поля класса и основные шаги сложного кода. Пишите комментарии по мере разработки кода. Если вы полагаете, что напишете их потом, то обманываете самого себя.

Вдобавок, убедитесь, что для вашего приложения или библиотеки есть руководство, объясняющее, что ваш код делает, определяющий его зависимости и предоставляющий инструкции для сборки, тестирования, установки и использования. Документ должен быть коротким и удобным; просто README-файла часто достаточно.

**Правило 4. Не повторяйтесь.**

Никогда не копируйте и не вставляйте код. Вместо этого выделите общую часть в метод или класс (или макрос, если нужно), и используйте его с соответствующими параметрами. Избегайте использования похожих данных и кусков кода. Также используйте следующие техники:

Создание справочников API из комментариев, используя Javadoc и Doxygen.

Автоматическая генерация Unit-тестов на основе аннотаций или соглашений об именовании.

Генерация PDF и HTML из одного размеченного источника.

Получение структуры классов из базы данных (или наоборот).

**Правило 5. Проверяйте на ошибки и реагируйте на них.**

Методы могут возвращать признаки ошибки или генерировать исключения. Обрабатывайте их. Не полагайтесь на то, что диск никогда не заполнится, ваш конфигурационный файл всегда будет на месте, ваше приложение будет запущено со всеми нужными правами, запросы на выделение памяти всегда будут успешно исполнены, или что соединение никогда не оборвётся. Да, хорошую обработку ошибок тяжело написать, и она делает код длиннее и труднее для чтения. Но игнорирование ошибок просто заметает проблему под ковёр, где ничего не подозревающий пользователь однажды её обнаружит.

**Правило 6. Разделяйте код на короткие, обособленные части.**

Каждый метод, функция или блок кода должн умещаться в обычном экранном окне (25-50 строк). Если получилось длиннее, разделите на более короткие куски. Даже внутри метода разделяйте длинный код на блоки, суть которых вы можете описать в комментарии в начале каждого блока.

Более того, каждый класс, модуль, файл или процесс должен выполнять определённый род задач. Если часть кода выполняет совершенно разнородные задачи, то разделите его соответственно.

**Правило 7. Используйте API фреймворков и сторонние** **библиотеки.**

Изучите, какие функции доступны с помощью API вашего фреймворка. а также что могут делать развитые сторонние библиотеки. Если библиотеки поддерживаются вашим системным менеджером пакетов, то они скорее всего окажутся хорошим выбором. Используйте код, удерживающий от желания изобретать колесо (при том бесполезной квадратной формы).

**Правило 8. Не переусердствуйте с проектированием.**

Проектируйте только то, что актуально сейчас. Ваш код можно делать довольно обобщённым, чтобы он поддерживал дальнейшее развитие, но только в том случае, если он не становится от этого слишком сложным. Не создавайте параметризованные классы, фабрики, глубокие иерархии и скрытые интерфейсы для решения проблем, которых даже не существует — вы не можете угадать, что случится завтра. С другой стороны, когда структура кода не подходит под задачу, не стесняйтесь рефакторить его.

**Правило 9. Будьте последовательны.**

Делайте одинаковые вещи одинаковым образом. Если вы разрабатываете метод, функциональность которого похожа на функциональность уже существующего, то используйте похожее имя, похожий порядок параметров и схожую структура тела. То же самое относится и к классам. Создавайте похожие поля и методы, делайте им похожие интерфейсы, и сопоставляйте новые имена уже существующим в похожих классах.

Ваш код должен соответствовать соглашениям вашего фреймворка. Например, хорошей практикой является делать диапазоны полуоткрытыми: закрытыми (включающими) слева (в начале диапазона) и открытыми (исключающими) справа (в конце). Если для конкретного случая нет соглашений, то сделайте выбор и фанатично придерживайтесь его.

**Правило 10. Избегайте проблем с безопасностью.**

Современный код редко работает изолированно. У него есть неизбежный риск стать мишенью атак. Они необязательно должны приходить из интернета; атака может происходить через входные данные вашего приложения. В зависимости от вашего языка программирования и предметной области, вам возможно стоит побеспокоиться о переполнении буфера, кросс-сайтовых сценариях, SQL-инъекциях и прочих подобных проблемах. Изучите эти проблемы, и избегайте их в коде. Это не сложно.

**Правило 11. Используйте эффективные структуры данных и алгоритмы.**

Простой код часто легче сопровождать, чем такой же, но изменённый ради эффективности. К счастью, вы можете совмещать сопровождаемость и эффективность, используя структуры данных и алгоритмы, которые даёт ваш фреймворк. Используйте map, set, vector и алгоритмы, которые работают с ними. Благодаря этому ваш код станет чище, быстрее, более масштабируемым и более экономным с памятью. Например, если вы сохраните тысячу значений в отсортированном множестве, то операция пересечения найдёт общие элементы с другим множеством за такое же число операций, а не за миллион сравнений.

**Правило 12. Используйте Unit-тесты.**

Сложность современного ПО делает его установку дороже, а тестирование труднее. Продуктивным подходом будет сопровождение каждого куска кода тестами, которые проверяют корректность его работы. Этот подход упрощает отладку, т.к. он позволяет обнаружить ошибки раньше. Unit-тестирование необходимо, когда вы программируете на языках с динамической типизацией, как Python и JavaScript, потому что они отлавливают любые ошибки только на этапе исполнения, в то время как языки со статической типизацией наподобие Java, C# и C++ могут поймать часть из них во время компиляции. Unit-тестирование также позволяет рефакторить код уверенно. Вы можете использовать XUnit для упрощения написания тестов и автоматизации их запуска.

**Правило 13. Сохраняйте код портируемым.**

Если у вас нет особой причины, не используйте функциональность, доступную только на определённой платформе. Не полагайтесь на то, что определённые типы данных (как integer, указатели и временные метки) будут иметь конкретную длину (например, 32 бита), потому что этот параметр отличается на разных платформах. Храните сообщения программы отдельно от кода и на зашивайте параметры, соответствующие определённой культуре (например, разделители дробной и целой части или формат даты). Соглашения нужны для того, чтобы код мог запускаться в разных странах, так что сделайте локализацию настолько безболезненной, насколько это возможно.

**Правило 14. Делайте свой код собираемым.**

Простая команда должна собирать ваш код в форму, готовую к распространению. Команда должна позволять вам быстро выполнять сборку и запускать необходимые тесты. Для достижения этой цели используйте средства автоматической сборки наподобие Make, Apache Maven, или Ant. В идеале, вы должны установить интеграционную систему, которая будет проверять, собирать и тестировать ваш код при любом изменении.

**Правило 15. Размещайте всё в системе контроля версий.**

Все ваши элементы — код, документация, исходники инструментов, сборочные скрипты, тестовые данные — должны быть в системе контроля версий. Git и GitHub делают эту задачу дешёвой и беспроблемной. Но вам также доступны и многие другие мощные инструменты и сервисы. Вы должны быть способны собрать и протестировать вашу программу на сконфигурированной системе, просто скачав её с репозитория.

Сделав эти 15 правил частью вашей ежедневной практики, вы в конце концов создадите код, который легче читать, который хорошо протестирован, с большей вероятностью запустится корректно и который будет гораздо проще изменить, когда придёт время. Вы также убережёте себя и ваших пользователей от большого числа головных болей.

