Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина»

УТВЕРЖДАЮ

Директор по образовательной деятельности

_____ С.Т. Князев

«_7_» _сентября_ 2023 г.

Системная аналитика

Учебно-методические материалы по направлению подготовки
09.03.03 Прикладная информатика
Образовательная программа «Прикладной искусственный интеллект»

Екатеринбург

РАЗРАБОТЧИКИ УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Доцент Базовой кафедры
«Аналитика больших данных и
методы видеоанализа» _____ М.Ю. Новиков

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Дисциплина «Системный аналитика» рассматривает теоретические и прикладные инструменты с целью реализации сервисов для области IT.

Целью дисциплины является формирование у студентов теоретических знаний в области применения инструментов системного анализа для определения наиболее эффективных форм развития популярных сервисов и возможностей для реализации и развития сферы IT.

Лекция 1. Введение в системную аналитику. Зоны ответственности системного аналитика

Основная цель системного аналитика - разработка, восстановление и сопровождение требований к ПО, продукту, средству, программно-аппаратному комплексу, автоматизированной информационной системе или автоматизированной системе управления (далее - системе) на протяжении их жизненного цикла.

Для достижения данной цели системный аналитик должен обладать компетенциями по следующим направлениям:

1. Предметная область:

- Знание юридических законов и стандартов;
- Знание нормативных документов;
- Знание продуктов;
- Знание трендов;
- Знание основных компетенций.

2. Модели и методологии ЖЦ ПО:

- Waterfall;
- Спиральная;
- Итеративная;
- Инкрементальная.

3. Моделирование и анализ процессов:

- BPMN;
- UML:
  - Диаграммы поведения;
  - Структурные диаграммы;
- ER.

4. Архитектура приложений.

Паттерны серверной архитектуры:

- Монолит;

- SOA;
- Микросервисная архитектура.

Паттерны клиент-серверной архитектуры:

- Клиент-серверная архитектура;
- Приложение;
- n-звенная.

5. Интеграция:

- Файловая;
- Через БД;
- Интеграции;
- Очередь;
- Шина.

6. СУБД:

- Основы проектирования;
- Знание SQL;
- Типы БД:
  - Реляционные;
  - Нереляционные.

7. Документирование;

8. Работа с требованиями;

9. Работа с Git;

10. Тестирование.

Лекция 2. Жизненный цикл ПО. Методологии разработки

Понятие жизненного цикла является одним из базовых понятий методологии проектирования информационных систем. Жизненный цикл информационной системы представляет собой непрерывный процесс, начинающийся с момента принятия решения о создании информационной системы и заканчивающийся в момент полного изъятия ее из эксплуатации.

Жизненный цикл информационных систем регламентирует международный стандарт ISO/IEC 12207-2010. Процесс жизни любой системы или программного продукта может быть описан посредством модели жизненного цикла, состоящей из стадий. Модель жизненного цикла представляется в виде последовательности стадий, которые могут перекрываться и (или) повторятся циклически в соответствии с областью применения, размером, сложностью, потребностью в изменениях и возможностями. Каждая стадия описывается формулировкой цели и выходов. Процессы и действия жизненного цикла отбираются и исполняются на этих стадиях для полного удовлетворения цели и результатов этой стадии.

Модель жизненного цикла ИС — структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Модель ЖЦ ИС включает в себя:

- стадии;
- результаты выполнения работ на каждой стадии;
- ключевые события — точки завершения работ и принятия решений.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления.

Типы моделей жизненного цикла ИС

В настоящее время наиболее известны и используются следующие модели жизненного цикла:

Каскадная модель (рис. 1) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.

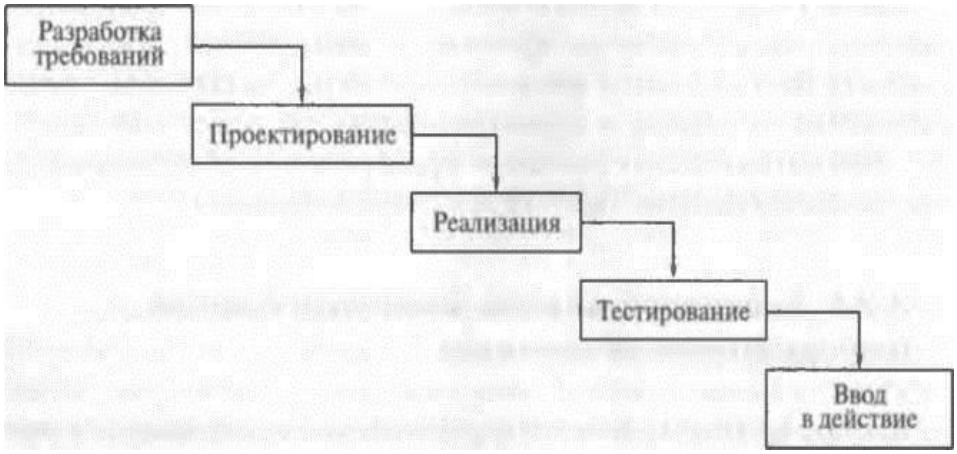


Рис. 1. Каскадная модель ЖЦ ИС

На первом этапе проводится исследование проблемы, которая должна быть решена, четко формулируются все требования заказчика. Результатом, получаемым на данном этапе, является техническое задание (разработка требований), согласованное со всеми заинтересованными сторонами.

На втором этапе разрабатывают проектные решения, удовлетворяющие всем требованиями, сформулированным в техническом задании. Результатом данного этапа является комплект проектной документации, содержащей все необходимые данные для реализации проекта.

Третий этап — реализация проекта. Здесь осуществляется разработка программного обеспечения (кодирование) в соответствии с проектными решениями, полученными на предыдущем этапе. Методы, используемые для реализации, не имеют принципиального значения. Результатом выполнения данного этапа является готовый программный продукт.

На четвертом этапе проводится проверка (тестирования) полученного программного обеспечения на предмет соответствия требованиям, заявленным в техническом задании. Опытная эксплуатация позволяет выявить и исправить

различного рода скрытые недостатки, проявляющиеся в реальных условиях работы информационной системы. Результат – готовая к эксплуатации система.

Последний этап — сдача готового проекта, ввод его в действие. Главная задача этого этапа — документально подтвердить, что все требования заказчика выполнены в полной мере. Результат – работающая ИС с полным комплектом сопроводительной документации, утвержденные заказчиком документы о принятии системы в эксплуатацию.

Можно выделить следующие положительные стороны применения каскадного подхода:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе.

Недостатки каскадной модели. Перечень недостатков каскадной модели при ее использовании для разработки информационных систем достаточно обширен:

- существенная задержка в получении результатов;
- ошибки и недоработки на любом из этапов проявляются, как правило, на последующих этапах работ, что приводит к необходимости возврата назад;
- сложность параллельного ведения работ по проекту;
- чрезмерная информационная перенасыщенность каждого из этапов;
- сложность управления проектом;
- высокий уровень риска и ненадежность инвестиций.

Основным недостатком этого подхода является то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении

или пересмотре ранее принятых решений. Это приводит к существенной задержке в получении результатов

В результате реальный процесс создания ИС оказывается соответствующим поэтапной модели с промежуточным контролем. Здесь разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки (рис. 2).

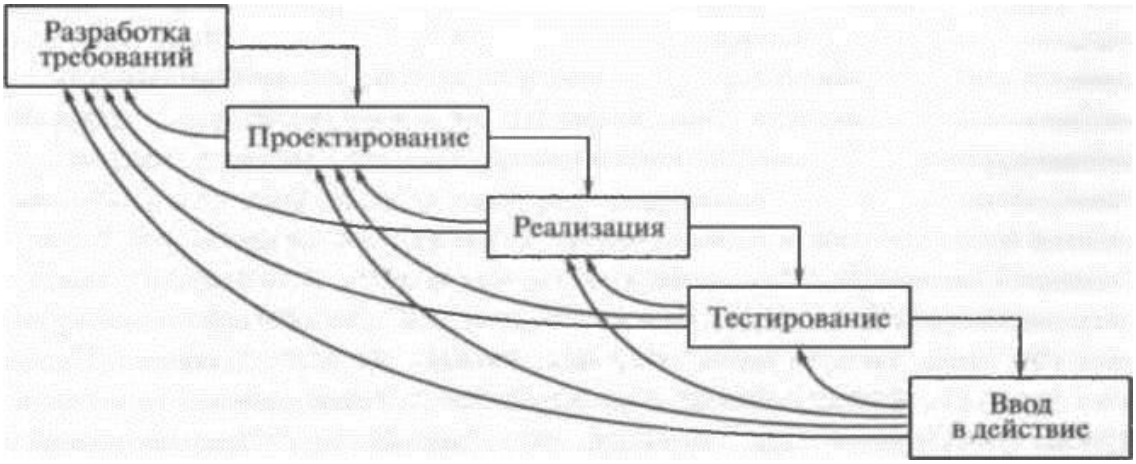


Рис. 2. Поэтапная модель с промежуточным контролем

Спиральная модель ЖЦ (рис. 3) была предложена для преодоления перечисленных проблем каскадной модели. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов.

Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный

вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

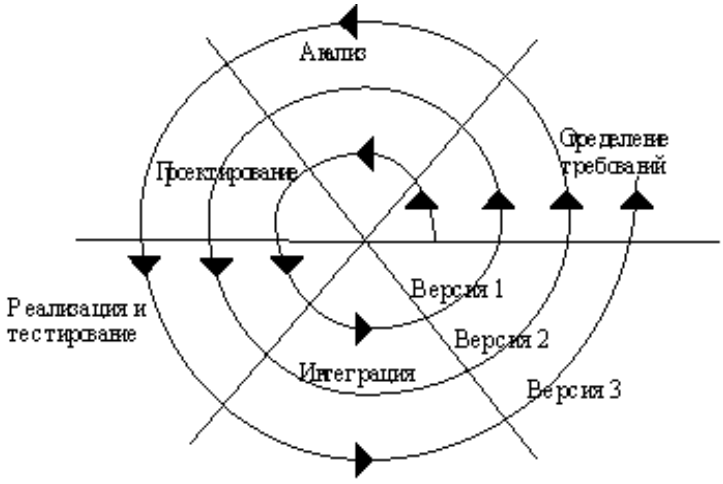


Рис. 3. Спиральная модель ЖЦ ИС

Использование спиральной модели позволяет перейти на следующий этап выполнения проекта, не дожидаясь полного завершения текущего — недоделанную работу можно будет выполнить на следующей итерации. Главная задача каждой итерации — как можно быстрее создать работоспособный продукт, который можно показать пользователям системы. Таким образом, существенно упрощается процесс внесения уточнений и дополнений в проект.

Рассмотрим преимущества итерационного подхода, применяемого в спиральной модели, более подробно.

1. Существенно упрощается внесение изменений в проект при изменении требований заказчика.

2. При использовании спиральной модели отдельные элементы информационной системы интегрируются в единое целое постепенно.

Поскольку интеграция начинается с меньшего количества элементов, то возникает гораздо меньше проблем при ее проведении (по некоторым оценкам, при использовании каскадной модели разработки интеграция занимает до 40 % всех затрат в конце проекта).

3. Уменьшение уровня рисков. Данное преимущество является следствием предыдущего, так как риски обнаруживаются именно во время интеграции. Поэтому уровень рисков максимален в начале разработки проекта.

По мере продвижения разработки ожидаемый уровень рисков снижается. Данное утверждение справедливо при любой модели разработки, однако при использовании спиральной модели снижение уровня рисков происходит с наибольшей скоростью. Это связано с тем, что при данном подходе интеграция выполняется уже на первой итерации, и на начальных витках спирали выявляются многие аспекты проекта, такие как пригодность используемых инструментальных средств и программного обеспечения, квалификация разработчиков и т. п.

На рис. 4 приведены в сравнении графики зависимости уровня рисков от времени разработки для каскадного и итерационного подходов.

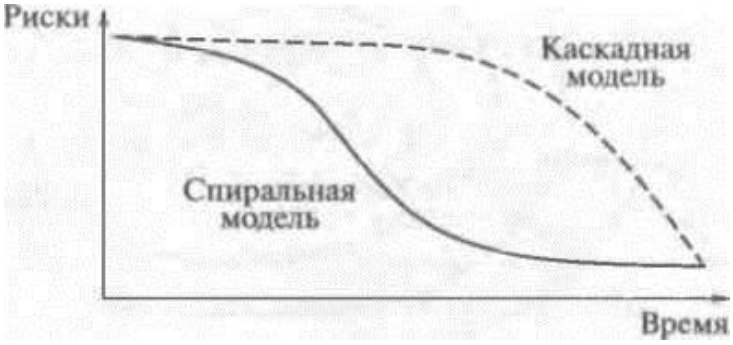


Рис. 4. Зависимость рисков от времени разработки

4. Итерационная разработка обеспечивает большую гибкость в управлении проектом, давая возможность внесения тактических изменений в разрабатываемое изделие. Например, можно сократить сроки разработки за счет снижения функциональности системы или использовать в качестве составных частей системы продукцию сторонних фирм вместо собственных разработок.

5. Итерационный подход спиральной модели ЖЦ упрощает повторное использование компонентов (реализует компонентный подход к программированию). Это обусловлено тем, что гораздо проще выявить (идентифицировать) общие части проекта, когда они уже частично разработаны, чем пытаться выделить их в самом начале проекта. Анализ проекта после проведения нескольких начальных итераций позволяет выявить общие многократно используемые компоненты, которые на последующих итерациях будут совершенствоваться.

6. Итерационный подход дает возможность совершенствовать процесс разработки — анализ, проводимый в конце каждой итерации, позволяет проводить оценку того, что должно быть изменено в организации разработки, и улучшить ее на следующей итерации.

Недостатки спиральной модели. Основная проблема спирального цикла — определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Иначе процесс разработки может превратиться в бесконечное совершенствование уже сделанного. При итерационном подходе полезно следовать принципу «лучшее — враг хорошего». Поэтому завершение итерации необходимо проводить строго в соответствии с планом, даже если не вся запланированная работа закончена. Планирование работ обычно проводится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Инкрементная модель жизненного цикла (рис. 5) представляет собой пример итеративного подхода к разработке программного обеспечения ИС, который предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включающий все фазы жизненного цикла в применении к созданию отдельных версий системы, обладающих меньшей функциональностью по сравнению с проектом, в целом.

При этом на каждой итерации получается работающая версия программной системы, обладающая функциональностью всех предыдущих плюс текущей итерации. В результате финальной итерации получается конечный продукт, обеспечивающий реализацию всех требований.

С точки зрения структуры жизненного цикла модель называют итеративной (говоря о процессе). С точки зрения развития продукта – инкрементной (имеется в виду наращивание функциональности продукта). Инкремент – приращение, увеличение (например, в языке программирование – увеличение значения переменной на 1).

Рис. 5. Инкрементная модель жизненного цикла

Для каждого инкремента выполняется:

- Анализ, на котором мы собираем требования и анализируем, и планируем сам инкремент;
- Проектирование, на котором происходит проектирование архитектуры, допроектирование тех вещей, которые не были сделаны на предыдущем инкременте;
- Разработка;
- Тестирование.

Важно, что каждый инкремент заканчивается работающим продуктом. Пусть он ограниченной функциональности, пусть у него не все реализовано, но это отдельный продукт, который можно показать, который можно отдать заказчику на тестирование.

Сам подход является однократным, т.е. мы весь наш проект делаем за один большой этап, имея в виду требования, но весь проект делится на инкременты, это небольшие версии продуктов, для которых заранее определена функциональность. Т.е. мы говорим, что у нас вся длительная разработка состоит, например, из 5 фаз. На первом этапе получим однопользовательскую версию, на втором инкременте (фазе) мы получим версию, которая поддерживает много пользователей, на третьем инкременте у нас появится поддержка веб-интерфейсов, а на четвертом какое-нибудь протоколирование и

т.д. (рис. 6). Т.е. хоть требования у нас все равно задаются только один раз, но на выходе у нас появляется несколько инкрементов.

Рис. 6. Иллюстрация различий между инкрементной и итеративной (спиральной) моделями

Преимущества инкрементной модели:

- на момент создания определенного инкремента требования стабилизируются, поскольку не являющиеся особо важными изменения отодвигаются на момент создания последующих инкрементов;
- не требуется заранее тратить средства, необходимые для разработки всего проекта, поскольку сначала выполняется разработка и реализация основной функции или функции из группы высокого риска;
- в результате выполнения каждого инкремента получается функциональный продукт;
- ускоряется начальный график поставки (что позволяет соответствовать возросшим требованиям рынка);
- риск распределяется на несколько меньших по размеру инкрементов (не сосредоточен в одном большом проекте разработки);
- в конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика.

Недостатки инкрементной модели:

- в модели не предусмотрены итерации в рамках каждого инкремента;

• определение полной функциональности системы должно осуществляться в начале жизненного цикла, чтобы обеспечить определение инкрементов;

• формальный критический анализ и проверку намного труднее выполнить для инкрементов, чем для системы в целом;

• поскольку создание некоторых модулей будет завершено значительно раньше других, возникает необходимость в четко определенных интерфейсах;

• для модели необходимо хорошее планирование и проектирование;

• может возникнуть тенденция к оттягиванию решений трудных проблем на будущее с целью продемонстрировать руководству успех, достигнутый на ранних этапах разработки.