Пример программного кода без соблюдения правил оформления кода (язык С++):

#include <iostream>

#include <iomanip>

#include <stdlib.h>

#include <time.h>

void printMatrix(int[][10], const int);

using namespace std;

int main()

{

const int SIZE=10;

int matrix[SIZE][SIZE];

int temp;

srand(time(NULL));

for(int i=0;i<SIZE;i++)

for(int j=0;j<SIZE;j++)

matrix[i][j]=1+rand()%100;

for(int N=1;N<SIZE\*SIZE;N++)

{

for(int i=0;i<SIZE;i++)

{

for(int j=0;j<SIZE-1;j++)

{

if(matrix[i][j+1]<matrix[i][j])

{

temp=matrix[i][j+1];

matrix[i][j+1]=matrix[i][j];

matrix[i][j]=temp;

}}}}

printMatrix(matrix,SIZE);

return 0;

}

void printMatrix(int mx[][10],const int SIZE)

{

for(int i=0;i<SIZE;i++)

{

cout << endl;

for(int j=0;j<SIZE;j++)

cout << setw(4) << mx[i][j];

cout << endl;

}}

Пример программного кода с соблюдением правил оформления кода (язык C++):

// SortArray.cpp Сортировка строк двумерного массива методом пузырька

// Создан 25.02.2015 Сидоров С.С. Заявка №12344

#include <iostream>

#include <iomanip>

#include <stdlib.h>

#include <time.h>

//печать массива

void printMatrix

(int[][10], // сортируемый массив

const int // размерность массива

);

using nameSpace std; //установка стандартного пространства имен

int main()

{

const int SIZE = 10; //размерность массива

int matrix[SIZE][SIZE]; //определяем массив

int temp; //временная переменная

srand(time(NULL));//определяем исходную точку для генератора случайных чисел

//заполняем массив случайным образом

for(int i = 0; i < SIZE; i++)

for(int j = 0; j < SIZE; j++)

matrix[i][j] = 1 + rand() % 100;

//сортируем пузырьком

for(int N = 1; N < SIZE \* SIZE; N++)

{

for(int i = 0; i < SIZE; i++)

{

for(int j = 0; j < SIZE - 1; j++)

{

//если нужно скорректировать порядок сортировки

if(matrix[i][j + 1] < matrix[i][j])

{

//делаем обмен элементов массива местами

temp = matrix[i][j + 1];

matrix[i][j + 1] = matrix[i][j];

matrix[i][j] = temp;

}

}// for(int j = 0; j < SIZE - 1; j++)

//for(int N = 1; N < SIZE \* SIZE; N++)

//выводим массив на печать

printMatrix(matrix, SIZE);

return 0;

}// end of main SortArray.cpp

//печать массива

void printMatrix(int varMX[][10], const int SIZE)

{

for(int i = 0; i < SIZE; i++)

{

cout << endl;

for(int j = 0; j < SIZE; j++)//печать j-массива на одной строке

cout << setw(4) << varMX[i][j];

cout << endl; // каждый j-массив на отдельной строке

}// for(int i = 0; i < SIZE; i++)

}// end of printMatrix

// end of SortArray.cpp

# **Лекция 6. Описание требований: унифицированный язык моделирования- UML.**

# **Лекция 7. Диаграммы UML.**

# **Лекция 8. Описание и оформление требований (спецификация).**

**Спецификация требований программного обеспечения** (англ. software requirements specification, SRS) — структурированный набор требований (функциональность, производительность, конструктивные ограничения и атрибуты) к программному обеспечению и его внешним интерфейсам. (Определение на основе IEEE Std 1012:2004) Предназначен для того, чтобы установить базу для соглашения между заказчиком и разработчиком (или подрядчиками) о том, как должен функционировать программный продукт.

Может включать ряд пользовательских сценариев (англ. use cases), которые описывают варианты взаимодействия между пользователями и программным обеспечением.

Пользовательские сценарии являются средством представления функциональных требований. В дополнение к пользовательским сценариям, спецификация также содержит нефункциональные требования, которые налагают ограничения на дизайн или реализацию (такие как требования производительности, стандарты качества, или проектные ограничения).

Каждая организация, специализирующаяся на разработке ПО, должна принять один или несколько стандартных шаблонов спецификации требований к ПО для использования в проектах. Существуют различные шаблоны спецификаций. Если вы беретесь за проекты различных типов и размеров, от конструирования новой объемной системы до небольших улучшений уже работающих систем, позаботьтесь для проектов каждого крупного класса завести отдельный шаблон спецификации.

Шаблон спецификации требований к ПО

Шаблон спецификации требований к ПО

Каждая организация, специализирующаяся на разработке ПО, должна принять один или несколько стандартных шаблонов спецификации требований к ПО для использования в проектах. Существуют различные шаблоны спецификаций. Если вы беретесь за проекты различных типов и размеров, от конструирования новой объемной системы до небольших улучшений уже работающих систем, позаботьтесь для проектов каждого крупного класса завести отдельный шаблон спецификации.

# **8.1 Шаблон спецификации требований**

Приведенный шаблон подходит для многих проектов. Состоит он из следующих разделов:

**1. Введение**

1.1 Назначение

1.2 Соглашения, принятые в документах

1.3 Границы проекта

1.4 Ссылки

**2. Общее описание**

2.1 Общий взгляд на продукт

2.2 Классы и характеристики пользователей

2.3 Операционная среда

2.4 Ограничения дизайна и реализации

2.5 Предположения и зависимости

**3. Функции системы**

3.x Функция системы X

3.x.1 Описание

3.x.2 Функциональные требования

**4. Требования к данным**

4.1 Логическая модель данных

4.2 Словарь данных

4.3 Отчеты

4.4 Получение, целостность, хранение и утилизация данных

**5. Требования к внешним интерфейсам**

5.1 Пользовательские интерфейсы

5.2 Интерфейсы ПО

5.3 Интерфейсы оборудования

5.4 Коммуникационные интерфейсы

**6. Атрибуты качества**

6.1 Удобство использования

6.2 Производительность

6.3 Безопасность

6.4 Техника безопасности

6.x [Другие]

**7. Требования по интернационализации и локализации**

**8. Остальные требования**

**Приложение A. Словарь терминов**

**Приложение Б. Модели анализа**

Иногда фрагмент информации логически подходит для нескольких разделов шаблона. Выберите один раздел и используйте именно его для информации такого типа в своем проекте. Не дублируйте информацию в нескольких разделах, даже если логически она ложится в эти разделы. Используйте перекрестные ссылки и гиперссылки, чтобы облегчить читателям поиск нужной информации.

При создании документов требований применяйте эффективные приемы и средства управления версиями, чтобы все читатели четко понимали, какую версию они читают в тот или иной момент времени. Ведите журнал изменений, в котором фиксируются суть изменений, автор, дата и причина.

# **8.2 Структура спецификации требований**

**1. Введение**

Введение представляет собой обзор, помогающий читателям разобраться в структуре и принципе использования спецификации требований к ПО.

**1.1 Назначение**

Определите продукт или приложение, требования для которого указаны в этом документе, в том числе редакцию или номер выпуска. Если эта спецификация требований к ПО относится только к части системы, идентифицируйте эту часть или подсистему. Опишите типы читателей, которым адресован этот документ, например разработчикам, менеджерам проектов, маркетологам, пользователям, тестировщикам или составителям документации.

**1.2 Соглашения, принятые в документах**

Опишите все стандарты или типографические соглашения, включая значение стилей текста, особенности выделения или нотацию. Если вы нумеруете требования вручную, можно определить принятый формат на случай, если кому-нибудь позже понадобиться добавить требование.

**1.3 Границы проекта**

Кратко опишите ПО и его назначение. Покажите, как связан продукт с пользователями или корпоративными целями, а также с бизнес-целями и стратегиями. Если имеется отдельный документ о концепции и границах проекта, не повторяйте его содержимое, а просто сошлитесь на него. Если спецификацию требований к ПО предполагается разрабатывать постепенно, она должна содержать собственное положение о концепции и границах продукта в качестве подраздела долгосрочной стратегической концепции. Можно предоставить высокоуровневую сводку главной функциональности выпуска или функций, которые он должен выполнять.

**1.4 Ссылки**

Перечислите все документы или другие ресурсы, на которые вы ссылаетесь в этой спецификации, в том числе гиперссылки на них, если их местоположение меняться не будет. Это могут быть руководства по стилям пользовательского интерфейса, контракты, стандарты, спецификации к системным требованиям, спецификации интерфейса и спецификации требований к ПО связанных продуктов. Объем информации должен быть достаточным для того, чтобы пользователь сумел при необходимости получить доступ к каждому указанному материалу, а именно: название, имя автора, номер версии, дата, источник, место хранения или URL-адрес.

**2. Общее описание**

В этом разделе представлен общий обзор продукта и среды, в которой он будет применяться, предполагаемая пользовательская аудитория, а также известные ограничения, предположения и зависимости.

**2.1 Общий взгляд на продукт**

Опишите контекст и происхождение продукта. Поясните, является он новым членом растущего семейства продуктов, новой версией существующей системы, заменой существующего приложения или совершенно новым продуктом. Если спецификация требований определяет компонент более крупной системы, укажите, как это ПО соотносится со всей системой и определите основные интерфейсы между ними. Неплохо также включить визуальные модели, такие как контекстную диаграмму или карту экосистемы, чтобы показать взаимосвязь продукта с другими системами.

**2.2 Классы и характеристики пользователей**

Определите различные классы пользователей, которые, как предполагается, будут работать с вашим продуктом, и опишите их соответствующие характеристики. Некоторые требования могут относиться только к определенным классам пользователей. Определите привилегированные классы пользователей. Классы пользователей представляют подмножество заинтересованных в проекте лиц, их описание приводится в документе концепции и границ проекта. Описания классов пользователей являются повторно используемым ресурсом. Если есть главный каталог классов пользователей, можно включить описания классов пользователей, просто указав их в каталоге, не дублируя информацию.

**2.3 Операционная среда**

Опишите рабочую среду, в которой будет работать ПО, включая аппаратную платформу, операционные системы и их версии, а также географическое местоположение пользователей, серверов и баз данных вместе с организациями, в которых располагаются соответствующие базы данных, серверы и веб-сайты. Перечислите все остальные компоненты ПО или приложения, с которыми система должна быть совместима. Если в связи с разработкой новой системы нужно произвести значительную работу с технической инфраструктурой, стоит подумать о создании отдельных требований к инфраструктуре, в которой детально изложить подробности этой работы.

**2.4 Ограничения дизайна и реализации**

Бывает, что нужно использовать вполне определенный язык программирования, определенную библиотеку, на разработку которой уже потрачено время, и т. п. Опишите все факторы, которые ограничат возможности, доступные разработчикам, и логически обоснуйте каждое положение. Требования, которые включают или написаны в форме идей по решению, а не потребностей накладывают ограничения на дизайн, часто неоправданные, поэтому за этим надо следить.

**2.5 Предположения и зависимости**

Предположение (assumption) — это утверждение, которое предполагается верным в отсутствие знаний или доказательств иного. Проблемы возможны в том случае, если предположение неверны, устарели, не находятся в совместном использовании или изменяются, поэтому определенные предположения можно отнести к группе рисков проекта. Один читатель спецификации требований к ПО может считать, что продукт будут соответствовать особому стандарту пользовательского интерфейса, тогда как другой предположит нечто совершенно иное. Разработчик может думать, что определенный набор функций написан специально для этого приложения, бизнес-аналитик — что он будет взят из предыдущего проекта, а менеджер проекта — что предполагается приобрести коммерческую библиотеку функций. Включаемые здесь предположения относятся к системной функциональности; предположения относящиеся к бизнесу представлены в документе концепции и границ проекта. Определите все зависимости проекта или создаваемой системы от внешних факторов или компонентов вне ее контроля. Например, до установки продукта может требоваться установить Microsoft .NET Framework 4.5 или более позднюю версию — это зависимость.

**3. Функции системы**

Приведенный здесь шаблон структурирован по функциям системы — это еще один способ систематизации функциональных требований. Другие методы классификации — по функциональным областям, рабочим потокам, вариантам использования, режимам работы, классам пользователей, стимулам и реакциям. Возможны также иерархические комбинации этих элементов, например варианты использования внутри классов пользователей. Не существует единственно правильного метода организации; выберите тот, при котором пользователям будет легче понять предполагаемые возможности продукта.

Опишем схему функций на примере.

**3.x Функция системы X**

Опишите название особенности несколькими словами, например «3.1 Проверка правописания». Так же назовите подразделы с 3.x.1 по 3.x.3 для каждой функции системы.

**3.x.1 Описание**

Кратко опишите функцию системы и укажите ее приоритет: высокий, средний или низкий приоритетом. Приоритеты являются динамичной характеристикой, они могут изменяться в ходе проекта. Если вы используете средство управления требованиями, определите атрибут требований для обозначения приоритета.

**3.x.2 Функциональные требования**

Перечислите по пунктам конкретные функциональные требования, которые связаны с этой функцией. Именно эти функции ПО нужно реализовать, чтобы пользователь мог использовать сервисы этой функции или реализовать вариант использования. Опишите, как продукт должен реагировать на ожидаемые ошибки, неправильный ввод информации или неверные действия. Присвойте каждому функциональному требованию уникальное имя. При использовании средства управления требованиями можно создать много атрибутов для каждого функционального требования, таких как основание, источник и состояние.

**4. Требования к данным**

Ценность информационных систем заключается в том, что они предоставляют возможность манипулировать данными. Используйте этот раздел шаблона для описания различных аспектов данных, которые будет потреблять система в качестве входной информации, как-то обрабатывать и возвращать в виде выходной информации.

**4.1 Логическая модель данных**

Модель данных это визуальное представление объектов и наборов данных, которые будет обрабатывать система, а также отношений между ними. Существует много видов нотации для моделирования данных, в том числе диаграммы отношений «сущность–связь» и диаграммы классов UML. Можно включить модель данных для бизнес-операций, выполняемых системой или логическое представление данных, с которыми будет работать система. Это не то же самое, что модель данных реализации, которая реализуется в виде дизайна базы данных.

**4.2 Словарь данных**

Словарь данных определяет состав структур данных, а также их значение, тип данных, длину, формат и разрешенные значения элементов данных, из которых состоят эти структуры. Серийные средства моделирования данных часто включают компонент-словарь данных. Во многих случаях словарь данных лучше хранить как отдельный артефакт, не внедряя его в спецификацию требований к ПО. Это повышает возможности повторного использования в других проектах.

**4.3 Отчеты**

Если приложение будет генерировать отчеты, перечислите их здесь и опишите их характеристики. Если отчет должен соответствовать определенному готовому макету, можно указать это как ограничение, возможно с примером. В противном случае сосредоточьтесь на логических описаниях, порядке сортировки, уровнях суммирования и т. п., отложив подробный макет до этапа дизайна.

**4.4 Получение, целостность, хранение и утилизация данных**

Если это важно, опишите, как получают и обслуживают данные. Например, при открытии канала номенклатуры данных может требоваться первым делом выполнить начальный дамп всей номенклатуры данных в принимающую систему, а после этого использовать каналы для передачи только изменений. Укажите все требования, относящиеся к защите целостности данных системы. Укажите все процедуры, которые могут потребоваться, например резервное копирование, создание контрольных точек, зеркальное отображение или проверка корректности данных. Укажите политики, которые должна применять система для хранения или утилизации данных, в том числе временных данных, метаданных, остаточных данных (таких как удаленные записи), данных в кеше, локальных копий, архивов и промежуточных архивов.

**5. Требования к внешним интерфейсам**

В этом разделе указывается информация, которая гарантирует, что система будет правильно взаимодействовать с пользователями и компонентам внешнего оборудования и ПО. Выработка согласия по внешнему и внутреннему интерфейсу системы признано оптимальным приемом в области разработки ПО (Brown, 1996). В сложной системе с множеством подкомпонентов следует использовать раздельные спецификации для интерфейсов или спецификацию системной архитектуры. В документацию по интерфейсу можно включить ссылки на материал из других документов. Например, ссылка может указать на руководство по работе с устройством, где перечислены коды ошибок, которые устройство может отправить программе.

|  |
| --- |
| **Войны интерфейсов**  Две команды разработчиков ПО объединились для создания флагманского продукта компании A. Datum Corporation. Команда, отвечающая за базу знаний, создала сложное ядро анализа на C++, а команда, отвечающая за приложения, реализовала пользовательский интерфейс на Java. Подсистемы взаимодействовали между собой посредством API. К сожалению, команда, отвечающая за базу знаний, периодически модифицировала API, в результате чего систему не удавалось собрать и запустить на выполнение должным образом. Команде, отвечающей за приложения, требовалось несколько часов, чтобы распознать все проблемы и определить основную причину — изменение API. Эти изменения не согласовывались, не доводились до сведения всех заинтересованных в проекте лиц и не были скоординированы с соответствующими модификациями в коде на Java. Изменение интерфейса обязательно требует уведомления об этом людей, группы или системы на другой стороне этого интерфейса. Интерфейс скрепляет компоненты вашей системы — включая пользователей, поэтому необходимо документировать детали интерфейса и синхронизировать модификации в процессе управления изменениями в проекте. |