Именно на инкрементальных подходах к разработке ПО основаны методы гибкой разработки.

Термин «гибкая разработка» приобрел популярность с момента публикации Манифеста гибкой разработки ПО в 2001 году.

Вот его 12 принципов:

1. Наивысшим приоритетом является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.

2. Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.

3. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.

4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

5. Над проектом должны работать мотивированные профессионалы.

6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.

7. Работающий продукт — основной показатель прогресса.

8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.

9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.

10. Простота — искусство минимизации лишней работы — крайне необходима.

11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.

12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Говоря о гибкой разработке, подразумевают набор методов разработки ПО, которые обеспечивают: первое: постоянное сотрудничество между заинтересованными лицами, и второе: быстрый и частый выпуск функциональности небольшими порциями. Существует много разных типов гибкой разработки, среди которых Scrum, Kanban, Экстремальное программирование, Бережливое производство. Методики гибкой разработки обладают разными характеристиками, но в основном в них предпочтение отдается адаптивному, а не прогнозному подходу. В прогнозном подходе, таком как водопадная методика, пытаются минимизировать риски в проекте, выполняя значительный объем планирования и документирования до начала создания ПО. Менеджеры проектов и бизнес-аналитики обеспечивают, чтобы все заинтересованные лица четко понимали, что планируется выпустить, до начала разработки. Это работает, если с самого начала есть четкое понимание требований, а сами требования остаются практически неизменными на протяжении всего проекта. Адаптивные подходы, такие как методы гибкой разработки, призваны обеспечить приспособляемость к неизбежным изменениям, которые будут происходить в проектах. Они также хорошо работают в проектах с неточными или изменчивыми требованиями.

Лекция 3. Работа с требованиями. Заинтересованные стороны

Требования — это высказывание, которое выражает потребность и связанные с ней ограничения и условия. Требования делятся на уровни и типы.

Выделяют 3 уровня требований:

Первый. Бизнес-требования - описывают, почему организации нужна такая система, или иными словами, это цели, которые организация намерена достичь с ее помощью. Основное их содержание — это бизнес-цели организации или клиента, заказывающих систему.

Например: компания хочет снизить затраты на документооборот на 50%. В качестве идеи может возникнуть желание использовать систему электронного документооборота.

Второй уровень. Пользовательские требования - содержат описание целей и задач, которые пользователи должны иметь возможность выполнять с помощью продукта или системы. Они также включают описание атрибутов или характеристик продукта, которые важны для удовлетворения пользователей.

Как правило, пользовательские требования оформляются в виде вариантов использования и пользовательских историй. Т.к. пользовательские требования описывают, что пользователь должен иметь возможность делать с системой, то лучшим источником информации являются реальные представители пользователей.

Примером сценария использования является прикрепление документа в системе эл документооборота для последующей его отправки на согласование. Если сформулировать это же требование в виде пользовательской истории, то оно может звучать следующим образом: «Я как пользователь хочу прикрепить выбранный мною документ, чтобы отправить его на согласование». Важно помнить, что в большинстве проектов есть несколько классов пользователей, а также других участников, потребности которых тоже надо выявить.

И, третий уровень — это функциональные требования. Именно они определяют, каким должно быть поведение продукта в тех или иных условиях,

т.е, что разработчики должны создать, чтобы пользователи смогли выполнить свои задачи в рамках бизнес-требований.

Функциональные требования описываются в форме утверждений со словами «должен» или «должна», например: «При отправке документа на согласование у пользователя должна быть возможность выбрать согласующих лиц».

Вся информация собирается в спецификации ПО.

Давайте посмотрим на визуализацию.

Овал обозначает вид требований.

Прямоугольник - документ, в котором эти требования хранятся.

Сплошная линия указывает на документ, в котором эта информация хранится.

Пунктирная линия означает, что блок, из которого она выходит, является источником информации и оказывает влияние на информацию другого типа или требования, на которое она направлена.

На все типы требований оказывает влияние внешняя информация, и следствием является особая реализация, которая учитывает это влияние.

Внешнее воздействие оказывают: системные требования, внешние интерфейсы, ограничения, бизнес-правила и атрибуты качества.

Говоря о работе над требованиями, мы выделяем 2 направления:

1. Разработка требований, которая включает в себя (Выявление и сбор, Анализ, Документирование, Утверждение);
2. Управление требованиями.

Начнем с рассмотрения этапов разработки требований.

Первое. Выявление и сбор.

В первую очередь нам нужно понять у кого мы можем получить требования, поэтому важно определить заинтересованные стороны и понять, что требования относятся именно к нашей системе.

Определив источник информации (т.е. того, у кого мы будем получать требования), мы переходим к этапу выявления требований при помощи различных методов.

В проектах по разработке ПО могут применяться разные методы выявления требований. На самом деле, вряд ли найдется проектная команда, в которой используется только один метод. Всегда есть несколько типов информации, которую надо выявлять, и разные заинтересованные лица предпочитают разные подходы. Один пользователь может четко сформулировать, как он использует систему, а за другим придется понаблюдать в работе, чтобы получить такое понимание сценария использования.

Методы выявления требований делятся на коллективные, в которых участвуют заинтересованные лица, и независимые, когда вы работаете сами над выявлением информации. Коллективные методы ориентируются на выявление пользовательских и бизнес-требований. Непосредственная работа с пользователями необходима, потому что пользовательские требования связаны с задачами, которые пользователи должны выполнять в системе. Независимые методы дополняют требования, предоставляемые пользователями, и позволяют выявить функциональность, о которой конечные пользователи могут не знать. В большинстве проектов используется сочетание коллективных и независимых методов выявления требований. Разные методы предоставляют возможность по-разному исследовать требования и даже могут выявлять разные требования.

Этап анализа требований подразумевает их исследование требований на предмет ошибок, пробелов и других недостатков, а также включает разбиение высокоуровневых требований на более детальные. Цель анализа требований — качественное и подробное описание требований, позволяющие менеджерам реалистично оценить все затраты на проект, а команде разработки — начать работу над проектом.

Следующий этап работы над требованиями – это документирование.

Данный этап предусматривает представление и хранение информации о проекте.

Целью документирования является преобразование собранных потребностей пользователей в письменные требования и диаграммы, пригодные для понимания, анализа и использования целевой аудиторией.

И заключительный этап работы с требованиями: их утверждение.

Данный этап позволяет выяснить, написали ли вы правильные требования, то есть что они соответствуют бизнес-целям.

Управление требованиями необходимо чтобы убедиться, что усилия, затраченные на разработку требований, не потрачены впустую. Эффективное управление требованиями может уменьшить неверные ожидания: заинтересованные лица будут проинформированы о текущем состоянии требований в ходе процесса разработки. Это позволяет понимать, куда движется проект, как идут дела и когда проект завершится.

Под управлением требованиями подразумевают все действия, по обеспечению целостности, точности и своевременности обновления соглашения о требованиях в ходе проекта.

Заинтересованные стороны.

Теория заинтересованных сторон и управления отношениями с ними появилась относительно недавно (понятие «стейкхолдер» ввел Фримен в 1984 г.), однако уже доказала свою жизнеспособность и практическую полезность. Согласно этой теории, отношения со всеми индивидами и группами лиц, непосредственно участвующими в проекте или теми, интересы которых могут быть затронуты в процессе реализации проекта или после его завершения, должны быть объектами особого внимания. При этом главная задача руководителя проекта — определить ключевых стейкхолдеров проекта и организовать взаимодействие с ними таким образом, чтобы снизить возможность и последствия их негативного влияния и усилить позитивное воздействие. Другими словами, чтобы проект был успешным, интересы стейкхолдеров обязательно должны учитываться менеджером проектов. Более того, Фримен определяет достижение баланса интересов стейкхолдеров в качестве ключевой цели для любой организации.

В управлении проектами, которое характеризуется высокой динамичностью окружающей среды, концепция стейкхолдеров приобретает критическое значение. Эта особенность современных проектов наиболее точно отражена в определении проекта, данном в стандарте P2M: «Проект — это обязательство создать ценность, которое должно быть выполнено в рамках согласованного времени, ресурсов и условий эксплуатации». Использование термина «ценность» вместо «результат» носит принципиальный характер, поскольку один и тот же результат представляет разную ценность для различных заинтересованных сторон. Более того, даже для одного стейкхолдера субъективно воспринимаемая ценность результата может меняться в ходе проекта, особенно если речь идет о длительных проектах или о проектах с невысокой степенью формализации результатов при входе.

Понятие и методы работы с заинтересованными сторонами проекта давно находятся в фокусе внимания многих исследователей и отражены в стандартах управления проектами.

Принято считать, что теория заинтересованных сторон появилась в 1984 г., когда Фримен определил стейкхолдеров как «любую группу или индивида, который может сам оказывать влияние или на которого окажет влияние достижение целей организации».

ПРОЦЕССЫ УПРАВЛЕНИЯ ЗАИНТЕРЕСОВАННЫМИ СТОРОНАМИ ПРОЕКТА

Пожалуй, наиболее полно раскрытым в специализированных источниках по управлению проектами аспектом управления заинтересованными сторонами являются процессы. Процессы управления заинтересованными сторонами нашли отражение во всех признанных международных и национальных стандартах, в частности в ISO 21500, ICB IPMA, PMBOK PMI, P2M , НТК СОВНЕТ , P-4R. Рассмотрим модель P-4R.

Управление заинтересованными сторонами в модели P-4R

Пожалуй, в наиболее развернутом виде описание процессов управления заинтересованными сторонами можно найти в модели P-4R. Проекты, как правило, имеют значительное количество различных заинтересованных сторон, зависящее от типа проекта, его масштаба и сложности. Соответственно, руководителю проекта для успешного выполнения своих задач необходимо как можно раньше определить и классифицировать все заинтересованные стороны, выявить уровень их интереса и влияния на проект, а также проанализировать их индивидуальные ожидания. Эти первоначально полученные данные должны регулярно пересматриваться и обновляться, поскольку на различных стадиях жизненного цикла проекта количество заинтересованных сторон, их требования и влияние, а также отношение к проекту и друг к другу могут меняться. Р. Динг предложил модель P-4R, в которой управление данными сторонами рассматривается как унифицированный повторяющийся процесс. Данная модель состоит из четырех ключевых шагов (требования, роли, риски и взаимоотношения).

Первый шаг включает выявление требований заинтересованных сторон проекта. Для этого в первую очередь необходимо определить состав этих сторон и стратегии работы с ними. Далее следует выявить их ожидания, степень их важности и перевести ожидания в формализованные требования с четкими критериями выполнения. Этот момент является принципиально важным для успешной реализации проекта, поскольку «участники объединяются, ожидая, что каждый из них получит конкретные выгоды». Наконец, еще одной стороной взаимоотношений являются конфликты, и на первом этапе реализации проекта необходимо найти способы разрешения потенциальных проблемных ситуаций с целью достижения консенсуса и гармонизации требований заинтересованных сторон.

Второй шаг состоит в разделении обязанностей и распределении руководящих ролей среди заинтересованных сторон. Руководящие роли в проекте подразумевают сочетание прав и обязанностей, при этом может потребоваться выполнение различных функций одним или несколькими

участниками проекта для удовлетворения требования другого участника. Исполнители руководящих ролей должны быть наделены адекватными правами в своей организации. В процессе осуществления проекта между исполнителями могут возникать конфликты, поэтому при определении ролей необходимо учитывать перспективы компромиссов и предупреждения появления противоречий. Это означает, что при назначении на руководящие роли должен учитываться не только формальный статус исполнителя, но и его профессиональные и личностные характеристики.

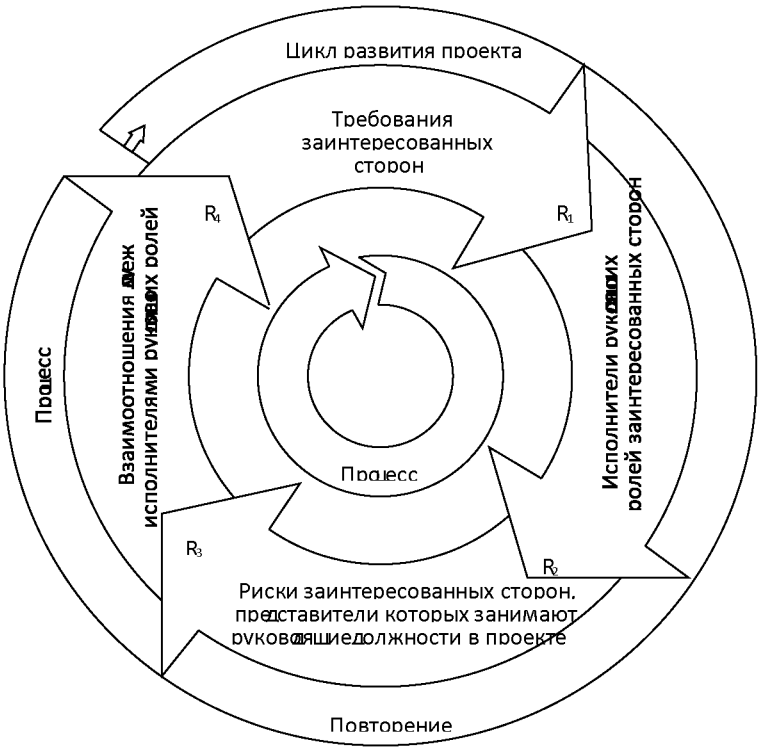
Третий шаг включает определение и анализ рисков и их распределения между руководящими ролями. Как правило, заинтересованные стороны проекта ставят свои частные интересы выше общих интересов проекта. Динг отмечает, что такое поведение нельзя объяснять понятиями честности или нечестности и что данный вопрос не решается на уровне контрактов или с помощью интриг. Отношение к проекту той или иной стороны может меняться в силу объективных причин, и единственный способ справиться с подобными проблемами — последовательно выявлять факторы рисков и управлять ими (определять инструменты измерения и снижения рисков, отслеживать состояние).

Четвертый шаг предполагает построение взаимоотношений между исполнителями руководящих ролей в проекте. Главной задачей этого шага является создание надежного альянса заинтересованных сторон и сохранения взаимного доверия. Поскольку формальный инструмент, регулирующий их взаимоотношения — контракт — этой задачи, как правило, не решает, акцент должен быть сделан на постоянном снижении неопределенности в проекте. Для этого могут быть использованы различные подходы в области планирования, организации управления, в технической области проекта. Одним из важнейших аспектов здесь является определение ролей управляющих рисками и выбор метода контроля их деятельности.

В течение жизненного цикла проекта ситуация в области заинтересованных сторон может измениться полностью: меняется состав

сторон, их требования, отношение к проекту, а также отношение друг к другу. Это требует постоянного (циклического) повторения как отдельных шагов процесса (требования, роли, риски и взаимоотношения), так и всего процесса в целом

Рис. 7. Схематическое описание, реализуемое для процессов в цикле проекта



Лекция 4. Документирование

Стандарт к оформлению документов. IEEE-830 Методика составления функциональной спецификаций.

Таблица 4.1. Стандарты оформления

Название раздела	Содержание
Введение	Цели Соглашения о терминах Предполагаемая аудитория и последовательность восприятия Масштаб проекта Ссылки на источники
Общее описание	Видение продукта Функциональность продукта Классы и характеристики пользователей Среда функционирования продукта (операционная среда) Рамки, ограничения, правила и стандарты Документация для пользователей Допущения и зависимости
Функциональность системы	Функциональный блок X (таких блоков может быть несколько) Описание и приоритет Причинно-следственные связи, алгоритмы (движение процессов, workflows) Функциональные требования

Продолжение таблицы 4.1:

Требования к внешним интерфейсам	Интерфейсы пользователя (UX) Программные интерфейсы Интерфейсы оборудования Интерфейсы связи и коммуникации
Нефункциональные требования	Требования к производительности Требования к сохранности (данных) Критерии качества программного обеспечения Требования к безопасности системы
Прочие требования	Приложение А: Глоссарий Приложение Б: Модели процессов предметной области и другие диаграммы Приложение В: Список ключевых задач

Лекция 5. Контроль версий Git

Основное предназначение Git – это сохранение снимков последовательно улучшающихся состояний вашего проекта (Pro git, 2019).

Эта статья для тех, кто имеет по крайней мере базовые знания и навык работы с git и желает расширить свои знания. Здесь рассматриваются только технические аспекты git'a, для более подробного погружения в философию git'a и его внутреннюю реализацию, советую прочитать несколько полезных книг (см. Рекомендуемая литература).

1. Настройка git

Прежде чем начинать работу с git необходимо его настроить под себя!

1.1 Конфигурационные файлы

- /etc/gitconfig — Общие настройки для всех пользователей и репозиториев;
- ~/.gitconfig или ~/.config/git/config — Настройки конкретного пользователя;
- .git/config — Настройки для конкретного репозитория.

Есть специальная команда git config [<опции>], которая позволит вам изменить стандартное поведение git, если это необходимо, но вы можете редактировать конфигурационные файлы вручную.

В зависимости какой параметр вы передадите команде git config (--system, --global, --local), настройки будут записываться в один из этих файлов. Каждый из этих “уровней” (системный, глобальный, локальный) переопределяет значения предыдущего уровня! Что бы посмотреть в каком файле, какие настройки установлены используйте git config --list --show-origin.