**5.1 Пользовательские интерфейсы**

Опишите логические характеристики каждого пользовательского интерфейса, который необходим системе. Некоторые особенные характеристики пользовательских интерфейсов могут упоминаться в разделе «6.1 Удобство использования». Некоторые из них перечислены здесь:

* + ссылки на стандарты графического интерфейса пользователей или стилевые рекомендации для семейства продуктов, которые необходимо соблюдать;
  + стандарты шрифтов, значков, названий кнопок, изображений, цветовых схем, последовательностей полей вкладок, часто используемых элементов управления, графики фирменного стиля, уведомления о зарегистрированных товарных знаках и о конфиденциальности и т.п.;
  + размер и конфигурация экрана или ограничения разрешения;
  + стандартные кнопки, функции или ссылки перемещения, одинаковые для всех экранов, например кнопка справки;
  + сочетания клавиш;
  + стандарты отображения и текста сообщений;
  + стандарты проверки данных (такие как ограничения на вводимые значения и когда нужно проверять содержимое полей);
  + стандарты конфигурации интерфейса для упрощения локализации ПО;
  + специальные возможности для пользователей с проблемами со зрением, различением цвета и другими ограничениями.

**5.2 Интерфейсы ПО**

Опишите связи продукта и других компонентов ПО (идентифицированные по имени и версии), в том числе другие приложения, базы данных, операционные системы, средства, библиотеки, веб-сайты и интегрированные серийные компоненты. Укажите назначение, форматы и содержимое сообщений, данных и контрольных значений, обмен которыми происходит между компонентами ПО. Опишите соответствия между входными и выходными данными между системами и все преобразования, которые должны происходить с данными при перемещении между системами. Опишите службы, необходимые внешним компонентам ПО, и природу взаимодействия между компонентами. Определите данные, которыми будут обмениваться и к которым будут иметь общий доступ компоненты ПО. Определите нефункциональные требования, влияющие на интерфейс, такие как уровни обслуживания для времени и частоты отклика или меры и ограничения безопасности. Часть этой информации может быть определена как требования к данным в разделе 4 или как требования к взаимодействию в разделе «6. Атрибуты качества».

**5.3 Интерфейсы оборудования**

Опишите характеристики каждого интерфейса между компонентами ПО и оборудования системы. В описание могут входить типы поддерживаемых устройств, взаимодействия данных и элементов управлений между ПО и оборудованием, а также протоколы взаимодействия, которые будут использоваться. Перечислите входные и выходные данные, их формат, разрешенные значения или их диапазоны, а также все временные характеристики, о которых должны знать разработчики. Если такой информации очень много, лучше создать отдельный документ спецификации интерфейса.

**5.4 Коммуникационные интерфейсы**

Укажите требования для любых функций взаимодействия, которые будут использоваться продуктом, включая электронную почту, веб-браузер, сетевые протоколы и электронные формы. Определите соответствующие форматы сообщений. Опишите особенности безопасности взаимодействия или шифрования, скорости передачи данных и механизмов согласования и синхронизации. Укажите все ограничения этих интерфейсов, например допустимость тех или иных типов вложений в сообщениях электронной почты.

**6. Атрибуты качества**

В этом разделе описываются нефункциональные требования помимо ограничений, описанных в разделе 2.4, и требований к внешним интерфейсам, описанным в разделе 5. Эти характеристики должны быть точно определены и поддаваться проверке и измерению. Укажите относительные приоритеты различных атрибутов, например приоритет простоты использования над легкостью изучения или приоритет безопасности над производительностью. Необходимые степени качества удается гораздо эффективнее описать с помощью подробных нотаций спецификации, таких, как Planguage, чем с помощью простых описательных утверждений.

**6.1 Удобство использования**

Требования к удобству использования подразумевают легкость изучения, простоту использования, предотвращение ошибок и восстановление, эффективности взаимодействия и специальные возможности. Указанные в этом разделе требования к удобству использования помогут дизайнеру интерфейсов создать максимально удобную для пользователя рабочую среду.

**6.2 Производительность**

Укажите конкретные требования к производительности для различных системных операций. Если у различных функциональных требований или функций имеются разные требования к производительности, то следует указывать задачи, связанные с производительностью, там же, в разделе соответствующих функциональных требований, а не включать их все в этот раздел.

**6.3 Безопасность**

Укажите все требования, касающиеся безопасности или конфиденциальности, которые ограничивают доступ или возможности использования продукта. Это может быть физическая безопасность, а также защита данных или ПО. Источником требований к безопасности, как правило, являются бизнес-правила, поэтому определите политики или положения, касающиеся защиты или конфиденциальности, которым продукт должен соответствовать. Если они задокументированы и хранилище бизнес-правил, просто сошлитесь на них.

**6.4 Техника безопасности**

В этом разделе укажите требования, связанные с возможными потерями, повреждениями или ущербом, которые могут быть результатом использования продукта. Определите меры безопасности или упреждающие действия, которые можно предпринять, так же как и потенциально опасные действия, которые можно предотвратить. Определите сертификаты по безопасности, политики или положения, которым продукт должен соответствовать.

**6.x [Другие]**

Создайте в спецификации требований к ПО отдельный раздел для каждого дополнительного атрибута качества продукта, чтобы описать характеристики, которые будут важны для клиентов или для разработчиков и людей, ответственных за поддержку. Это может быть доступность, возможность установки, целостность, возможность модификации, переносимость, надежность, устойчивость, масштабируемость и контролируемость.

**7. Требования по интернационализации и локализации**

Требования по интернационализации и локализации обеспечивают возможность использовать продукт в других странах, региональных стандартах и географических районах, отличающихся от тех, в которых он был создан. Такие требования могут быть направлены на разрешение различий в валютах, форматировании дат, чисел, адресов и телефонных номеров, языках, в том числе различных вариантах одного языка (например, американского и британского вариантов английского), используемых символах и наборах символов, личных именах и фамилиях, часовых поясах, международных нормативных актах и законах, культурных и политических традициях, размере используемой бумаги, единицах веса и меры, электрическом напряжении и конфигурации электрических разъемов и во многом другом. Требования по интернационализации и локализации вполне могут повторно использоваться во многих проектах.

**8. [Остальные требования]**

Определите все другие требования, которые еще не были описаны в спецификации требований к ПО. Примером могут служить юридические, законодательные или финансовые требования и требования стандартов, требования к установке, конфигурированию, запуску и остановке продукта, а также к журналированию, мониторингу и контрольному следую. Вместо того чтобы сливать всю эту информацию в один раздел, добавьте любые новые разделы к шаблону вашего проекта. Пропустите этот раздел, если все необходимые требования уже расписаны в других разделах. В этот раздел можно включить требования к переходу, которые необходимы для миграции с предыдущей системы на новую, если они относятся к создаваемому ПО (например, программы преобразования данных), в противном случае их можно включить в план управления проектом (например, разработка обучающих материалов или поставка).

**Приложение A. Словарь терминов**

Определите все специальные термины, которые читателю необходимо знать для правильного понимания спецификации требований к ПО, включая сокращения и аббревиатуры. Расшифруйте каждое сокращение и приведите его определение. Подумайте о создании расширенного общекорпоративного словаря для нескольких проектов, который включает по ссылке все термины, относящиеся к данному проекту. В этом случае в спецификации требований к ПО будут определены только те термины, которые относятся лишь к данному проекту и которых нет в общекорпоративном словаре. Заметьте, что определения данных находятся в словаре данных, а не терминов.

**Приложение Б. Модели анализа**

В этом необязательном разделе описывается, а точнее напоминается, о таких моделях анализа, как диаграммы потоков данных, деревья функций, диаграммы переходов состояния и диаграммы «сущность-связь». Часто читателю удобнее, когда определенные модели внедрены в соответствующие разделы спецификации, а не собраны скопом в конце.

# **Лекция 9. Анализ требований и стратегии выбора решения.**

Перед специалистом IT-отдела, либо независимым консультантом, которому поручен выбор решения в области автоматизации предприятия, стоит нелёгкая задача. Ведь на рынке представлены сотни решений в области автоматизации, и, чтобы хотя бы бегло ознакомиться с каждым из них, может потребоваться несколько человеко-лет.