Игнорирование файлов. В git вы сами решаете какие файлы и в какой коммит попадут, но возможно вы бы хотели, что бы определённые файлы никогда не попали в индекс и в коммит, да и вообще не отображались в списке не отлеживаемых. Для этого вы можете создать специальный файл (.gitignore) в вашем репозитории и записать туда шаблон игнорируемых файлов. Если вы не

хотите создавать такой файл в каждом репозитории вы можете определить его глобально с помощью core.excludesfile (см. Полезные настройки). Вы также можете скачать готовый .gitignore file для языка программирования на котором вы работаете. Для настройки .gitignore используйте регулярные выражения bash.

1.2 Настройки по умолчанию

Есть куча настроек git'a как для сервера, так и для клиента, здесь будут рассмотрены только основные настройки клиента. Используйте git config name value, где name это название параметра, a value его значение, для того чтобы задать настройки.

Пример:

git config --global core.editor nano установит редактор по умолчанию nano. Вы можете посмотреть значение существующего параметра с помощью git config --get [name] где name это параметр, значение которого вы хотите получить. Полезные настройки:

- user.name — Имя, которое будет использоваться при создании коммита;
- user.email — Email, который будет использоваться при создании коммита;
- core.excludesfile — Файл, шаблон которого будет использоваться для игнорирования определённых файлов глобально;
- core.editor — Редактор по умолчанию;
- commit.template — Файл, содержимое которого будет использоваться для сообщения коммита по умолчанию (См. Работа с коммитами);
- help.autocorrect — При установке значения 1, git будет выполнять неправильно написанные команды;
- credential.helper [mode] — Устанавливает режим хранения учётных данных. [cache] — учётные данные сохраняются на определённый период, пароли не сохраняются (--timeout [seconds] количество секунд, после которого данные удаляются, по умолчанию 15 мин). [store] — учётные данные сохраняются на неограниченное время в открытом виде (--file [file] указывает путь для хранения данных, по умолчанию ~/.git-credentials).

1.3 Псевдонимы (aliases)

Если вы не хотите печатать каждую команду для Git целиком, вы легко можете настроить псевдонимы. Для создания псевдонима используйте:

git config alias.SHORT_NAME COMMAND,

где SHORT_NAME это имя для сокращения, а COMMAND команда(ы) которую нужно сократить.

Пример: git config --global alias.last 'log -1 HEAD' после выполнения этой команды вы можете просматривать информацию о последнем коммите на текущей ветке выполнив git last.

Я советую вам использовать следующие сокращения (вы также можете определить любые свои):

- st = status;
- ch = checkout;
- br = branch;
- mg = merge;
- cm = commit;
- reb = rebase;
- lg = «git log --pretty=format:'%h — %ar: %s'».

Для просмотра настроек конфигурации используйте: git config --list.

2. Основы git

Здесь перечислены только обязательные и полезные (на мой взгляд) параметры, ибо перечисление всех неуместно. Для этого используйте git command -help или --help, где command — название команды справку о который вы хотите получить.

2.1 Создание репозитория

- git init [<опции>] — Создаёт git репозитории и директорию .git в текущей директории (или в директории указанной после --separate-git-dir <каталог-git>, в этом случае директория .git будет находится в другом месте);
- git clone [<опции>] [--] <репозиторий> [<каталог>] [-o, --origin <имя>] [-b, --branch <ветка>] [--single-branch] [--no-tags] [--separate-git-dir

<каталог-git>] [-c, --config <ключ=значение>] — Клонирует репозитории с названием origin (или с тем которое вы укажите -o <имя>), находясь на той ветке, на которую указывает HEAD (или на той которую вы укажите -b <ветка>). Также вы можете клонировать только необходимую ветку HEAD (или ту которую укажите в -b <ветка>) указав --single-branch. По умолчанию клонируются все метки, но указав --no-tags вы можете не клонировать их. После выполнения команды создаётся директория .git в текущей директории (или в директории указанной после --separate-git-dir <каталог-git>, в этом случае директория .git будет находится в другом месте).

2.2 Состояние файлов

Для просмотра состояния файлов в вашем репозитории используйте: git status [<опции>]. Эта команда может показать вам: на какой ветке вы сейчас находитесь и состояние всех файлов. Обязательных опций нет, из полезных можно выделить разве что -s которая покажет краткое представление о состоянии файлов.

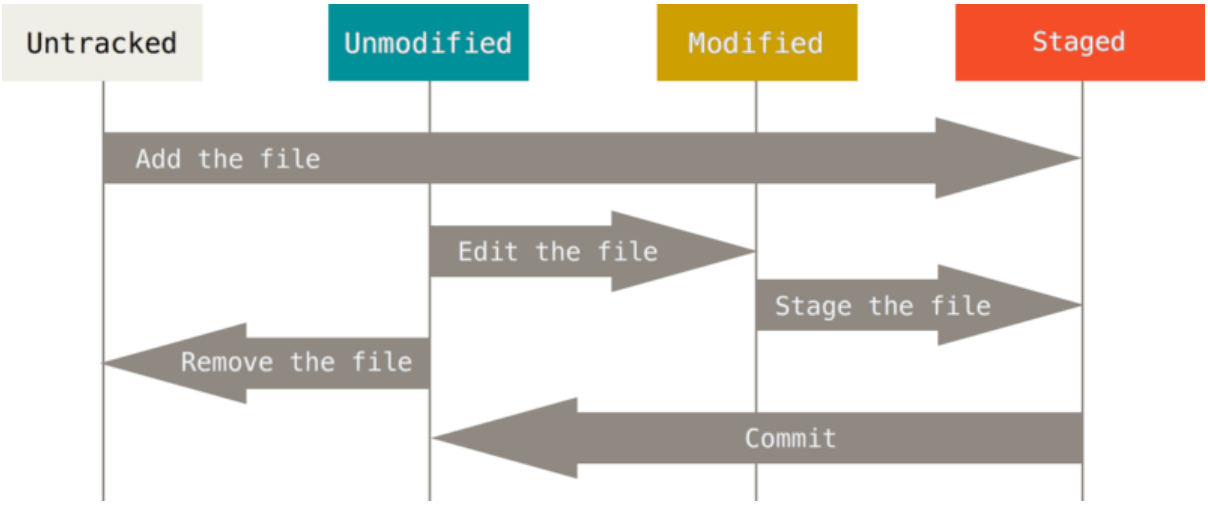


Рис. 8. Жизненный цикл файлов

Как видно на картинке файлы могут быть не отслеживаемые (Untracked) и отслеживаемые. Отслеживаемые файлы могут находиться в 3 состояниях: Не изменено (Unmodified), изменено (Modified), подготовленное (Staged). Если вы добавляете (с помощью git add) «Не отслеживаемый» файл, то он переходит в состояние «Подготовлено». Если вы изменяете файл в состояния «Не изменено», то он переходит в состояние «Изменено». Если вы сохраняете

изменённый файл (то есть находящийся в состоянии «Изменено») он переходит в состояние «Подготовлено». Если вы делаете коммит файла (то есть находящийся в состоянии «Подготовлено») он переходит в состояние «Не изменено». Если версии файла в HEAD и рабочей директории отличаются, то файл будет находиться в состоянии «Изменено», иначе (если версия в HEAD и в рабочем каталоге одинакова) файл будет находиться в состоянии «Не изменено». Если версия файла в HEAD отличается от рабочего каталога, но не отличается от версии в индексе, то файл будет в состоянии «Подготовлено». Этот цикл можно представить следующим образом:Unmodified -> Modified -> Staged -> Unmodified То есть вы изменяете файл сохраняете его в индексе и делаете коммит и потом все сначала.

2.3 Работа с индексом

Надеюсь, вы поняли, как выглядит жизненный цикл git репозитория. Теперь разберём как вы можете управлять индексом и файлами в вашем git репозитории.

Индекс — промежуточное место между вашим прошлым коммитом и следующим. Вы можете добавлять или удалять файлы из индекса. Когда вы делаете коммит в него попадают данные из индекса, а не из рабочей области. Чтобы просмотреть индекс, используйте git status. Чтобы добавить файлы в индекс используйте:

```
git add [<опции>]
```

Полезные параметры команды git add:

- -f, --force — добавить также игнорируемые файлы;
- -u, --update — обновить отслеживаемые файлы.

Чтобы удалить файлы из индекса вы можете использовать 2 команды git reset и git restore.git-restore — восстановит файлы рабочего дерева.git-reset — сбрасывает текущий HEAD до указанного состояния. По сути, вы можете добиться одного и того же с помощью обеих команд. Чтобы удалить из индекса некоторые файлы используйте: git restore --staged <file>, таким образом, вы

восстановите ваш индекс (или точнее удалите конкретные файлы из индекса), будто бы git add после последнего коммита не выполнялся для них. С помощью этой команды вы можете восстановить и рабочую директорию, чтобы она выглядела так, будто бы после коммита не выполнялось никаких изменений. Вот только эта команда имеет немного странное поведение — если вы добавили в индекс новую версию вашего файла вы не можете изменить вашу рабочую директорию, пока индекс отличается от HEAD. Поэтому вам сначала нужно восстановить ваш индекс и только потом рабочую директорию. К сожалению, сделать это одной командой невозможно, так как при передаче обеих аргументов (git restore -SW) не происходит ничего. И точно также при передаче -W тоже ничего не произойдет если файл в индексе и HEAD разный. Наверное, это сделали для защиты что бы вы случайно не изменили вашу рабочую директорию. Но в таком случае почему аргумент -W передаётся по умолчанию? В общем мне не понятно зачем было так сделано и для чего вообще была добавлена эта команда. По мне так reset справляется с этой задачей намного лучше, да и еще и имеет более богатый функционал так как может перемещать индекс и рабочую директорию не только на последний коммит, но и на любой другой.

Но собственно разработчики рекомендуют для сброса индекса использовать именно git restore -S . Вместо git reset HEAD.

С помощью git status вы можете посмотреть какие файлы изменились, но если вы также хотите узнать, что именно изменилось в файлах то воспользуйтесь командой:

```
git diff [<options>],
```

таким образом, выполнив команду без аргументов, вы можете сравнить ваш индекс с рабочей директорией. Если вы уже добавил в индекс файлы, то используйте git diff --cached что бы посмотреть различия между последним коммитом (или тем который вы укажите) и рабочей директории. Вы также можете посмотреть различия между двумя коммитами или ветками передав их

как аргумент. Пример: git diff 00656c 3d5119 покажет различия между коммитом 00656c и 3d5119.

2.4 Работа с коммитами

Теперь, когда ваш индекс находится в нужном состояний, пора сделать коммит ваших изменений. Запомните, что все файлы, для которых вы не выполнили git add после момента редактирования — не войдут в этот коммит. На деле файлы в нём будут, но только их старая версия (если таковая имеется). Для того, чтобы сделать коммит ваших изменений используйте:

```
git commit [<опции>].
```

Полезные опции команды git commit:

- -F, --file [file] — Записать сообщение коммита из указанного файла;
- --author [author] — Подменить автора коммита;
- --date [date] — Подменить дату коммита;
- -m, --mesage [message] — Сообщение коммита;
- -a, --all — Закоммитеть все изменения в файлах;
- -i, --include [files...] — Добавить в индекс указанные файлы для следующего коммита;
- -o, --only [files...] — Закоммитеть только указанные файлы;
- --amend — Перезаписать предыдущий коммит.

Вы можете определить сообщение для коммита по умолчанию с помощью commit.template. Эта директива в конфигурационном файле отвечает за файл, содержимое которого будет использоваться для коммита по умолчанию. Пример: git config --global commit.template ~/.gitmessage.txt. Вы также можете изменить, удалить, объединить любой коммит. Как вы уже могли заметить вы можете быстро перезаписать последний коммит с помощью git commit --amend. Для изменения коммитом в вашей истории используйте:

```
git rebase -i <commit>,
```

где commit это верхний коммит в вашей цепочке, с которого вы бы хотели что-либо изменить.

После выполнения git rebase -i в интерактивном меню выберите, что вы хотите сделать:

- pick <коммит> = использовать коммит;
- reword <коммит> = использовать коммит, но изменить сообщение коммита;
- edit <коммит> = использовать коммит, но остановиться для исправления;
- squash <коммит> = использовать коммит, но объединить с предыдущим коммитом;
- fixup <коммит> = как «squash», но пропустить сообщение коммита;
- exec <команда> = выполнить команду (остаток строки) с помощью командной оболочки;
- break = остановиться здесь (продолжить с помощью «git rebase --continue»);
- drop <коммит> = удалить коммит;
- label <метка> = дать имя текущему HEAD;
- reset <метка> = сбросить HEAD к указанной метке.

Для изменения сообщения определённого коммита. Необходимо изменить pick на edit над коммитом который вы хотите изменить. Пример: вы хотите изменить сообщение коммита 750f5ae.

pick 2748cb4 first commit
edit 750f5ae second commit
pick 716eb99 third commit

После сохранения скрипта вы вернётесь в командную строку и git скажет, что необходимо делать дальше:

Остановлено на 750f5ae … second commit

You can amend the commit now, with git commit –amend

Once you are satisfied with your changes, run git rebase –continue

Как указанно выше, необходимо выполнить git commit --amend для того, чтобы изменить сообщение коммита. После чего выполнить git rebase --continue. Если вы выбрали несколько коммитов для изменения названия, то данные операций необходимо будет проделать над каждым коммитом.

Для удаления коммита необходимо удалить строку с коммитом.

Пример: вы хотите удалить коммит 750f5ae

Нужно изменить скрипт с такого:

pick 2748cb4 third commit
pick 750f5ae second commit
pick 716eb99 first commit
на такой:
pick 2748cb4 first commit
pick 716eb99 third commit

Для объединения коммитов необходимо изменить pick на squash над коммитами, которые вы хотите объединить.

Пример: вы хотите объединить коммиты 750f5ae и 716eb99.

Необходимо изменить скрипт с такого:

pick 2748cb4 third commit
pick 750f5ae second commit
pick 716eb99 first commit
На такой
pick 2748cb4 third commit
squash 750f5ae second commit
squash 716eb99 first commit

Заметьте, что в интерактивном скрипте коммиты изображены в обратном порядке нежели в git log. С помощью squash вы объедините коммит 750f5ae с

716eb99, а 750f5ae с 2748cb4. В итоге получая один коммит содержащий изменения всех трёх.

2.5 Просмотр истории

С помощью команды

```
git log [<опции>] [<диапазон-редакций>]
```

вы можете просматривать историю коммитов вашего репозитория. Есть также куча параметров для сортировки и поиска определённого коммита.

Полезные параметры команды git log:

- -p — Показывает разницу для каждого коммита.
- --stat — Показывает статистику измененных файлов для каждого коммита.
- --graph — Отображает ASCII граф с ветвлениями и историей слияний.

Также можно отсортировать коммиты по времени, количеству и т.д.:

- -(n) Показывает только последние n коммитов.
- --since, --after — Показывает коммиты, сделанные после указанной даты.
- --until, --before — Показывает коммиты, сделанные до указанной даты.
- --author — Показывает только те коммиты, в которых запись author совпадает с указанной строкой.
- --committer — Показывает только те коммиты, в которых запись committer совпадает с указанной строкой.
- --grep — Показывает только коммиты, сообщение которых содержит указанную строку.
- -S — Показывает только коммиты, в которых изменение в коде повлекло за собой добавление или удаление указанной строки.

Вот несколько примеров:

- git log --since=3.weeks — Покажет коммиты за последние 2 недели

- git log --since=«2019-01-14» — Покажет коммиты сделанные 2019-01-14
- git log --since=«2 years 1 day ago» — Покажет коммиты сделанные 2 года и один день назад.

Также вы можете настроить свои формат вывода коммитов с помощью

```
git log --format:["format"].
```

Варианты форматирования для git log –format:

- %H — Хеш коммита;
- %h — Сокращенный хеш коммита;
- %T — Хеш дерева;
- %t — Сокращенный хеш дерева;
- %P — Хеш родителей;
- %p — Сокращенный хеш родителей;
- %an — Имя автора — %ae — Электронная почта автора;
- %ad — Дата автора (формат даты можно задать опцией --date=option);
- %ar — Относительная дата автора;
- %cn — Имя коммитера;
- %ce — Электронная почта коммитера;
- %cd — Дата коммитера;
- %cr — Относительная дата коммитера;
- %s — Содержание.

Пример:

```
git log --pretty=format:"%h - %ar : %s"
```

покажет список коммитов состоящий из хэша времени и сообщения коммита.

2.6 Работа с удалённым репозиторием

Так как git – это распределённая СКВ, вы можете работать не только с локальными, но и с внешними репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые на внешнем сервере.

Для работы с внешними репозиториями используйте:

```
git remote [<options>].
```

Если вы с клонировали репозитории через http URL то у вас уже имеется ссылка на внешний. В другом случае вы можете добавить её с помощью:

```
git remote add [<options>] <name> <adres>.
```

Вы можете тут же извлечь внешние ветки с помощью -f, --fetch (вы получите имена и состояние веток внешнего репозитория). Вы можете настроить репозитории только на отправку или получение данных с помощью --mirror[=(push|fetch)]. Для получения меток укажите --tags.

Для просмотра подключённых внешних репозиториев используйте git remote без аргументов или git remote -v для просмотра адресов на отправку и получение данных от репозитория. Для отслеживания веток используйте git branch -u <rep/br> где rep это название репозитория, br название внешней ветки, а branch название локальной ветки. Либо git branch --set-upstream local_br origin/br для того, чтобы указать какая именно локальная ветка будет отслеживать внешнюю ветку. Когда ваша ветка отслеживает внешнюю вы можете узнать какая ветка (локальная или внешняя) отстаёт или опережает и на сколько коммитов. К примеру, если после коммита вы не выполняли git push то ваша ветка будет опережать внешнюю на 1 коммит. Вы можете узнать об этом выполнив git branch -vv, но прежде выполните git fetch [remote-name] (--all для получения обновления со всех репозиториев) что бы получить актуальные данные из внешнего репозитория. Для отмены отслеживания ветки используйте git branch --unset-upstream [<local_branch>].

Для загрузки данных с внешнего репозитория используйте git pull [rep] [branch]. Если ваши ветки отслеживают внешние, то можете не указывать их при выполнении git pull. По умолчанию вы получите данные со всех отслеживаемых веток.

Для загрузки веток на новую ветку используйте git checkout -b <new_branch_name> <rep/branch>. Для отправки данных на сервер используйте:

```
git push [<rep>] [<br>],
```

где rep – это название внешнего репозитория, а br – локальная ветка, которую вы хотите отправить. Также вы можете использовать такую запись git push origin master:dev. Таким образом, вы выгрузите вашу локальную ветку master на origin (но там она будет называется dev). Вы не сможете отправить данные во внешний репозитории если у вас нет на это прав. Также вы не сможете отправить данные на внешнюю ветку, если она опережает вашу (в общем-то, отправить вы можете используя -f, --forse – в этом случае вы перепишите историю на внешнем репозитории). Вы можете не указывать название ветки, если ваша ветка отслеживает внешнюю.

Для удаления внешних веток используйте git push origin --delete branch_name. Для получения подробной информации о внешнем репозитории (адреса для отправки и получения, на что указывает HEAD, внешние ветки, локальные ветки настроенные для git pull и локальные ссылки настроенные для git push):

git remote show <remote_name>.

Для переименования названия внешнего репозитория используйте:

git remote rename <last_name> <new_name>.

Для удаления ссылки на внешний репозитории используйте:

```
git remote rm <name>.
```

3. Ветвление в git

Ветвление – это мощные инструмент и одна из главных фич git'a, поскольку позволяет вам быстро создавать и переключаться между различным ветками вашего репозитория. Главная концепция ветвления состоит в том, что вы можете откланяться от основной линии разработки и продолжать работу независимо от нее, не вмешиваясь в основную линию. Ветка всегда указывает на последний коммит в ней, а HEAD указывает на текущую ветку (см. Указатели в git).

3.1 Базовые операций

Для создания ветки используйте git branch <branch_name> [<start_commit>]

Здесь branch_name это название для новой ветки, а start_commit это коммит на который будет указывать ветка (то есть последний коммит в ней). По умолчанию ветка будет находиться на последнем коммите родительской ветки.

Опции git branch:

- -r | -a [--merged | --no-merged] — Список отслеживаемых внешних веток -r. Список и отслеживаемых и локальных веток -a. Список слитых веток --merged. Список не слитых веток --no-merged.
- -l, -f <имя-ветки> [<точка-начала>] — Список имён веток -l. Принудительное создание, перемещение или удаление ветки -f. Создание новой ветки <имя ветки>.
- -r (-d | -D) — Выполнить действие на отслеживаемой внешней ветке -r. Удалить слитую ветку -d. Принудительное удаление (даже не слитой ветки) -D.
- -m | -M [<Старая ветка>] <Новая ветка> — Переместить/переименовать ветки и ее журнал ссылок (-m). Переместить/переименовать ветку, даже если целевое имя уже существует -M.
- (-c | -C) [<старая-ветка>] <новая-ветка> — Скопировать ветку и её журнал ссылок -c. Скопировать ветку, даже если целевое имя уже существует -C.

- -v, -vv — Список веток с последним коммитом на ветке -v. Список и состояние отслеживаемых веток с последним коммитом на них.

Больше информации смотрите в git branch -h | --help. Для переключения на ветку используйте git checkout. Также вы можете создать ветку выполнив git checkout -b <ветка>.

3.2 Слияние веток

Для слияния 2 веток git репозитория используйте git merge. Полезные параметры для git merge:

- --squash — Создать один коммит вместо выполнения слияния. Если у вас есть конфликт на ветках, то после его устранения у вас на ветке прибавится 2 коммита (коммит со сливаемой ветки + коммит слияния), но указав этот аргумент у вас прибавится только один коммит (коммит слияния).
- --ff-only — Не выполнять слияние если имеется конфликт. Пусть кто ни будь другой разрешает конфликты :D
- -X [strategy] — Использовать выбранную стратегию слияния.
- --abort — Отменить выполнение слияния.

Процесс слияния. Если вы не выполняли на родительской ветке новых коммитов, то слияние сводится к быстрой перемотке «fast-forward», будто бы вы не создавали новую ветку, а все изменения происходили прямо тут (на родительской ветке).

Если вы выполняли коммиты на обеих ветках, но при этом не создали конфликт, то слияния пройдёт в «recursive strategy», то есть вам просто нужно будет создать коммит слияния что бы применить изменения (используйте опцию --squash что бы не создавать лишний коммит). Если вы выполняли коммиты на обоих ветках, которые внесли разные изменения в одну и ту же часть одного и того же файла, то вам придётся устранить конфликт и зафиксировать слияние коммитом.

При разрешении конфликта вам необходимо выбрать какую часть изменений из двух веток вы хотите оставить. При открытии конфликтующего

файла, в нём будет содержатся следующее: <<<<<<< HEAD. Тут будет версия изменения последнего коммита текущей ветки

Тут будет версия изменений последнего коммита сливаемой ветки >>>>>>>. Тут название ветки, с которой сливаем. Разрешив конфликт, вы должны завершить слияния выполнив коммит. Во время конфликта вы можете посмотреть какие различия в каких файлах имеются. git diff --ours — Разница до слияния и после git diff --theirs — Разница сливаемой ветки до слияния и после git diff --base — Разница с обеими ветками до слияния и после Если вы не хотите разрешать слияние то используйте различные стратегии слияния, выбрав либо «нашу» версию (то есть ту которая находится на текущей ветке) либо выбрать «их» версию находящуюся на сливаемой ветке при этом не исправляя конфликт. Выполните git merge --Xours или git merge --Xtheirs соответственно.

3.3 Rerere

Rerere — «reuse recorded resolution” — “повторное использование сохраненных разрешений конфликтов». Механизм rerere способен запомнить каким образом вы разрешали некую часть конфликта в прошлом и провести автоматическое исправление конфликта при возникновении его в следующий раз.

Что бы включить rerere выполните:

```
git config --global rerere.enabled true
```

Таrже вы можите включить rerere создав каталог .git/rr-cache в нужном репозитории. Используйте git rerere status для того, чтобы посмотреть для каких файлов rerere сохранил снимки состояния до начала слияния. Используйте git rerere diff для просмотра текущего состояния конфликта. Если во время слияния написано: Resolved 'nameFile' using previous resolution. Значит rerere уже устранил конфликт используя кэш.

Для отмены автоматического устранения конфликта используйте git checkout --conflict=merge таким образом вы отмените авто устранение конфликта и вернёте файл(ы) в состояние конфликта для ручного устранения.

4. Указатели в git

В git есть такие указатели как HEAD branch. По сути, всё очень просто HEAD указывает на текущую ветку, а ветка указывает на последний коммит в ней. Но для понимания лучше представлять, что HEAD указывает на последний коммит.

4.1 Перемещение указателей

В книге Pro git приводится очень хороший пример того, как вы можете управлять вашим репозиторием поэтому я тоже буду придерживаться его. Представите, что Git управляет содержимым трех различных деревьев. Здесь под «деревом» понимается “набор файлов”. В своих обычных операциях Git управляет тремя деревьями:

- HEAD — Снимок последнего коммита, родитель следующего
- Индекс — Снимок следующего намеченного коммита
- Рабочий Каталог — Песочница

Собственно, git предоставляет инструменты для манипулирования всеми тремя деревьями. Далее будет рассмотрена команда git reset, позволяющая работать с тремя деревьями вашего репозитория.

Используя различные опций этой команды, вы можете:

- --soft — Сбросить только HEAD
- --mixed — Сбросить HEAD и индекс
- --hard — Сбросить HEAD, индекс и рабочий каталог

Под сбросить понимается переместить на указанный коммит. По умолчанию выполняется --mixed.

Примеру 1. Вы сделали 3 лишних коммита каждый из которых приносит маленькие изменения, и вы хотите сделать из них один, таким образом вы можете с помощью git reset --soft переместить указатель HEAD при этом оставив индекс и рабочий каталог нетронутым и сделать коммит. В итоге в вашей истории будет выглядеть так, что все изменения произошли в одном коммите.

Пример 2. Вы добавили в индекс лишние файлы и хотите их оттуда убрать. Для этого вы можете использовать git reset HEAD <files...>. Или вы хотите, чтобы в коммите файлы выглядели как пару коммитов назад. Как я уже говорил ранее вы можете сбросить индекс на любой коммит в отличии от git restore который сбрасывает только до последнего коммита. Только с опцией mixed вы можете применить действие к указанному файлу!

Пример 3. Вы начали работать над новой фичей на вашем проекте, но вдруг работодатель говорит, что она более не нужна и вы в порыве злости выполняете git reset --hard возвращая ваш индекс, файлы и HEAD к тому моменту, когда вы ещё не начали работать над фичей. А на следующей день вам говорят, что фичу всё-таки стоит запилить. Но что же делать? Как же переместится вперёд ведь вы откатили все 3 дерева и теперь в истории с помощью git log их не найти. А выход есть — это журнал ссылок git reflog. С помощью этой команды вы можете посмотреть куда указывал HEAD и переместится не только вниз по истории коммитов, но и вверх. Этот журнал является локальным для каждого пользователя.

Лекция 6. Диаграммы. Нотация BPMN

Нотация BPMN позволяет начать с разработки высокоуровневой аналитической модели, которая дает общее представление о характере исполнения бизнес-процесса. По мере роста понимания того, как должен исполняться бизнес-процесс, модель расширяется, уточняется и углубляется. Результатом моделирования становится исполняемая модель бизнес-процесса. Преимущество нотации BPMN заключается в том, что она позволяет описать поведение ИТ системы с минимальным программированием. Таким образом, большую часть работ по созданию исполняемой модели может выполнить бизнес-аналитик, без участия программистов и разработчиков. Применение нотации BPMN обеспечивает бизнес-пользователям ряд преимуществ. В первую очередь, это уменьшение разрыва между моделями «Как Есть» и «Как Должно Быть». Исполняемая модель помогает не только раскрыть и верифицировать модель бизнес-процесса, но и испытать ее в условиях реальной эксплуатации. Такие тесты позволяют эффективно выявлять и расшить узкие места процесса, найти более эффективные способы обработки информации. На практике это означает, что работа с процессом всегда начинается с модели «Как Есть», а переход к модели «Как Должно Быть» осуществляется путем проведения небольших эволюционных изменений, причем при постоянном контроле за изменением метрик процесса. Т.о. можно говорить, что полученные модели «Как Должно Быть» носят объективный и достоверный характер, поскольку базируются на результатах измерений, а не на субъективном представлении консультанта. Интересно отметить, если аналитик и разработчик в паре работают над единой моделью, они при этом видят разный набор слоев модели. Благодаря работе с единой моделью упрощается взаимодействие между участниками команды, что выгодно отличает предлагаемый подход от традиционного, где разные категория разработчиков использует различные модели, что приводит к непониманию и ошибкам. Бизнес-пользователи видят эту же модель. Благодаря этому удается сблизить

представления бизнес-пользователей и разработчиков о модели процесса. Простота моделирования позволяет бизнес-аналитику с минимальной помощью разработчика не только создать работающий прототип, но и протестировать его работу, на самом раннем этапе выявить степень соответствия модели реальному бизнес-процессу и таким образом сделать процедуру верификации бизнес-процесса более объективной.

ОБЛАСТЬ ПРИМЕНЕНИЯ НОТАЦИИ BPMN

Нотация BPMN предназначена для описания:

• Порядка исполнения работ, образующих бизнес-процесс;

• Потоков данных между операциями процесса;

• Потоков сообщений между процессами;

• Ассоциации обрабатываемых объектов данных с операциями процесса.

Моделирование осуществляется с помощью визуальных диаграмм, что позволяет участникам быстрее понять логику исполнения.

Нотация BPMN не позволяет моделировать другие аспекты модели бизнес-процесса, например:

• Функциональную (структурную) декомпозицию работ;

• Организационную структуру предприятия;

• Модель данных;

• Бизнес правила,

• Бизнес-стратегию компании.

Поскольку интегрированная модель бизнес-процесса включает не только поведенческую перспективу, но также другие аспекты, описываемые перечисленными моделями, спецификация BPMN уделяет повышенное внимание вопросам интеграции моделей. Интеграция осуществляется на уровне метамодели BPMN.

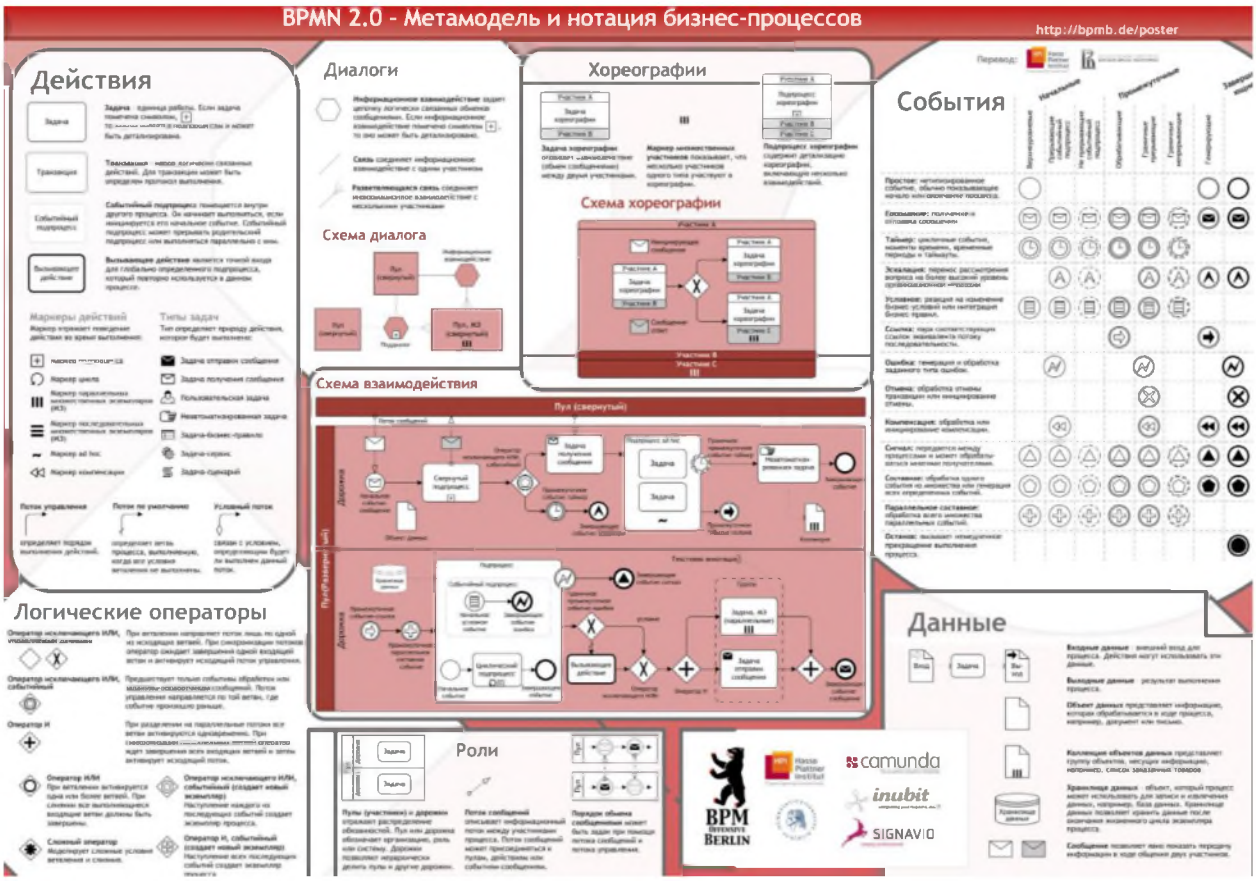


Рис. 9. Метамодель и нотация бизнес-процессов

Лекция 7. Диаграммы. Нотация UML

UML предназначен для моделирования. Сами авторы UML определяют свое детище следующим образом.

Язык UML – это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем.

Мы полностью согласны с этим определением, и не только одобряем выбор ключевых слов, но придаем большое значение порядку, в котором они перечислены.

В типичных случаях в процессе разработки приложений участвуют по меньше мере два действующих лица: заказчик (конкретный человек или группа лиц, или организация) и разработчик (это может быть программист-одиночка, временная команда проекта или целая организация, специализирующаяся на разработке программного обеспечения). Из-за того, что действующих лиц двое, очень многое зависит от степени их взаимопонимания.

Одним из ключевых этапов разработки приложения является определение того, каким требованиям должно удовлетворять разрабатываемое приложение. В результате этого этапа появляется формальный или неформальный документ (артефакт), который называют по-разному, имея в виду примерно одно и то же: постановка задачи, требования, техническое задание, внешние спецификации и др.