На практике, конечно, вряд ли путь подробного изучения всех известных на рынке решений можно рассматривать всерьёз. В простейшем случае рассматриваются аналитические обзоры, подготовленные независимыми экспертами, оцениваются финансовые возможности предприятия внедрения и на третейский суд инвестора предоставляются 2-3 решения.

Чтобы сделать выбор обоснованным, используются стратегии выбора решения. Основные из них:

* анализ требований;
* анализ несоответствия;
* подход на основе "лучших практик".

# **9.1 Анализ требований**

Рациональным приёмом, позволяющим снизить затраты на подготовку вариантов решения, а заодно снизить риски, является формирование документа требований (не только ко вновь создаваемому, но и к выбираемому продукту). Тем самым, часть работы можно переложить на представителей маркетинговых отделов вендоров – пусть они объяснят и продемонстрируют, как на практике реализуются Ваши требования.

Практическое применение методов анализа требований, применительно к задаче выбора покупного решения, требует ответа на следующие вопросы:

* рамки проекта;
* широта анализа требований;
* глубина проработки требований;
* требуемые ресурсы.

**Рамки проекта.** Какие бизнес-процессы, подразделения, отделы предприятия необходимо автоматизировать? Практика внедрения ERP-систем показывает, что даже на самых передовых в информационном смысле предприятиях степень охвата бизнеса информационными технологиями редко превышает порог в 90%. Возможно, крупномасштабный проект автоматизации следует разбить на несколько этапов, начав, допустим, с планки в 30%. Другая стратегия – "всё и сразу" — предполагает попытку сразу выйти на максимально возможный охват функций.

**Широта анализа требований.** Сколько требований вы в состоянии сформулировать в течение разумного промежутка времени? Сколько времени уйдёт у вендоров на анализ документа требований, который вы подготовите? Д. О’Лири приводит следующий порядок: документ описания требований для предприятия с годовым оборотом в $40 млн. содержит около 1000 позиций (требований).

**Глубина проработки требований.** Какую выбрать степень детализации требований? Уровень документа "Концепция" вряд ли окажется достаточным для серьёзного рассмотрения. Необходимо отражать как функциональные, так и нефункциональные требования. Можно остановиться на кратких формулировках функций (лаконично, трудно проверяемо). Можно перейти на язык сценариев (появляется возможность доскональной проверки, но это потребует значительных ресурсов). Хорош вариант с комбинацией этих способов описания: критичный функционал описать в форме сценариев, остальное – в виде функций.

**Требуемые ресурсы.** Ключевой ресурс предприятия – человеческий ресурс. Найдут ли топ-менеджеры предприятия время на скрупулёзную оценку требований к системе? Насколько хорошо смогут сформировать требования руководители линейных подразделений? Насколько удастся задействовать ресурс поставщиков решений? Стоит ли привлекать внешних консультантов?

Какова цена обследования? Как быстро удастся сформировать требования? Какое время следует выделить на получение ответов от вендоров, на анализ этих ответов перед окончательным выбором?

# **9.2 Анализ несоответствия**

Анализ несоответствия при выборе КИС базируется на классических методах бизнес-анализа, предполагающих построения моделей "как есть", "как надо" и переходной модели, показывающей путь реформирования предприятия .

Нюанс заключается в том, что в данном случае анализ "как есть" применяется не к бизнес-системе в чистом виде, а к её комбинации с имеющейся автоматизированной системой. Он призван высветить слабые стороны и имеющегося решения, и связанные с ним проблемы бизнеса.

Анализ "как надо" может производиться по одному из следующих сценариев: "реинжиниринга чистого листа" и "реинжиниринга, запускаемого технологией".

В первом случае используются классические подходы к реинжинирингу предприятий в комбинации с методами анализа и формирования требований.

Во втором случае речь идёт о реинжиниринге под конкретное ERP-решение. В этом случае основой для пересмотра действующих бизнес-процессов и уровня их автоматизации являются "лучшие практики", заложенные в соответствующей ERP-системе.

Как и в стратегии анализа требований, стратегия анализа несоответствия оставляет множество подводных камней и вопросов. Допустим, мы составили модель "надо" и рассматриваем спецификации конкурирующих ERP-систем. Как сопоставить эти два документа? Как измерить степень соответствия – по количеству функций? Где гарантии того, что функции, называющиеся одинаково в рассматриваемых документах, действительно одинаково работают?

Альтернативой рассмотренным выше стратегиям, позволяющей снизить степень неопределённости принятия решений, в определённой мере является подход на основе лучших практик .

# **9.3 Подход на основе лучших практик**

Подход основан на отказе от моделей "как есть". Предлагается не "зацикливаться" на существующих на предприятии бизнес-процессах, а запустить процесс реинжиниринга, основываясь на "лучших практиках" внедряемого ERP-решения. Основные доводы за применение данного подхода .

1. Каждый пакет ERP соответствует нуждам организации. Выбранная ERP-система имеет приблизительно тот же набор лучших практик, что и любая другая и все они примерно эквивалентны.

2. Копировать существующие процессы не имеет смысла. Организация должна осуществлять автоматизацию, основываясь на "реинжиниринговом потенциале" ERP-системы.

3. Априорный анализ лучших практик не должен использоваться. Анализ лучших практик должен осуществляться в контексте конкретной части ПО выбранной ERP-системы.

4. Следует не "загибать решение под существующий бизнес", а напротив, реорганизовать бизнес на основе лучших практик так, чтобы изменения в ERP-системы были минимальны. Это позволит снизить цену внедрения и цену владения ERP-системами.

# **9.4 Процесс выбора решения**

Выбор решения для построения на предприятии корпоративной АИС – очень ответственный процесс, связанный с вопросами защиты инвестиций и выживания на рынке. Значительное количество проектов по внедрению ERP-систем заканчивается провалом, особенно это касается российской практики. Более того, на Западе существуют прецеденты, когда предприятия, поставившие "на карту ERP" своё благосостояние и связавшие с ними планы развития, попросту обанкротились, не справившись с задачей "реинжиниринга под ERP".

Поэтому задачу выбора решения следует рассматривать в рамках проектного и процессного подходов, со всеми вытекающими отсюда последствиями.

В формально проработанных процессах внедрения решений недостатка нет. Как правило, они разрабатываются и поставляются вендорами вместе с самими решениями. Вопросы организации выбора решений проработаны в литературе значительно хуже.

Ниже рассмотрен практический процесс, принятый компанией Chesapeake Display & Packaging при выборе ERP-решения – процесс быстрого выбора для быстрого внедрения на базе обзора, представленного в.

Процесс организован достаточно просто. Он предусматривает выполнение следующих этапов:

* cформировать команду "выборщиков";
* организовать демонстрацию КИС поставщиками;
* осуществить предварительный отбор;
* осуществить выбор.

**Сформировать команду.** Команда должна быть небольшой и тщательно подобранной. Критерии отбора – знание бизнеса и бизнес-процессов; включение людей с различными взглядами, сочетание узкоспециализированных специалистов и специалистов широкого профиля. Ограничение на размеры команды – не более 10 человек.

**Организовать демонстрацию КИС поставщиками.** Выбор ограниченного количества производителей высококачественных КИС. Предложение выбранным производителям подготовить демонстрацию для команды "выборщиков". Ограничение доступа представителей производителей к членам команды до момента демонстрации. Срок подготовки – 3 недели. Продолжительность демонстрации – 2 дня. Свобода выбора для вендора оборудования и СУБД.

**Осуществить предварительный отбор.** Поставщикам высказываются следующие требования:

* показать, что КИС сможет работать с вашим бизнесом;
* показать, что вы сможете внедрить его в течение требуемого времени;
* показать своё понимание отрасли.

После проведения демонстрации на основе указанных требований осуществляется выбор тройки лидеров.

**Осуществить выбор.** Выбор осуществляется на основе голосования, большинством голосов. Голосование производится на основе двух факторов:

* наиболее подходящие функциональные возможности;
* лучший персонал по внедрению.