Аналогичные по назначению, но, может быть, отличные по форме и содержанию артефакты появляются и на других этапах разработки, особенно если в разработку включено много действующих лиц. Для них также используются различные названия: функциональные спецификации, архитектура приложения и др. Мы будем все такие артефакты называть спецификациями.

Спецификация – это декларативное описание того, как нечто устроено или работает.

Необходимо принимать во внимание три толкования спецификаций.

То, которое имеет в виду действующее лицо, являющееся источником спецификации (например, заказчик).

То, которое имеет в виду действующее лицо, являющееся потребителем спецификации (например, разработчик).

То, которое объективно обусловлено природой специфицируемого объекта.

Эти три трактовки спецификаций могут не совпадать, и, к сожалению, как показывает практика, сплошь и рядом не совпадают, причем значительно. Заказчик может не осознавать своих объективных потребностей, или неверно их интерпретировать, или заблуждаться относительно природы своих затруднений, пытаясь с помощью заказного офисного приложения лечить симптомы, а не причину болезни своего бизнеса. Разработчик может не разбираться в предметной области заказчика и интерпретировать формулировки спецификаций совершенно превратным образом. Если же в формулировке спецификаций участвует разработчик, то злоупотребление технической терминологией может совершенно дезориентировать заказчика.

Основное назначение UML – предоставить, с одной стороны, достаточно формальное, с другой стороны, достаточно удобное, и, с третьей стороны, достаточно универсальное средство, позволяющее до некоторой степени снизить риск расхождений в толковании спецификаций.

Визуализация

Известная поговорка гласит, что лучше один раз увидеть, чем сто раз услышать. Мы добавим: тем паче тысячу раз прочитать пересказ. Особенности человеческого восприятия таковы, что текст с картинками воспринимается легче, чем голый текст. А картинки с текстом (это называется «комиксы») – еще легче. Модели UML допускают представление в форме картинок, причем эти картинки наглядны, интуитивно понятны, практически однозначно

интерпретируются и легко составляются. Фактически, развитие и детализация этого тезиса составляет большую часть содержания остальной части книги. Мы не будем забегать вперед, и просто приведем пример без всяких объяснений.

Жизненный цикл работника на предприятии

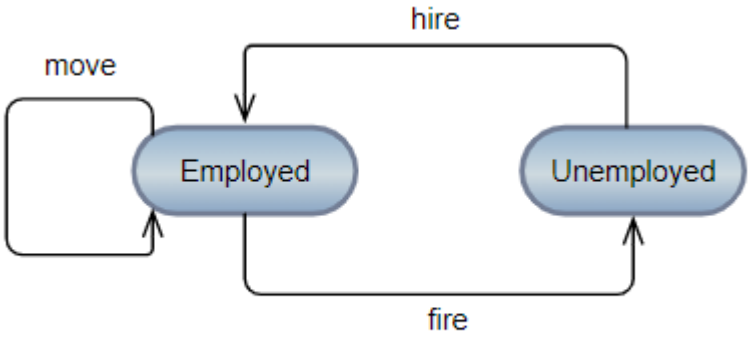


Рис. 10. Жизненный цикл работника на предприятии

Таким образом, второе по важности назначение UML состоит в том, чтобы служить адекватным средством коммуникации между людьми. Разумеется, наглядность визуализации моделей UML имеет значение, только если они должны составляться или восприниматься человеком – это назначение UML не имеет отношения к компьютерам.

Графическое представление модели UML не тождественно самой модели. Это важное обстоятельство часто упускается из виду при первом знакомстве с UML.

Проектирование

В оригинале данное назначение UML определено с помощью слова construct, которое мы передаем осторожным термином "проектирование". Речь идет о том, что UML предназначен не только для описания абстрактных моделей приложений, но и для непосредственного манипулирования артефактами, входящими в состав этих приложений, в том числе такими, как программный код. Другими словами, одним из назначений UML является, например, создание таких моделей, для которых возможна автоматическая генерация программного кода (или фрагментов кода) соответствующих приложений. Более того, природа моделей UML такова, что возможен и

обратный процесс: автоматическое построение модели по коду готового приложения.

По-английски автоматическое построение модели по коду готового приложения называется reverse engineering и обычно переводится на русский как "обратное проектирование". Нам этот перевод категорически не нравится: какое же это проектирование и почему оно обратное? Есть неплохой альтернативный вариант: "инженерный анализ программ", но он не получил пока распространения.

Сказанное в предыдущем абзаце требует оговорок "до некоторой степени", "в известной мере" буквально после каждого утверждения. Самое досадное, что в данный момент точно указать "степень" и "меру" не представляется возможным. Причина не в том, что никто не удосужился этим заняться, а в том, что это очень трудная задача, но не безнадежная. Инструменты, поддерживающие UML, все время совершенствуются, так что в перспективе третье предназначение UML может выйти и на первое место.

Некоторым уставшим от бесконечной отладки разработчикам может показаться, что стоит изучить UML, и все проблемы программирования будут решены. К сожалению, это не так.

Документирование

Модели UML являются артефактами, которые можно хранить и использовать как в форме электронных документов, так и в виде твердой копии. В последних версиях UML с целью достижения более полного соответствия этому назначению сделано довольно много. В частности, специфицировано представление моделей UML в форме документов в формате XMI, что обеспечивает практическую интероперабельность при работе с моделями. Другими словами, модели UML не являются вещью в себе, которой можно только любоваться – это документы, которые можно использовать самыми разными способами, начиная с печати картинок и заканчивая автоматической генерацией человекочитаемых текстовых описаний.

XMI (XML Metadata Interchange) – внешний формат данных, основанный на языке XML (схема и набор правил использования тэгов), предназначенное для сериализации моделей и обмена ими.

Поясним последнюю фразу предыдущего абзаца. Стандарт требует, чтобы во внутреннем представлении модели для каждого элемента моделирования было отведено место, где можно хранить неформальное текстовое описание этого элемента. Большинство инструментов это требование выполняют: буквально для каждой линии или фигуры на диаграмме можно ввести текст, который поясняет смысл и назначение именно этой линии или фигуры. Более того, многие инструменты умеют из этих текстовых описаний собирать цельные, вполне осмысленные и хорошо отформатированные текстовые документы, которые можно использовать именно как привычные текстовые описания моделируемой системы. К сожалению, это замечательная возможность на практике используется меньше, чем она того заслуживает. Дело в том, что так же, как программисты не любят и ленятся писать осмысленные комментарии к программному коду, так и архитекторы не любят и ленятся писать текстовые пояснения к своим диаграммам.

Инструментальная поддержка

Рассмотрим, как соотносится сегодняшняя практика использования UML с декларированным выше назначением языка.

Проводя в жизнь принцип обучения на примерах, для иллюстрации вышесказанного обратимся к одной из диаграмм UML – диаграмме использования.

По мнению авторов, можно выделить три основных варианта использования UML.

Инструментальная поддержка

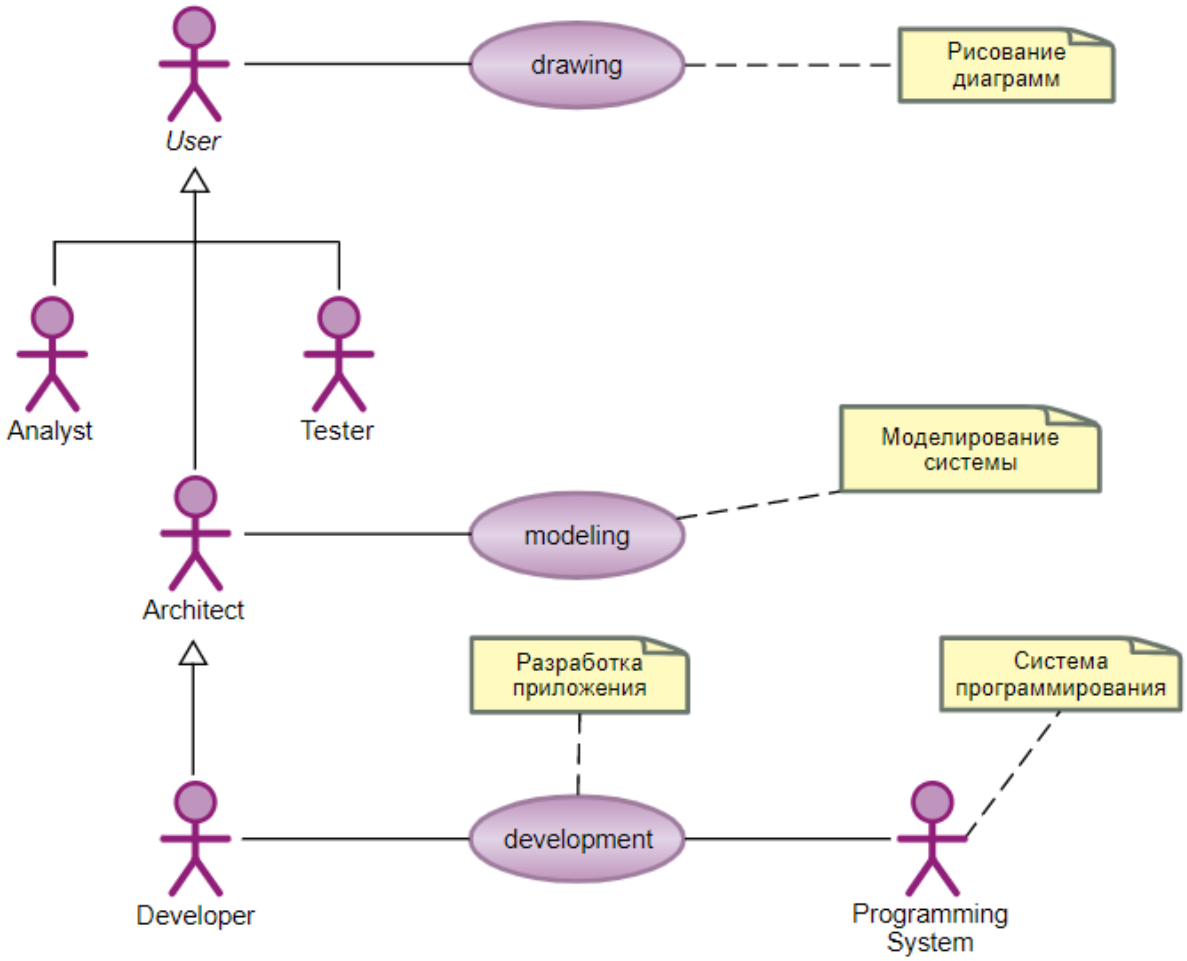


Рис. 11. Инструментальная поддержка

Вариант использования drawing («Рисование диаграмм») подразумевает изображение диаграмм UML с целью обдумывания, обмена идеями между людьми, документирования и тому подобного. Значимым для пользователя User результатом в этом случае является само изображение диаграмм. Вообще говоря, в этом варианте использования языка поддерживающий инструмент не очень нужен. Иногда рисование диаграмм от руки фломастером с последующим фотографированием цифровым аппаратом может оказаться практичнее.

Вариант использования modeling («Моделирование систем») подразумевает создание и изменение модели системы в терминах тех элементов моделирования, которые предусматриваются метамоделью UML. Значимым результатом в этом случае является машинно-читаемый артефакт с описанием модели. Мы будем для краткости называть такой артефакт просто моделью, деятельность по составлению модели называть моделированием, а субъекта моделирования называть архитектором Architect.

Вариант использования development ("Разработка приложений") подразумевает детальное моделирование, реализацию и тестирование приложения в терминах UML. Значимым для пользователя Developer результатом в этом случае является работающее приложение, которое может быть скомпилировано в язык, поддерживаемый конкретной системой программирования Programming System или сразу интерпретировано средой выполнения инструмента. Этот вариант использования наиболее сложен в реализации.

Лекция 8. Сетевая архитектура

Направлением развития современных информационно-вычислительных сетей (ИВС) является их глобализация и объединение (интеграция). Это приводит к расширению ИВС, совместному использованию программного обеспечения (ПО), объединению различных сетей и т.п.

До недавнего времени пользователи были вполне удовлетворены теми возможностями, которые им предоставляли базовые технологии таких локальных сетей как Ethernet, Token Ring и глобальные сети стандарта X.25. При этом степень развития этих систем не требовала какой – либо унификационной основы.

Базовая технология – согласованный набор (стек) протоколов и реализующих их программно-аппаратных средств, достаточный для построения любой сетевой архитектуры.

Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов сети любой базовой технологии называется стеком коммуникационных протоколов.

Протоколы, обеспечивающие работу сети, специально разрабатывались, для совместной работы, поэтому при создании сети не требовалось прилагать каких –либо усилий для обеспечения совместимости. Вычислительная сеть продолжала исправно работать после соединения аппаратных и программных средств в соответствии с требованиями стандарта, относящегося к данной базовой технологии.

Однако появление новых сетевых технологий или архитектур, таких как Fast Ethernet, 100VG –AnyLan, FDDI, Giga-Ethernet привело к повышению неоднородности при объединении локальных сетей.

К этому времени резко расширился список фирм - производителей сетевого оборудования. Поэтому сразу возникли проблемы несовместимости оборудования.

Все эти причины привели к необходимости разработки базовой модели сетевого взаимодействия.

В 1984 году с целью упрощения взаимодействия устройств в вычислительных сетях Международная организация по стандартизации (Internetional Organization of Standardization -ISO) предложила семиуровневую эталонную коммуникационную модель «Взаимодействия Открытых Систем» (Open Sistem Interconnection-OSI).

Модель OSI стала основой новых разработок в области сетевой коммуникации и позволила соединить в единую сеть все многообразие выпускаемых программно-аппаратных средств. В документах Международного Союза Электросвязи (Internetional Telecommunications Union, ITU) модель OSI нашла отражение в рекомендации X.200.

Кроме того, в отечественной литературе стандарт OSI называют также Эталонной моделью взаимодействия открытых систем (ЭМВОС), но более чаще её называют моделью ISO/OSI, отмечая вклад ISO в её формирование.

Модель OSI является самой методологической основой для изучения сетевых технологий.

Основная идея модели OSI в том, что одинаковые уровни различных систем, не имея возможности связываться непосредственно, должны выполнять свои функции абсолютно одинаково и в одном объеме. Таким же одинаковым должен быть и сервис между соответствующими уровнями различных систем. Нарушение этого принципа может привести к тому, что информация посланная от одной системы к другой, после всех преобразований будет непохожа на исходную.

Эталонная модель OSI не является реализацией сети, она только определяет функции каждого уровня и дает простое представление о движении данных в сети. И, наконец, не все уровни нуждаются в присоединении заголовков. Некоторые просто выполняют трансформацию получаемых данных, чтобы сделать их формат подходящим для смежных с ними уровней.

Следует четко различать модель OSI и стек OSI, в то время как модель является концептуальной схемой взаимодействия открытых систем, стек OSI представляет собой набор вполне конкретных протоколов. В отличие от других стеков протоколов стек OSI полностью соответствует модели OSI.

Наиболее популярными протоколами являются прикладные протоколы. К ним относятся: протокол передачи файлов FTAM, протокол эмуляции терминала VTP, протоколы справочной службы X.500, электронной почты X.400 и ряд других.

Стек OSI- международный, независимый от производителей стандарт его поддерживает правительство США в своей программе GOSIP, в соответствии с которой компьютерные сети, устанавливаемые в правительственных учреждениях, должны поддерживать стек OSI. Из тех, кто работает в этом направлении, можно назвать Военно-морское ведомство США и сеть NFSNET. Одним из крупнейших производителей, поддерживающих OSI является копания AT&T, её сеть Stargroup полностью базируется на этом стеке.

Кроме OSI, наиболее популярными являются стандарты TCP/IP, IPX/STX, NETBIOS/SMB, Decnet, SNA, 3COM, Banyn VINES и т.д.

В большинстве случаев разработчики стандартов отдавали предпочтение скорости работы сети в ущерб модульности - ни один стек протоколов, кроме стека OSI не разбит на семь уровней. Чаще всего в стеке протоколов выделяется 3-4 уровня: уровень сетевых адаптеров, на котором реализуются протоколы физического и канального уровней, транспортный уровень и уровень служб, вбирающий в себя функции сеансового, представительного и прикладного уровней.

Используемые в настоящее время сетевые архитектуры на базе других стандартных моделей существенно различаются по принципам взаимодействия уровней, своим характеристикам и перечню предоставляемых услуг, что связано как с реальными потребностями пользователя, так и с развитием самих этих архитектур. В реальных сетях используется множество сетевых

архитектур, таких как TCP/IP, IPX/SPX, XNS XEROX, Apple Talk, SNA, Banyan VINES, ISO, 3COM, DECnet и ряд других.

Однако поистине всемирное распространение получили два подхода

— архитектура TCP/IP американского научно-исследовательского центра DARPA и архитектура сети на базе стандарта ISO (в дальнейшем просто ISO). Причём принципиальные отличия в методологических основах реализации этих архитектур проистекают из учёта качества используемых каналов связи. Так архитектура TCP/IP ориентирована на применение достаточно хороших каналов связи с низким коэффициентом ошибок (порядка 10-5), в то время как архитектура ISO допускает использование каналов с вероятностью ошибки порядка 10-3. Наконец, реализации ИВС на каналах очень хорошего качества (коэффициент ошибок порядка 10-7), например, волоконно-оптических или спутниковых, позволяют применять современные высокоскоростные сетевые технологии типа Frame Relay.

Поскольку основная задача ИВС общего пользования состоит в организации взаимодействия разнородных пользователей на значительных территориях, то главными требованиями к сетевой архитектуре являются: наличие мощной, открытой и гибкой системы адресации, позволяющей обеспечить обслуживание значительного количества пользователей; высокая эффективность передачи полезной информации в сети, как по времени, так и по верности доставки; высокая степень адаптации к изменяющимся внешним условиям (неисправности, подключение новых ресурсов или пользователей), что требует тщательно сбалансированной системы протоколов взаимодействия на всех уровнях модели ISO.

Стек основных протоколов сетевых архитектур ISO и TCP/IP представлен в таблице 1.1.

Можно выделить следующие существенные отличия данных архитектур:

1. Архитектура ISO предусматривает жёсткий набор протоколов на всех уровнях модели, когда на каждом уровне между взаимодействующими объектами сначала устанавливается логическая связь, а уже затем передаются

данные. При этом сверху донизу сохраняется последовательность передачи протокольных единиц (блоков, фрагментов, пакетов, кадров) и предпринимаются специальные меры для сохранения целостности этих порций данных. В случае потери или искажения протокольной единицы на каждом уровне (кроме физического) осуществляется перезапрос и повторная передача искажённой протокольной единицы.

2. Архитектура TCP/IP предусматривает возможность ветвления протоколов, и даже добавление новых. За целостностью данных следит транспортный уровень (протокол TCP), либо сам пользователь (протокол UDP).

Различия в идеологии построения сетевых архитектур порождают существенные различия механизма передачи данных на всех уровнях стандарта ISO за исключением физического и канального, где могут применяться протоколы LAP-B и X.21, но могут и другие. Основные отличия в алгоритме передачи данных состоят, во-первых, в идеологии защиты от ошибок, и, во-вторых, в реализации режима коммутации пакетов (КП).

Рассмотрим сначала методы борьбы с ошибками.

На этих уровнях очень много внимания уделено вопросам защиты данных от ошибок и сбоев. Для этого выделяется второй (канальный) уровень. Обнаружение ошибок выполняется с помощью мощного помехоустойчивого кода типа БЧХ (Реком. V. 42) с минимальным кодовым расстоянием d0=5, что позволяет обнаруживать любую 4-кратную ошибку. Исправление ошибок выполняется с помощью алгоритмов с обратной связью — РОСОЖ или (чаще) РОС-НП. Для борьбы со вставками и выпадениями кадров используется тайм-аут и циклическая нумерация кадров. На сетевом уровне обеспечивается нумерация пакетов и их перезапрос. Всё это позволяет использовать передающую среду практически любого качества, однако платой за это является высокая степень вносимой избыточности, т.е. падение реальной скорости передачи информации.

В архитектуре TCP/IP первый и второй уровни вообще не оговорены, т.е. передача может вестись даже без защиты от ошибок. Повышение верности

возложено на транспортный протокол TCP. Если используются хорошие каналы, например, волоконно-оптические линии связи (ВОЛС), то на транспортном уровне используется протокол UDP, где не предусмотрена защита от ошибок. В этом случае обнаружение и исправление ошибок осуществляется на прикладном уровне специальными программами пользователя. Такой подход становится понятным, т.к. архитектура TCP/IP первоначально была реализована в сети ARPANET, где использовались выделенные высокоскоростные каналы.

Перейдём теперь к различиям в способах коммутации пакетов, т.е. реализации 3-го уровня ISO. Здесь различия наиболее существенные.

В архитектуре ISO за маршрутизацию (доставку пакетов по адресу) отвечает третий (сетевой) уровень (Рек. X.25). Предусматривается создание виртуальных соединений или каналов от источника до получателя, а затем по этому соединению передаются пакеты. Такой режим называется виртуальным режимом КП и по принципам напоминает традиционную коммутацию каналов (КК). В архитектуре TCP/IP реализуется другой подход, называемый дейтаграммным режимом КП. Этот режим резко упрощает задачу маршрутизации, но порождает проблему сборки сообщений из пакетов, т.к. пакеты одного сообщения могут доставляться по разным маршрутам и поступать к получателю в разное время. Дейтаграммный режим КП по принципам напоминает коммутацию сообщений (КС).

Проведём сравнения виртуального и дейтаграммного методов КП по следующим характеристикам:

- установление соединения;
- адресация;
- процедура передачи пакета по сети;
- управление входным потоком сообщений; − эффективность использования сетевых ресурсов.

Установление соединения. При виртуальной КП до передачи сообщения устанавливается логическое соединение между взаимодействующими

объектами транспортного уровня (а возможно и более высоких уровней ISO). Этот логический канал запоминается в маршрутных таблицах всех центров коммутации пакетов (ЦКП), которые участвуют в соединении. Пакеты передаются только по установленному логическому каналу, поэтому порядок их следования при этом не нарушается.

При дейтаграммной КП логического соединения не устанавливается, поэтому пакеты одного сообщения передаются по тем маршрутам, которые оптимальны в данный момент, т.е. возможно разными маршрутами. Проблема сборки сообщения из пакетов решается на транспортном уровне (4 уровень по ISO).

Адресация. При виртуальном режиме КП полный адрес объектаполучателя передаётся только при установлении логического соединения, т.е. с первым пакетом. Получив этот пакет, объект-получатель извещает отправителя о согласии на проведение сеанса связи (или несогласии). Создаётся логическое соединение, и передаются остальные пакеты, содержащие только номер логического канала.

При дейтаграммном режиме КП каждый передаваемый пакет обязательно должен содержать полный адрес получателя (и отправителя) и номер пакета в сообщении.

Передача пакета по базовой сети ПД. Виртуальный режим КП предусматривает выделение специальной базовой сети передачи данных (ПД) и передачу пакетов в этой сети ПД по готовому логическому каналу, создаваемому по инициативе транспортного уровня.

При дейтаграммном режиме каждый пакет передаётся по разным маршрутам, что позволяет эффективнее использовать сетевые ресурсы, т.к. в больших сетях загрузка каналов меняется очень быстро, поэтому маршрут доставки желательно корректировать чаще. В данном случае можно построить глобальную сеть без выделения отдельной базовой сети ПД.

Управление входящим потоком. При виртуальном режиме КП управление потоком входящих сообщений (но не пакетов) возможно лишь на

входе виртуального канала, т.е. на конкретном центре коммутации пакетов для данного сообщения.

Дейтаграммный режим КП является более гибким и позволяет управлять входящим потоком сообщений практически с любого ЦКП, что улучшает гибкость управления.

Эффективность использования сетевых ресурсов. В виртуальном режиме КП оптимальный маршрут выбирается только в момент установления логического соединения, поэтому при быстром изменении ситуации на сети путь, оптимальный для первого пакета сообщения, может быть не оптимальным для последующих пакетов одного и того же сообщения.

При дейтаграммном режиме коррекция маршрута производится чаще, что позволяет более равномерно загрузить каналы всей сети и, в конечном счёте, уменьшить время доставки сообщения.

Сфера применения архитектур ISO и TCP/IP определяется их свойствами, которые порождают основные достоинства и недостатки используемых сетевых архитектур.

Так к основным достоинствам архитектуры ISO следует отнести:

– возможность реализации сетей даже наплохих каналах связи, за счёт развитой системы защиты от ошибок и сбоев;

– возможность работать в реальном масштабе времени, простота реализации режима диалога и передачи речи в цифровой форме, поскольку задержки в доставке пакетов одного и того же сообщения незначительны;

– высокая степень стандартизации протоколов на всех уровнях, что упрощает построение ИВС заданных размеров с требуемыми показателями качества обслуживания. Недостатки архитектуры ISO следующие:

- высокая избыточность за счёт большого объёма необходимой служебной информации;

- необходимость реализации большого набора достаточно сложных протоколов взаимодействия, причём отсутствие хотя бы одного протокола приводит к невозможности передачи данных;

- существенные трудности при организации взаимодействия различных сетей, особенно при различной сетевой архитектуре.

Рассмотрим теперь основные достоинства и недостатки архитектуры TCP/IP.

Достоинства архитектуры TCP/IP:

- небольшие затраты на реализацию протоколов взаимодействия за счёт меньшего набора требуемых протоколов;
- существенное упрощение процедуры маршрутизации, что снижает стоимость базовой сети передачи данных за счёт использования более простых ЦКП;
- возможность построения крупномасштабной ИВС с использованием разнотипного оборудования;
- возможность реализации взаимодействия различных сетей с применением простых алгоритмов согласования.

К недостаткам архитектуры TCP/IP можно отнести:

- возможность реализации только при использовании «хороших» каналов связи (желательно выделенных);
- необходимость решения проблемы сборки пакетов, которые могут поступать на транспортный уровень в произвольном порядке;
- возможность потери сообщения из-за несвоевременной доставки одного из пакетов этого сообщения;
- усложнение прикладных программ пользователя за счёт введения процедур контроля и исправления ошибок в получаемых сообщениях.

Теперь, опираясь на проведённый анализ, можно определить сферу применения сетевых архитектур.

Сетевая архитектура ISO эффективна при применении «плохих» каналов связи, необходимости работы в реальном масштабе времени и однородной структуре оборудования, причём основным выступает качество каналов связи. Поэтому у нас получили распространение сети с использованием архитектуры ISO (например, сеть РОСПАК).

При построении глобальных сетей, когда решающим фактором выступает простота согласования работы различных национальных сетей, реализуемых, как правило, на разнотипном оборудовании, наиболее эффективно применение архитектуры TCP/IP, данный вывод подтверждается практикой, т.к. в Internet используют именно архитектуру TCP/IP.

Лекция 9. Архитектура приложений

Архитектура приложений — это структурный принцип, по которому создано приложение. Каждому архитектурному виду свойственны свои характеристики, свойства и отношения между компонентами. Компонент — мелкая или крупная логическая и независимая часть архитектурной системы приложения. Например:

- база данных;
- процесс;
- подсистема;
- вычислительный узел;
- библиотека;
- и др.

Каждое приложение вокруг нас построено по какой-либо архитектуре. Даже в тех случаях, где принципиально не придерживались никакому из ее видов, все равно приложение будет работать в рамках одного из видов архитектуры. По-другому не получается.

Многие объединяют термины «архитектура» и «структура» в один. Но это не одно и то же, хотя структура приложения напрямую зависит от архитектуры. Когда мы произносим «архитектура приложений», то речь идет в широком смысле о компонентах будущей системы приложения. Однако каждый архитектурный компонент может быть организован и представлен разными решениями. Эти решения и образуют структуру приложения. То есть структура более точно описывает инструменты и модули будущего приложения.

Архитектура приложений

Архитектуру приложений важно продумывать перед стартом разработки. Ведь четко продуманная архитектура — это залог работоспособности и удобства будущей программы. Реакция программы на действия пользователей, возможность справляться с высокими нагрузками, зависания — все это и другие

потенциальные проблемы зависят от того, насколько качественно изначально продумали архитектуру приложения.

Каждое архитектурное решение должно удовлетворять несколько заинтересованных лиц:

- пользователя — в его интересах, чтобы приложение было безопасным, понятным и производительным;
- администратора приложения — в его интересах, чтобы у приложения была понятная система управления;
- CEO-специалиста и маркетолога — в их интересах, чтобы приложение обладало уникальными функциями и было конкурентоспособным;
- разработчика — в его интересах, чтобы ко внутреннему устройству приложения предъявляли понятные и не противоречивые требования;
- заказчика или руководителя — в их интересах, чтобы результат работы над приложением был предсказуемым, рациональным и точным.

Архитектура приложений: выбор и виды

Не бывает такого, что просто выбирают определенный вид архитектуры, и точка. Совсем наоборот. К приложению предъявляют требования, смотрят на наличие инструментов, структурных компонентов и разработчиков, учитывают различные факторы влияния на приложение, а только потом выбирают подходящую архитектуру. Получается, что выбор архитектуры основывается на:

- заинтересованных лицах: заказчиках, пользователях, разработчиках и др.;
- миссии приложения — отсюда вытекают требования к приложению;
- внутренних и внешних технических ограничениях: на наличии инструмента, профессионализме разработчиков, операционной системе и архитектуре устройств, для которых создается приложение и др.

Из-за того, что IT-мир постоянно расширяется и количество приложений постоянно растет, количество видов архитектур также постоянно растет, а

существующие архитектуры видоизменяются. Разработать собственную архитектуру или подогнать уже созданную под свои нужды становится обычным делом. Например, есть такие виды:

- архитектура «клиент-сервер»;
- монолитная архитектура;
- микропроцессорная архитектура;
- событийная;
- структурированная;
- многослойная;
- трехуровневая;
- сервис-ориентированная;
- поиск-ориентированная;
- неявная;
- и др.

Архитектура приложений: распространенные виды

Есть несколько архитектурных стратегий по разработке приложения, которые пользуются популярностью. На них мы остановимся немного подробнее.

Многослойная архитектура

Эта стратегия подразумевает разделение ответственности в приложении на несколько слоев, которые накладываются друг на друга. Каждый отдельный слой имеет собственное значение.

Многослойная архитектура разделяет программу на следующие слои:

- слой представления — отвечает за интерфейс пользователя;
- слой бизнес-логики — отвечает за функционал и логику приложения и не отвечает за интерфейс;
- слой передачи данных — отвечает за работу с базами данных и обработку информации.

Каждый элемент в приложении «проходит» через все слои. Такая архитектура считается:

- простой в реализации;
- абстрактной;
- защищенной за счет изолированности каждого слоя;
- легко управляемой и масштабируемой.

Такая архитектура свойственна небольшим приложениям, так как ее реализация возможна при монолитной структуре. Поэтому обслуживание и масштабирование больших приложений на основе этой архитектуры затруднено.

Многоуровневая архитектура приложений

Эта архитектура может быть в несколько уровней, поэтому разделяется на несколько видов:

- одноуровневая,
- двухуровневая,
- трехуровневая.

Одноуровневый вид такой архитектуры используют самые простые однопользовательские приложения, которые работают либо на стороне сервера, либо на стороне клиента.

Двухуровневый вид имеет еще одно название — архитектура «клиент-сервер». Приложения на такой архитектуре задействуют два места для обработки приложения:

- клиент — отвечает за обработку интерфейса, логики приложения и передачу информации;
- сервер — отвечает за работу хранилищ и баз данных, а также за обработку информации.

Трехуровневый вид архитектуры подразумевает наличие трех точек для работы приложения:

клиент, сервер для обработки, базы данных для хранения.

Главное отличие от двухуровневой системы — обработка и хранение информации делаются в разных местах. Такие системы свойственны высоконагруженным приложениям. Они трудны и более дорогие в реализации.

Микросервисная архитектура приложений

Микросервисная архитектура приложений — это модель, при которой разрабатываемое приложение разделяется на несколько отдельных сервисов. Такие сервисы разрабатываются отдельно друг от друга, а объединяются в одном приложении при помощи API. Каждый отдельный сервис — это отдельная изолированная функция приложения.

Такая архитектура часто встречается в разработке, так как она обладает рядом неоспоримых преимуществ:

- изолированность каждого отдельного компонента;
- сбой одного компонента не затрагивает работоспособность всего приложения;
- удобно масштабировать и внедрять обновления.

Альтернатива данной модели — монолитная архитектура, у которой вся функциональность приложения реализована в одном месте. Эти две системы имеют собственные достоинства и недостатки. Монолитные приложения разрабатываются быстрее, считаются более безопасными, но их трудно обновлять и масштабировать. Поэтому монолит рекомендуется применять в небольших приложениях, которые нетрудно будет обслуживать. Микросервисная архитектура приложений рекомендуется для масштабных проектов с богатым функционалом, над которым постоянно нужно будет работать и масштабировать.

Лекция 10. Интеграция. REST и SOAP

Понятие интеграции многогранно. В него входят такие задачи, как использование несколькими системами общих справочников (например, списка клиентов, каталога товаров), действия одного сервиса в ответ на событие в другом, организация одних и тех же бизнес-процессов в двух и более приложениях. Автоматический обмен данными с клиентами и партнерами, обеспечение единого стандарта взаимодействия между филиалами — это тоже примеры корпоративной интеграции программного обеспечения.

Сложилась тенденция предоставлять своим сотрудникам, партнерам и клиентам *доступ* ко всем типам информации и сервисов посредством сети Веб. Однако в корпоративных сетях компаний функционирует огромное число разнородных бизнес-приложений, созданных в различное время, различными организациями, на базе различных технологий. Задача веб-интеграции заключается в том, чтобы объединить разнородные веб-приложения и системы в единую среду на базе сети Веб.

Практикуются следующие подходы к веб-интеграции:

- Интеграция на *уровне представления. Данный* уровень позволяет пользователю взаимодействовать с приложением. Интеграция на уровне представления дает *доступ к пользовательскому интерфейсу удаленных приложений*.
- Интеграция на уровне функциональности. Данная интеграция подразумевает обеспечение прямого доступа к бизнес-логике приложений. Это достигается непосредственным взаимодействием приложений с *API* (программному интерфейсу приложений) или же взаимодействием посредством *веб-сервисов*.
- Интеграция на уровне данных. В данном случае предполагается доступ к одной или нескольким *базам данных*, используемых удаленным приложением.