Каждая из опций ПО оценивается в трёхбалльной шкале (3 – наивысший балл). Оценки осуществляются членами команды по группам компетенции.

Анализ требований – пожалуй, самая важная часть программной инженерии. Недопонимание этого, недостаточное внимание к вопросам выявления, документирования и управления требованиями – одна из наиболее существенных причин возникновения проблем при выполнении программных проектов, зачастую приводящих к их краху.

В учебном курсе рассмотрены основные понятия, связанные с требованиями, их классификация, структура потока работ анализа требований, используемые при этом модели и техники. Раскрыта связь между анализом требований и другими потоками работ программной инженерии, показана важность и значимость требований при построении информационных систем. Помимо работы с требованиями "глазами разработчика", в последней главе рассмотрен и "взгляд заказчика" на выбор АИС для автоматизации предприятий,

Для слушателей курса, которые хотят продолжить обучение в этой области и стать "профи" в аналитике требований, в первую очередь следует порекомендовать обратиться к "классике жанра" .

Особо хочется отметить книгу Карла Вигерса – наиболее детальную работу в этой области. Для наилучшего понимания методов и техник RUP в работе с требованиями можно порекомендовать обратиться к первоисточнику, либо к его переводам Леонида Новикова, частично доступных на www.interface.ru. Для работы с требованиями в agile-команде советую ознакомиться с книгами и статьями Алистера Коберна и Мартина Фаулера.

Еще один ценный источник информации – свод знаний о программной инженерии SWEBOK, область знаний "Software Requirements" который, в свою очередь, базируется и содержит ссылки на множество международных стандартов IEEE, IEC и другие литературные источники.Ознакомившись с указанной литературой, будет несложно овладеть технической документацией конкретных программных пакетов в области управления требованиями, будь то 3SL Cradle, IBM - Rational DOORS, IBM - Rational RequisitePro или другие.

# **Лекция 10. Цели, задачи и виды тестирования. Стандарты качества программной документации. Меры и метрики.**

**Отладка ПС** - это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ.

**Тестирование ПС** - это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения этих программ. Указанный набор данных называется тестом.

**Отладка** - многократное повторение процессов: тестирования, констатирующего наличие в ПС ошибки; поиска места ошибки в программах и документации ПС; редактирования программ и документации с целью устранения обнаруженной ошибки.

Исходной информацией для тестирования является знание о том, как система должна себя вести, т.е. требования к ней или к ее отдельным частям.

Самым распространенным способом тестирования является тестирование методом черного ящика, т.е. когда реализация системы недоступна тестеровщикам, а тестируется только ее интерфейс. Часто это закрепляется и организацией коллектива, где тестеровщики являются отдельными сотрудниками и даже принципиально не общаются с разработчиками, чтобы минимально знать реализационных деталей и максимально выступить в роли проверяющей инстанции. Существует тестирование методом белого ящика, когда код программ доступен тестеровщикам и используется в качестве источника информации о системе.

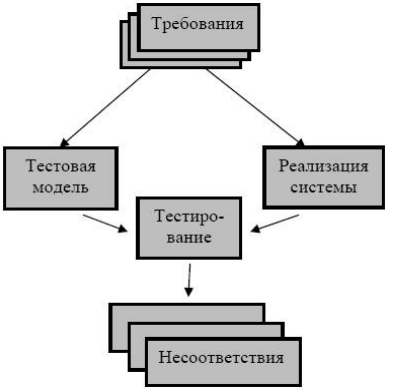


Рис. 10.1. Тестирование методом белого ящика

На рисунке видно, что на основе требований к системе создается

реализация и тестовая модель системы. Тестирование есть сопоставление двух этих представлений с целью выявить их несоответствия. Чем независимее друг от друга будут эти представления, тем больше прока от их сопоставления.

Тесты могут быть «ручными» и автоматизированными. «Ручной» тест – это последовательность действий тестеровщика, которую он может воспроизвести до возникновения ошибки. Как правило, в средствах контроля над ошибками такие последовательности действий содержатся в описании ошибки. Автоматический тест – это некоторая программа, которая воздействует на систему и проверяет то или иное ее свойство. Автоматический тест, по сравнению с «ручным», можно воспроизводить без участия человека. Можно создавать наборы тестов и прогонять их часто, например, в режиме регрессионного тестирования.

При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Нельзя гарантировать, что тестированием ПС можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи.

1. Подготовить такой набор тестов и применить его к ПС, чтобы обнаружить в нём по возможности большее число ошибок. Однако, чем дольше продолжается процесс тестирования, тем большей становится стоимость ПС.

2. Определить момент окончания отладки ПС. Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами множества различных ситуаций, возникающих при выполнении программ, и относительно редкое проявление ошибок в ПС.

Тесты проектируются на основании:

1. спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Модули рассматриваются как черные ящики;

2. текстов программ с целью протестировать все пути выполнения каждой программы ПС.

Полный перебор всех наборов входных данных, и перебор различных путей выполнения программ ПС может оказаться чрезвычайно большим, что делает их тестирование практически неосуществимым. Оптимальная стратегия проектирования тестов расположена между этими подходами, но ближе к первому варианту, так как проектирование значительной части тестов осуществляется по спецификациям.

Оптимальная стратегия проектирования тестов базируется на принципах: на каждую функцию - хотя бы один тест; на каждую область и на каждую границу изменения входной величины - хотя бы один тест; на каждую исключительную ситуацию, указанную в спецификациях, - хотя бы один тест.

Во втором случае стратегия базируется на принципе: каждая команда программы ПС должна проработать хотя бы на одном тесте.

# **10.1 Виды тестирования**

Можно выделить следующие виды тестирования.

**Модульное тестирование (автономная отладка)** - тестируется каждый отдельный модуль, в отрыве от остальной системы. Самый распространенный случай применения – тестирование модуля самим разработчиком, проверка того, что отдельные модули делают действительно то, что от них ожидается.

Различные среды разработки поддерживают средства модульного тестирования, например, свободно распространяемая библиотека для Visual Studio NUnit.

Созданные разработчиком модульные тесты часто включаются в пакет регрессионных тестов, и могут запускаться многократно.

**Интеграционное тестирование** – компоненты тестируются на совместимость, поскольку разные компоненты могут создаваться разными людьми, в разное время, по разным технологиям. Этот вид тестирования должен применяться самими программистами, чтобы удостовериться, что все живет вместе в первом приближении. Далее тонкости интеграции могут исследовать тестеровщики. «Ошибки на стыках» непросто обнаруживать и устранять. Во время разработки все компоненты вместе не готовы, интеграция откладывается, а в конце обнаруживаются трудные ошибки (их устранение требует существенной работы). Здесь выходом является ранняя интеграция системы и в дальнейшем использование практики постоянной интеграции.

**Системное тестирование** – это тестирование всей системы в целом, как правило, через ее пользовательский интерфейс. При этом внимание акцентируются на том, как ПС выглядит и работает в целом, удобно ли оно, удовлетворяет ли ожиданиям заказчика.

**Регрессионное тестирование** – тестирование системы в процессе ее разработки и сопровождение на не регресс. То есть проверяется, что изменения системы не ухудшили уже существующей функциональности. Для этого создаются пакеты регрессионных тестов, которые запускаются с определенной периодичностью, например, в пакетном режиме, связанные с процедурой постоянной интеграции.

**Нагрузочное тестирование** – тестирование системы на корректную работу с большими объемами данных. Например, проверка баз данных на корректную обработку предельного объема записей, исследование поведения серверного ПО при большом количестве клиентских соединений, эксперименты с предельным трафиком для сетевых и телекоммуникационных систем, одновременное открытие большого числа файлов, проектов и т.д.

**Стрессовое тестирование** – тестирование системы на устойчивость к непредвиденным ситуациям. Этот вид тестирования нужен далеко не для каждой системы, так как подразумевает высокую планку качества.

**Приемочное тестирование** – тестирование, выполняемое при приемке системы заказчиков. Различные стандарты часто включают в себя наборы приемочных тестов.

# **10.2 Автономная отладка программного средства**

Автономная отладка ПС означает последовательное, раздельное тестирование различных программ, входящих в ПС, с поиском и исправлением в них ошибок. Она включает отладку каждого программного модуля и отладку сопряжения модулей.