- Комплексная интеграция. Коммерческие решения по веб-интеграции, как правило, включают все три типа интеграции

Использование веб-интеграции выгодно *по* многим причинам:

- *Веб-интеграция* позволяет развертывать информационные системы на базе сторонних приложений без необходимости разбираться в их родительских системах, программных средах и архитектурах баз данных.
- *SOA* и *веб-сервисы* используют программный язык и платформонезависимые интерфейсы между приложениями корпоративной инфраструктуры ИТ. Это дает очевидные преимущества в поддержке, управляемости, развертывании информационных сетей.
- Веб-интеграция позволяет конструировать комплексную функциональность, комбинируя разнородные компоненты посредством протоколов веб-сервисов.
- Веб-интеграция позволяет использовать веб-сервисы разработчиков.
- Веб-интеграция позволяет развивать программные интерфейсы приложений через протоколы веб-сервисов без программирования.

Для веб-интеграции обычно используется коммерческое *ПО* или популярные технологии, такие как *PHP/Python/Perl, XForms, SOAP* и т.д.

Интеграция на основе XML

Большое количество систем, стандартов и технологий приводит к тому, что эффективно связать разные источники данных в одну систему не получается. Даже такие, на первый взгляд однородные источники, как системы управления базами данных, применяют языки запросов и форматы представления выбираемой информации, которые редко полностью совместимы между собой. Как следствие, проекты интеграции в таких условиях требуют больших усилий - требуется вникать в детали различных баз данных, протоколов, операционных систем и так далее. В результате *интеграция* нескольких приложений или систем реализуется *по* схеме, представленной ниже:

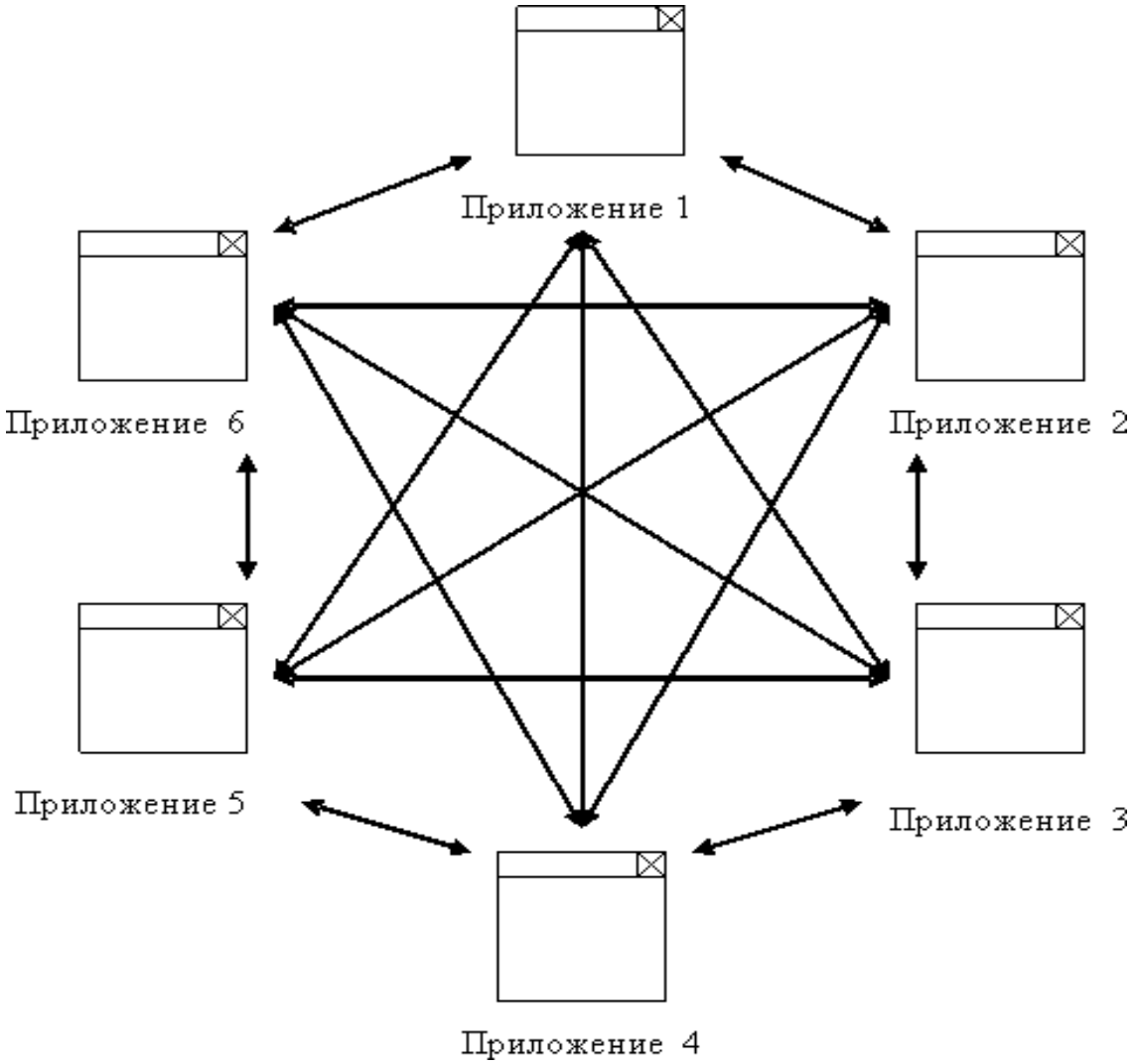


Рис. 12. Интеграция нескольких приложений

Заставить разные системы работать вместе - чрезвычайно трудоемкая задача. Идея использования *XML* в интеграции информационных систем сводится к созданию общего *XML*-языка, которым могла бы пользоваться каждая из них.

Такое решение сразу же намного упрощает проект. Вместо реализации взаимодействия между каждой парой систем следует всего лишь научить каждую из них "говорить" на *XML* языке. Иначе говоря, все сводится к разработке нескольких *врапперов* (*wrapper* - упаковщик, программное средство создания системной оболочки для стандартизации внешних обращений и изменения функциональной ориентации действующей системы), которые будут

переводить со стандартного *XML*-языка интегрированной системы на язык, понятный каждой системе в отдельности:

- средства разработки и стандартные библиотеки для XML существуют практически на всех платформах и для большинства популярных языков программирования;
- методы работы с XML достаточно стандартны для того, чтобы в разных системах можно было пользоваться одинаковыми приемами;
- информация, оформленная в виде XML, может обрабатываться не только машинами, но и человеком (что намного облегчает отладку).

В принципе, *интеграция по XML*-схеме не отличается коренным образом от интеграции на основе любого другого общего стандарта. Вместе с тем она имеет *целый* ряд весомых преимуществ:

- XML языки не зависят от аппаратных и программных платформ, что позволяет связывать разнородные системы;
- выразительная мощность XML достаточно велика для того, чтобы описать данные практически любой сложности;

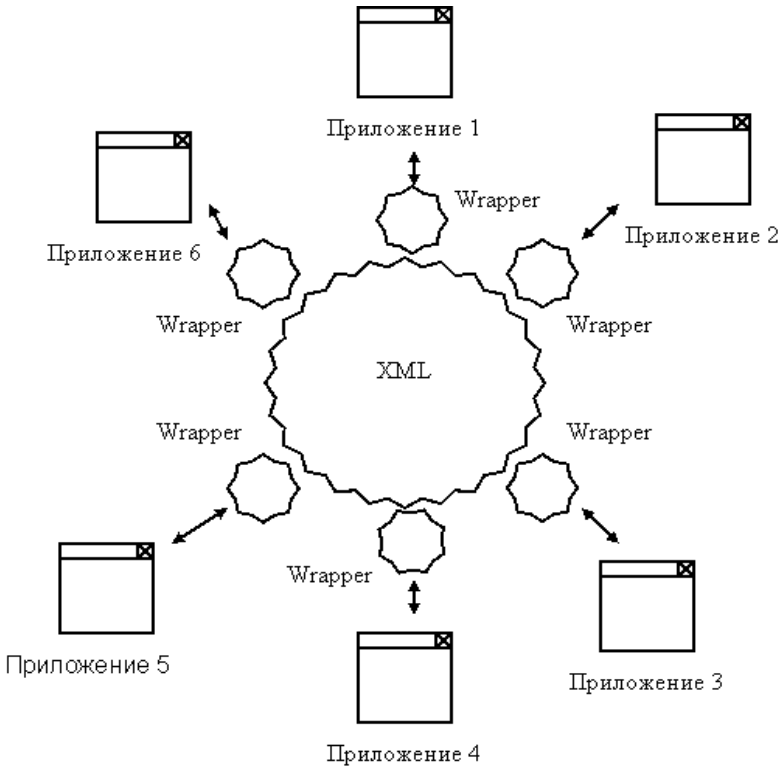


Рис. 13. Интеграция по XML

Интеграция на основе *XML* практически реализуется в рамках протоколов:

- *XML-RPC*. Это протокол удаленного вызова процедур с передачей данных в формате XML через TCP-порт 80, т.е. HTTP -порт.
- *WDDX* (Web Distributed Exchange). Представляет собой механизм обмена сложными структурами данных по протоколу HTTP. Протокол базируется не на структурах, а на событиях.
- *ebXML* (electronic buisiness XML) – XML для электронного бизнеса. Основное назначение – предоставление открытой XML-инфраструктуры, обеспечивающей безопасное глобальное использование информации электронного бизнеса.
- *Веб-сервисы (веб-службы)*.

Веб-сервисы

Веб-сервис (*web service*) — программная система, имеющая *идентификатор URI*, и общедоступные интерфейсы которой определены на языке *XML*. Описание этой программной системы может быть найдено другими приложениями, которые могут взаимодействовать с ней в соответствии с этим описанием посредством сообщений, основанных на *XML*, и передаваемых с помощью интернет-протоколов. Веб-служба является единицей модульности при использовании *сервис-ориентированной архитектуры* приложения.

Сервис-ориентированная архитектура (*SOA, service-oriented architecture*) — модульный подход к разработке программного обеспечения, основанный на использовании сервисов со стандартизированными интерфейсами.

В основе *SOA* лежат принципы многократного использования функциональных элементов ИТ, унификации типовых операционных процессов. Компоненты программы могут быть распределены *по* разным узлам сети, и предлагаются как независимые и слабо связанные, заменяемые сервисы-приложения.

Интерфейс компонентов SOA-программы осуществляет инкапсуляцию деталей реализации конкретного компонента (ОС, языка программирования и т. п).

SOA хорошо зарекомендовала себя при построении крупных корпоративных программных систем. *Целый* ряд разработчиков и интеграторов предлагают инструменты и решения на основе *SOA* (например, платформы Microsoft .*NET* , *IBM* WebSphere, *SAP* NetWeaver, Diasoft и др.).

Веб-сервисы .*NET* имеют следующие достоинства:

- *Открытость стандартов*. В веб-сервисах отсутствуют какие-либо скрытые или недоступные элементы. Каждый аспект технологии, от способа поиска веб-сервисы до ее описания и организации связи с ней, определен общедоступными стандартами.
- *Межплатформенность*. Язык программирования, который позволяет создавать XML-документы и отправлять информацию посредством HTTP, позволяет взаимодействовать с любым веб-сервисом. Можно получать веб-услугу из системы, отличной от .NET.
- *Простота*.
- *Поддержка сообщений на понятном человеку языке*. Переход от двоичных стандартов, применяемых в COM и CORBA, к XML-тексту позволил упростить исправление ошибок и обеспечил возможность осуществлять взаимодействие с веб-сервисами по обычным каналам HTTP.

Реализация веб-сервисов .*NET* осуществляется так же просто, как и активизация удаленной веб-сервисы или *вызов метода* локального класса. Это достигается за счет применения инструментов, предоставляемых системой .*NET* Framework, которые позволяют создать полноценный *веб-сервис*, без необходимости изучения деталей работы таких стандартов, как *SOAP, WSDL* и *UDDI*. При этом выполняются следующие действия:

1. Веб-сервис разрабатывается как .NET-класс с атрибутами, которые идентифицируют его как веб-сервис с некоторыми функциями.

2. В среде .NET автоматически создается документ WSDL, где описывается, как клиент должен взаимодействовать с веб-сервисом.

3. Потребитель находит созданный веб-сервис и может добавить соответствующую веб-ссылку в проект Visual Studio .NET.

4. В среде .NET осуществляется автоматическая *проверка документа* WSDL и генерируется прокси-класс, который позволяет потребителю взаимодействовать с веб-сервисом.

5. Потребитель вызывает один из методов вашего класса веб-сервиса. С его точки зрения этот вызов внешне ничем не отличается от вызова метода любого другого класса, хотя взаимодействие происходит на самом деле с прокси-классом, а не с веб-сервисом.

6. Прокси-класс преобразует, переданные параметры в сообщение SOAP и отправляет его веб-сервису.

7. Затем прокси-класс получает SOAP-ответ, преобразует его в соответствующий тип данных и возвращает его как обычный тип данных .NET.

8. Потребитель использует полученные данные.

При работе веб-сервисов *.NET* используется технология *ASP .NET*, являющаяся частью системы *.NET* Framework. Она также требует поддержки со стороны сервера Microsoft *IIS.*

Работа веб-сервисов построена на использовании нескольких открытых стандартов:

- *XML* - расширяемый язык разметки, предназначенный для хранения и передачи структурированных данных;
- *SOAP* - протокол обмена сообщениями на базе XML;
- *WSDL* - язык описания внешних интерфейсов веб-сервисов на базе XML;
- *UDDI* - универсальный интерфейс распознавания, описания и интеграции (Universal Discovery, Description, and Integration). Каталог веб-сервисов и сведений о компаниях, предоставляющих веб-сервисы во всеобщее пользование или конкретным компаниям.

Спецификация WSDL

Каждый *веб-сервис* предоставляет документ *WSDL* (*Web Service Description Language* - язык описания веб-сервиса), в котором описывается все, что клиенту необходимо для работы с этим сервисом. *WSDL*-документ предоставляет простой и последовательный способ задания разработчиком синтаксиса вызова любого веб-метода. Более того, этот документ позволяет использовать инструменты автоматического генерирования прокси-классов, подобные включенным в среды *Visual Studio .NET* и *.NET* Framework. Благодаря указанным средствам использование веб-сервиса является таким же простым, как и применение локального класса.

WSDL-документ имеет основанный на *XML* формат, в соответствии с которым *информация* подразделяется на пять групп. Первые три группы представляют собой абстрактные определения, не зависящие от особенностей платформы, сети или языка, а оставшиеся две группы включают конкретные описания.

Протокол SOAP

Связь между веб-сервисами и их клиентами осуществляется посредством сообщений в формате *XML*.

SOAP (Simple *Object* Access *Protocol* - простой протокол доступа к объектам) представляет собой протокол сообщений для выбора веб-сервисов.

Основная идея стандарта *SOAP* заключается в том, что сообщения должны быть закодированы в стандартизированном *XML*-формате.

Кроме сообщений *SOAP*, для обмена данными с сервисами *.NET* можно использовать методы *GET* и *POST* протокола *HTTP*.

Преимущества применения формата *SOAP* перед другими форматами для передачи данных:

- Кодировать в XML структуры данных и наборы DataSet с использованием SOAP так же легко, как и данные простых скалярных типов.

- При использовании SOAP-сообщений предоставляются дополнительные инструменты, позволяющие легко добавлять, например, функции обеспечения безопасности или трассировки.
- Имеются наборы инструментов SOAP для различных языков программирования (и даже для предыдущих версий Microsoft C++ и Visual Basic). Иначе, для того чтобы обеспечить связь с сервисом посредством методов GET и POST протокола HTTP, придется, очевидно, самостоятельно конструировать строку запроса, а затем проводить синтаксический анализ ответа.

API — это набор определений и протоколов. API нужны для разработки и интеграции приложений, поскольку облегчают обмен данными между двумя частями программного обеспечения, например, поставщиком информации (сервером) и пользователем.

API определяют содержимое, доступное клиенту, выполняющему вызов от производителя, возвращающего ответ. Программы используют API для взаимодействия, получения информации или выполнения функций.

API выступает в роли посредника между пользователями (клиентами) и ресурсами (серверами).

Когда пользователи делают запросы к API или посещают интернет-магазин, они ожидают быстрого ответа. Поэтому необходимо оптимизировать Magento TTFB (Time To First Byte) или использовать другие стратегии повышения производительности.

Причины для интеграции API:

- оптимизация обмена ресурсами и информацией;
- контроль доступа с помощью аутентификации и определения прав;
- безопасность;
- отсутствие необходимости разбираться в специфике ПО;
- согласованное взаимодействие между сервисами, даже если сервисы используют разные технологии.

PI — это набор определений и протоколов. API нужны для разработки и интеграции приложений, поскольку облегчают обмен данными между двумя частями программного обеспечения, например, поставщиком информации (сервером) и пользователем.

API определяют содержимое, доступное клиенту, выполняющему вызов от производителя, возвращающего ответ. Программы используют API для взаимодействия, получения информации или выполнения функций. API выступает в роли посредника между пользователями (клиентами) и ресурсами (серверами).

Когда пользователи делают запросы к API или посещают интернет-магазин, они ожидают быстрого ответа. Поэтому необходимо оптимизировать Magento TTFB (Time To First Byte) или использовать другие стратегии повышения производительности, которые лучше работают для выбранной CMS.