При автономной отладке ПС каждый модуль тестируется в некотором программном окружении:

* модулей отлаживаемой программы, которые уже отлажены;
* модулей, управляющих отладкой - отладочные модули.

При автономной отладке тестируется программа, построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой.

В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями. Такой процесс называется интеграцией программы. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы .

При восходящем тестировании окружение отлаживаемого модуля содержит только один отладочный модуль - ведущий в тестируемой программе.

Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля, осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

Достоинствами восходящего тестирования являются:

* простота подготовки тестов;
* возможность полной реализации схемы тестирования модуля, которая связана с тем, что тестовое состояние информационной среды готовится непосредственно перед обращением к отлаживаемому модулю (ведущим отладочным модулем).

К недостаткам восходящего тестирования относится то, что:

* тестовые данные готовятся не в той форме, которая рассчитана на пользователя;
* необходимо выполнять большой объем отладочного программирования;
* необходимо выполнять специальное тестирование сопряжения модулей.

Тестирование сопряжения модулей при восходящем тестировании осуществляется путем обращения к отлаживаемому модулю из ведущего отладочного модуля.

При нисходящем тестировании окружение отлаживаемого модуля в качестве отладочных модулей содержит отладочные имитаторы. К таким модулям относятся модули, к которым может обращаться отлаживаемый модуль. Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

Достоинствами нисходящего тестирования являются:

* возможность готовить тесты в форме, рассчитанной на пользователя;
* небольшой объем отладочного программирования (имитаторы модулей просты);
* отпадает необходимость тестирования сопряжения модулей.

Недостатком данного подхода является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно, так как оно является результатом применения уже отлаженных модулей к тестовым данным. Некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать.

Для осуществления тестирования отлаживаемого модуля в указанных ситуациях используют имитаторы, чтобы создать требуемое состояние информационной среды.

Необходимо организовать отладку программы таким образом, чтобы раньше были отлажены модули, осуществляющие ввод данных, тогда тестовые данные можно готовить в форме, рассчитанной на пользователя. Пока модули, осуществляющие ввод данных, не отлажены, тестовые данные поставляются некоторыми имитаторами.

Тестирование сопряжения модулей при нисходящем тестировании делается попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования.

Нисходящее тестирование является предпочтительным.

Комбинация восходящего и нисходящего тестирования заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы. Этот метод позволяет воспользоваться достоинствами обоих методов и нейтрализовать их недостатки.

# **10.3 Комплексная отладка ПС**

Комплексная отладка означает тестирование ПС в целом с поиском и исправлением ошибок во всех документах, относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые могут возникнуть у пользователя в реальных условиях. При комплексной отладке тестируется ПС в целом. Тесты готовятся по каждому из документов. Тестирование этих документов производится в порядке, обратном их разработке .

Тестирование архитектуры ПС. Целью этого тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны со взаимодействием подсистем, в частности, с реализацией архитектурных функций. Проверяются все пути взаимодействия между подсистемами ПС.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Указанные расхождения могут быть из-за того, что внутренние спецификации программ не соответствуют функциональной спецификации ПС.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Завершённость ПС проверяется уже при тестировании внешних функций. Тестируются примитивы качества: точность, устойчивость, защищённость, временная эффективность, эффективность по памяти, эффективность по устройствам, расширяемость и независимость от устройств. Лёгкость применения ПС оценивается при тестировании документации по применению ПС.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности между документацией по применению ПС и совокупностью программ ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки. Все тесты на этом этапе готовятся на основании документации по применению ПС. Должны быть протестированы все примеры, использованные в документации.

Тестирование определения требований ПС (валидация). Целью тестирования является выяснение, в какой мере ПС соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет пользователь ПС. Обычно это тестирование производится с помощью контрольных задач, для которых известен результат решения.

Когда разрабатываемое ПС должно придти на смену другой версии ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС с последующим сопоставлением результатов. Иногда в качестве формы такого тестирования используют опытную эксплуатацию ПС - ограниченное применение нового ПС с анализом использования.

Всегда должен быть четко оговорен конечный результат тестирования. Еще до начала работ или на одной из ранних стадий должен быть создан и согласован с заказчиком набор тестов, успешный пропуск которых свидетельствует об успешном окончании работы в целом. Для регламентации этих работ можно применять ГОСТ 19.301 «Программа и методика испытаний.

# **10.4 Работа с ошибками**

Между программистами и тестеровщиками необходим контакт, так как ошибок находится много, их исправление требует времени, а в их исправлении тестеровщики должны удостовериться. Кроме того, менеджерам нужна статистика по найденным и исправленным ошибкам для контроля проекта.

Чтобы справиться с этим потоком информации и обеспечить необходимые в работе, удобные сервисы, существует специальный класс программных средств - средств контроля ошибок.

Как правило, описание ошибки в системе контроля ошибок имеет следующие основные атрибуты:

* ответственного за ее проверку – тестеровщика, который ее нашел и проверяет, что исправления, сделанные разработчиком, действительно устраняют ошибку;
* ответственного за ее исправление – разработчика, которому ошибка отправляется на исправление;
* состояние, например: ошибка найдена, ошибка исправлена, ошибка закрыта, ошибка вновь проявилась и т.д.

Использование этих систем стало общей практикой в разработке ПС, наравне со средствами версионного контроля и иными инструментами. Они включают в себя:

* базу данных для хранения ошибок;
* интерфейс к этой базе данных для внесения новых ошибок и задания их многочисленных атрибутов, для просмотра ошибок на основе различных фильтров, например: все найденные ошибки за последний месяц, все ошибки, за которые отвечает данный разработчик и т.д.;
* сетевой доступ, так как проекты все чаще оказываются распределенными;
* программный интерфейс для возможностей программной интеграции таких систем с другим ПО, поддерживающим разработку ПО.

Очень важным при работе с ошибками оказываются различные отчеты.

**Заповеди отладки ПС**

1. Считать тестирование ключевой задачей разработки ПС, поручать

его самым квалифицированным и одарённым программистам.

2. Нежелательно тестировать свою собственную программу.

3. Хорош тот тест, для которого высока вероятность обнаружения ошибки, а не тот, который демонстрирует правильную работу программы.

4. Готовить тесты для правильных, и для неправильных данных.

5. Документировать пропуск тестов через компьютер, детально изучать результаты каждого теста, избегать тестов, пропуск которых нельзя повторить.

6. Каждый модуль подключать к программе только один раз, никогда не изменяя программу, чтобы облегчить ее тестирование.

7. Пропускать заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в неё были внесены изменения.

# **Лекция 11. Тестовое покрытие.**

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

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

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

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

Существуют следующие подходы к оценке и измерению тестового покрытия:

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

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

**Тестовое покрытие на базе анализа потока управления** - оценка покрытия основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

**Различия:**

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

**Ограничения:**

Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом.

Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

# **11.1 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. Пример разработки тестовых случаев по имеющимся требованиям рассмотрен в разделе: "Практическое применение техник тест дизайна при разработке тест кейсов"

# **11.2 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.

# **11.3 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:

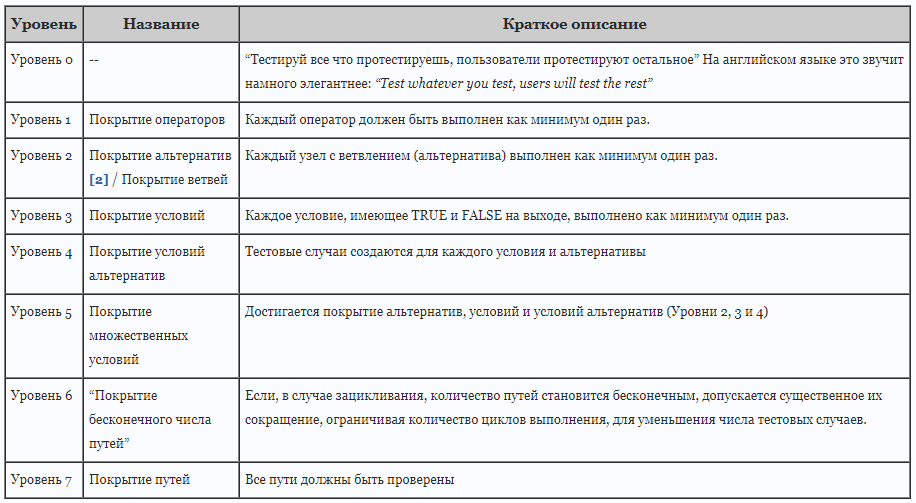


Рис.11.1 – Уровни тестового покрытия

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

# **Лекция 12. Тестовый сценарий, тестовый пакет.**

На этапе контроля качества реализованной функциональности используется тестовая документация, в которой записаны стандартные и альтернативные сценарии работы с приложением, используемые при тестировании очередной версии приложения.

Тестовая документация состоит из тестовых сценариев, то есть специальным образом разработанного описания последовательности действий в системе и ожидаемого поведения. Тестовые сценарии используются для проведения различных видов ручного тестирования:

* функционального тестирования;
* приемочного тестирования;
* нагрузочного или стресс-тестирования;
* исследовательского тестирования;
* smoke-тестирования и др.

Для разработки тестовых сценариев и выполнения тестов используются системы управления тестированием, существенно повышающие производительность тест-дизайнеров и тестировщиков, а также обеспечивающие видимость уровня качества приложений среди всех участников проекта.

Тестовые сценарии неразрывно связаны с требованиями, изменения в которых должны своевременно отражаться в тестовой документации, что позволяет сделать система управления жизненным циклом разработки приложений, при помощи механизма трассировок.

При выполнении теста тестировщик отмечает результат прохождения одного шага или всего тестового сценария, прикрепляет обнаруженные ошибки и другую вспомогательную информацию: скриншоты, дампы, логи и т.п.

Тестовые сценарии удобно объединять в тест-планы по назначению:

* тестирование релиза, то есть очередной версии продукта;
* тестирование развертывания;
* тестирование удобства использования;
* конфигурационное тестирование;
* тестирование безопасности и т.п.

Зачастую ручное тестирование превращается в рутину и занимает значительное время, что отрицательно сказывается на скорости выпуска релизов позволяет:

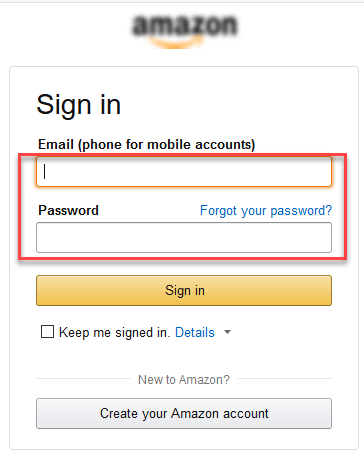
* высвободить ресурсы для проведения более сложных видов тестирования;
* снизить количество дефектов, доходящих до стадии контроля качества;
* ускорить выпуск релизов.

Сведение результатов автоматических и ручных тестов в системе управления качеством, позволяет всем участникам проекта видеть уровень качества очередного релиза, контролировать его изменение и опираться на эту информацию при планировании своей работы.

Пример 1. Сценарий тестирования для приложения электронной коммерции

Для приложения электронной коммерции несколько тестовых сценариев

**Тестовый сценарий 1**: проверка функциональности входа



Чтобы помочь вам понять разницу между сценарием тестирования и тестовыми сценариями, для этого сценария тестирования будут использоваться конкретные тестовые сценарии.

Проверьте поведение системы при вводе действительного адреса электронной почты и пароля.

Проверьте поведение системы при вводе неверного идентификатора электронной почты и действительного пароля.

Проверьте поведение системы при вводе действительного адреса электронной почты и неверного пароля.

Проверьте поведение системы при вводе неверного идентификатора электронной почты и неверного пароля.

Проверьте поведение системы, если адрес электронной почты и пароль оставлены пустыми и введен вход.

Проверить Забыли пароль работает как положено

Проверьте поведение системы при вводе действительного / недействительного номера телефона и пароля.

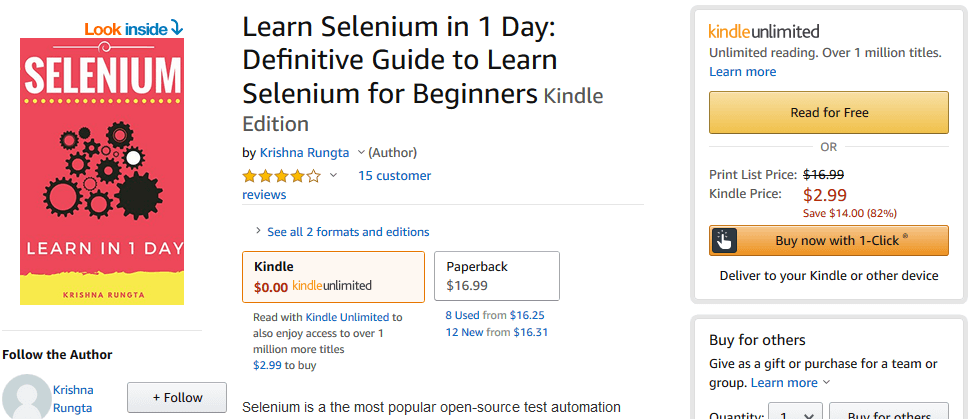
Проверять поведение системы, когда установлен флажок «Держать меня в подписи»

Как видно, тестовые случаи являются более конкретными.

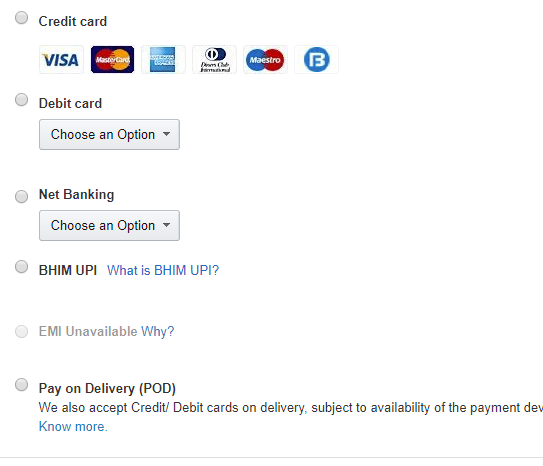
**Тестовый сценарий 2.** Проверка функциональности поиска

тестовый сценарий

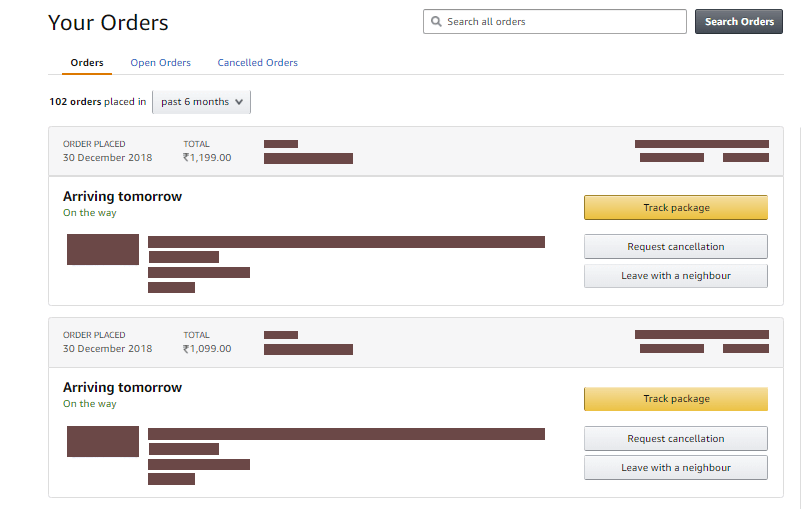
**Тестовый сценарий 3**: проверьте страницу описания продукта



**Сценарий тестирования 4**: Проверьте функциональность платежей



**Тестовый сценарий 5:** проверка истории заказов



Помимо этих 5 сценариев, здесь приведен список всех других сценариев.

* Проверьте поведение домашней страницы для постоянных клиентов
* Проверьте категорию / страницы продукта
* Проверьте службу поддержки / контактные страницы
* Проверьте страницы ежедневных предложений

# **Лекция 13. Анализ спецификаций. Верификация и аттестация программного обеспечения.**

# 