Причины для интеграции API:

- оптимизация обмена ресурсами и информацией;
- контроль доступа с помощью аутентификации и определения прав;
- безопасность;
- отсутствие необходимости разбираться в специфике ПО;
- согласованное взаимодействие между сервисами, даже если сервисы используют разные технологии.

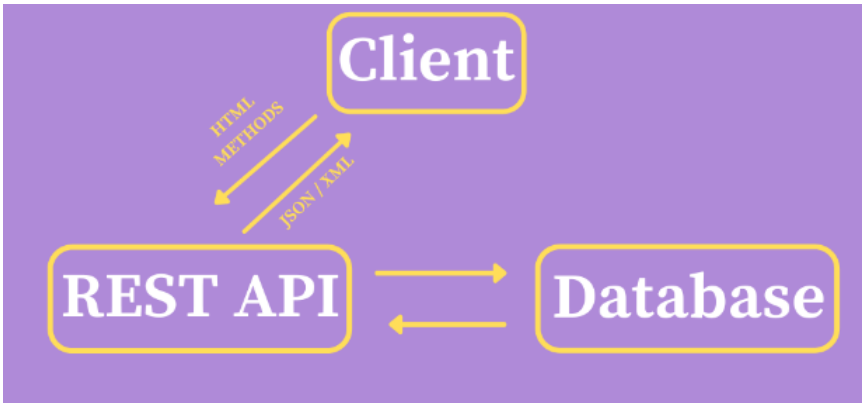


Рис. 14. Концепция REST API

Данные (такие как изображения, видео и текст) воплощают ресурсы в REST. Клиент посещает URL-адрес и отправляет серверу запрос, чтобы получить ответ.

Запрос (URL, к которому обращаетесь) содержит четыре компонента:

1. Конечная точка, являющаяся URL-адресом со структурой root-endpoint/?
2. Метод с типом (GET, POST, PUT, PATCH, DELETE).
3. Заголовки, выполняющие функции аутентификации, предоставление информации о содержимом тела (допустимо использовать параметр -H или --header для отправки заголовков HTTP) и т. д.
4. Данные (или тело) – это то, что отправляется на сервер с помощью опции -d или --data при запросах POST, PUT, PATCH или DELETE.

HTTP-запросы разрешают работать с базой данных, например:

1. POST-запрос для создания записей.
2. GET-запрос на чтение или получение ресурса (документа или изображения, набора других ресурсов) с сервера.
3. PUT и PATCH-запросы для обновления записей.
4. DELETE-запрос на удаление ресурса с сервера.

Эти операции обозначают четыре возможных действия CRUD: создание, чтение, обновление и удаление.

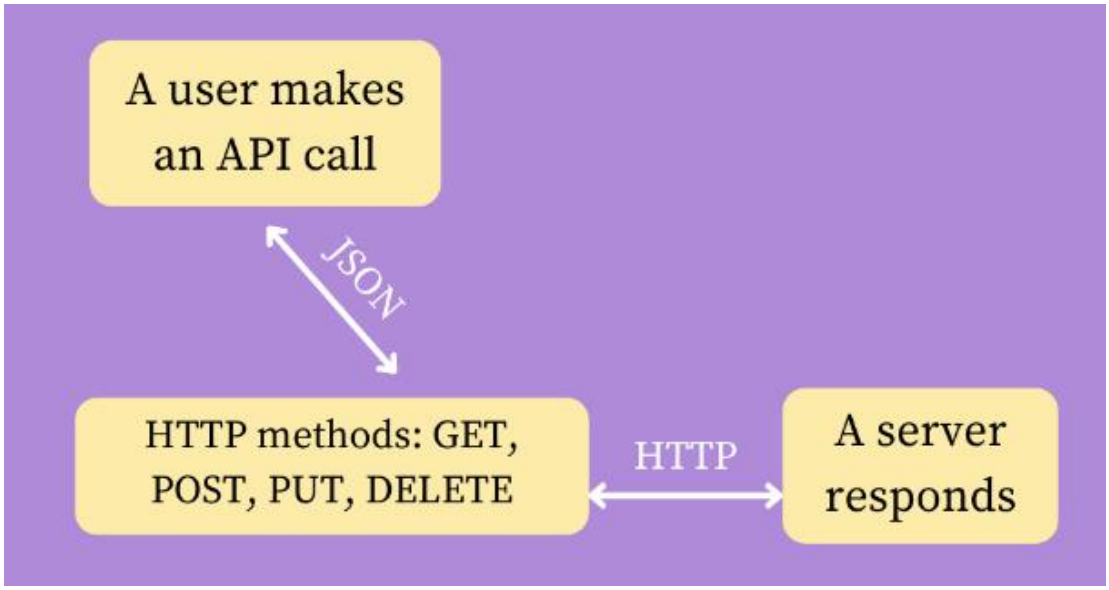


Рис. 15. Особенности стиля архитектуры REST ful

Разработчикам приходится учитывать жёсткую структуру некоторых API, таких как SOAP или XML-RPC. Но REST API — это другое. REST API поддерживают широкий спектр типов данных и могут быть написаны практически на любом языке программирования.

Шесть архитектурных ограничений REST являются принципами разработки решения и заключаются в следующем:

Унифицированный интерфейс (последовательный пользовательский интерфейс)

Эта концепция диктует, что запросы API к одному и тому же ресурсу, независимо от происхождения, должны быть идентичными, то есть на одном конкретном языке. Один универсальный идентификатор ресурса (URI) ассоциируется с одними и теми же данными, такими как имя пользователя или адрес электронной почты.

Другой принцип унифицированного интерфейса гласит, что сообщения должны быть информативными. Сообщения должны быть понятны серверу, чтобы определить, как с ними обращаться (например, тип запроса, MIME-типы и т. д.).

Разделение клиента и сервера

Архитектурный стиль REST использует особый подход к реализации клиента и сервера. Дело в том, что клиент и сервер могут работать независимо и не обязаны знать друг о друге.

Например, у клиента только универсальный идентификатор запрошенного ресурса (URI) и не может общаться с серверной программой другим способом. Однако, сервер не должен влиять на клиентское ПО. Поэтому сервер отправляет данные по HTTP. Это означает что, если клиентский код изменится, это не повлияет на работу сервера.

Клиентские и серверные программы могут быть модульными и независимыми до тех пор, пока каждая сторона знает, какой формат сообщения доставлять другой. Отделение пользовательского интерфейса от ограничений

хранения данных улучшает гибкость интерфейса на разных платформах и повышает масштабируемость.

Кроме того, каждый компонент выигрывает от разделения, поскольку может развиваться независимо. Интерфейс REST помогает клиентам:

Иметь доступ к одним и тем же конечным точкам REST.

Выполнять одинаковые действия.

Получать одинаковые ответы.

Нестационарная связь между клиентами и серверами

Системы на основе REST не имеют состояния, то есть состояние клиента остаётся неизвестным для сервера и наоборот. Это ограничение разрешает серверу и клиенту понимать отправленное сообщение, даже если они не видели предыдущие.

Чтобы обеспечить соблюдение этого ограничения без статичности, требуется использовать ресурсы, а не команды. Это имена существительные в сети. Их цель – описать объект, который требуется сохранить или передать другим службам.

Допустимо контролировать, изменять и повторно использовать компоненты, затрагивая систему частично, поэтому преимущества этого ограничения включают достижение:

- стабильности;
- скорости;
- масштабируемости RESTful-приложений.

Обратите внимание, что каждый запрос должен содержать всю информацию, необходимую для выполнения. Клиентские приложения должны сохранять состояние сессии, поскольку серверные приложения не должны хранить данные, связанные с клиентским запросом.

Кэшируемые данные

REST требует кэширования ресурсов на стороне клиента или сервера везде, где это возможно. Кэширование данных и ответов имеет решающее

значение, поскольку обеспечивает высокую производительность на стороне клиента.

Хорошо управляемое кэширование может уменьшить или устранить некоторые взаимодействия клиент-сервер.

Это также даёт серверу больше возможностей масштабирования благодаря меньшей нагрузке на сервер. Кэширование увеличивает скорость загрузки страниц и разрешает получить доступ к ранее просмотренному контенту без подключения к интернету.

Архитектура многоуровневой системы

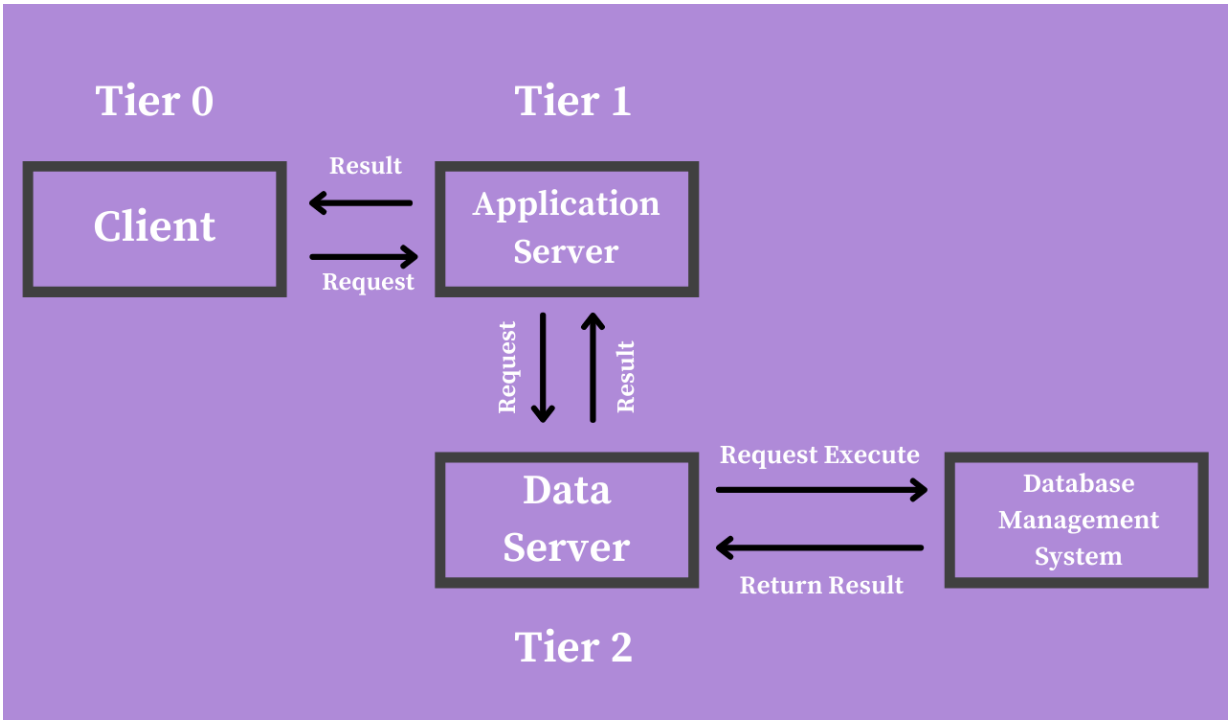


Рис. 16. Архитектура многоуровневой системы

Следующее обсуждаемое ограничение — это многоуровневая структура RESTful. Этот принцип включает в себя группировку слоёв с определёнными функциями.

Слои REST API имеют свои обязанности и располагаются в иерархическом порядке. Например, один слой может отвечать за хранение данных на сервере, второй — за развёртывание API на другом сервере, а третий — за аутентификацию запросов на другом сервере.

Эти слои действуют как посредники и предотвращают прямое взаимодействие между клиентскими и серверными приложениями. В результате клиент не знает, к какому серверу или компоненту обращается.

Когда каждый слой выполняет заданную функцию до передачи данных следующим, что повышает общую безопасность и гибкость API, так как добавление, изменение, удаление API не влияет на другие компоненты интерфейса.

Лекция 11. Интеграция. Часть 2

Обеспечение интеграции разных информационных систем между собой можно через сервисную шину ESB. Это тип связующего ПО, которое дает возможность службам, созданным в различных средах, легко и быстро взаимодействовать. Обмен данными происходит через шину с использованием различных протоколов и форматов, позволяя избежать доработок интегрируемых систем. ESB, таким образом, — это промежуточное ПО, которое обеспечивает преобразование сообщений в нужный формат, контроль транзакций, маршрутизацию с учетом смысла, равномерное распределение нагрузки на сервисы и безопасность обмена данными.

Для наглядности рассмотрим на примере, как это работает. Предположим, пользователь входит в личный кабинет на сайте страховой компании. Он видит свое имя, даты окончания страховки, новые предложения от компании. Все эти данные собраны из разных систем и предоставляются через различные интерфейсы. Более того, каждое приложение может быть создано на различных технологических стеках разными командами разработчиков.

Рис. 17. Взаимодействие приложений между собой

Как все эти сервисы могут напрямую взаимодействовать друг с другом? Ведь для того, чтобы получить данные из другого приложения, придется пройти сложную многоуровневую цепочку операций. Хорошо, если таких сервисов

пять–шесть, а если их десятки или даже сотни? Непрерывный обмен сообщениями между ними грозит превратиться в настоящий хаос. На стороне пользователя это будет проявляться длительным ожиданием и постоянными сбоями в работе приложений. А если хотя бы одну из систем потребуется обновить, изменить или распределить между отделами, это неизбежно затронет все прочие сервисы.

ESB шина полностью меняет дело. С ней приложениям больше не нужно непрямую обращаться друг к другу — каждое из них взаимодействует только с интеграционной платформой. Это сразу устраняет необходимость в огромном количестве методов доступа — интерфейсов потребуется ровно столько, сколько существует сервисов.

Если в одну из систем потребуется внести изменения, это никак не повлияет на работу других корпоративных приложений. За все эти задачи будет отвечать только шина данных ESB [2].

Таким образом, ESB-подход, в отличие от традиционной архитектуры «точка-точка» (когда сервисы напрямую взаимодействуют друг с другом), обладает большей гибкостью. Сценарии интеграции можно модифицировать с минимальным вмешательством разработчиков [3].

Преимущества решения очевидны. Упрощение интеграции приложений путем внедрения ESB дает экономию времени и денег, улучшает функционирование сервисов, что в конечном итоге работает на повышение эффективности организации и на увеличение прибыли предприятия.

Как устроена интеграционная шина? Решение включает в себя несколько компонентов:

Брокер сообщений — обеспечивает управление очередностью сообщений и выступает посредником между приложением-источником и приложением-приемником;

Комплект адаптеров — программных компонентов, которые служат для связи приложений с ESB и преобразуют один интерфейс в другой. Чем больше

различных адаптеров заложено в интеграционную шину, тем шире ее функционал;

В современных ESB-решениях реализованы принципы микросервисной архитектуры. В соответствии с ними весь функционал системы распределяется между микросервисами, каждый из которых может работать независимо от других;

Средства для контроля и мониторинга.

Интеграция программных модулей

Мы выяснили, зачем компаниям нужна корпоративная сервисная шина. Теперь пора разобраться с ее возможностями. Посмотрим, какие процессы реализует интеграционная шина данных.

Маршрутизация сообщений

В этом заключается основная функция ESB: она получает данные из одних приложений и направляет их в другие в соответствии с заданными правилами, выстраивает пути движения потоков информации, их последовательность. Сервисная шина данных содержит инструменты настройки, с помощью которых можно задавать нужные параметры управления информационными потоками.

Преобразование сообщений

Данные из разных систем могут быть представлены в различных форматах — XML, CSV, JSON, DBF и других. При классическом подходе «точка-точка» это затрудняет процесс «общения» приложений между собой. Сервисная шина предприятия решает проблему, преобразуя данные из неподходящего формата в подходящий. Например, если нужно отправить одно и то же сообщение и в ERP, и в CRM, ESB трансформирует данные нужным образом и передаст в соответствующие системы.

Масштабируемость

Благодаря этому свойству ESB может работать с различным количеством информационных систем и различными объемами данных, распределяя нагрузку между приложениями. Интеграционная шина обеспечивает передачу

данных любого объема, разбивая крупные массивы на более мелкие. Обработка информации по частям в случае сбоя предотвращает потерю данных и необходимость заново передавать уже отправленные пакеты. Масштабируемость также обеспечивает возможность неограниченного наращивания информационных мощностей предприятия, причем IT-ландшафт не обязательно должен быть однородным.

Традиционная SOA-архитектура с ESB в качестве центрального компонента еще несколько лет назад была на пике популярности. Но совершенству нет предела, и классический подход получает новое развитие. Следующим этапом эволюции технологий интеграции с ESB стала микросервисная архитектура. Она позволила решить типичные проблемы, обнаружившиеся в процессе «обрастания» шины бизнес-логикой: тяжеловесность, многослойность с тесной взаимосвязью слоев, сложность внесения изменений и другие недостатки, свойственные монолитности.

Система очередей сообщений обеспечивает *асинхронный метод взаимодействия для программ*. Метод не требует установления прямой связи между интегрируемыми программами и позволяет поддерживать обмен данными в виде *сообщений*, независимо от аппаратной или операционной системы. При этом гарантируется, что *сообщение* не будет потеряно или получено дважды.

Система очередей сообщений предоставляет прикладным программам сервис для отправки и получения *сообщений*. *Очереди* представляют промежуточное *место* хранения *сообщений*, как бы специализированную базу данных со всеми механизмами, гарантирующими сохранность: *журналирование*, *восстановление после сбоев*, *транзакционная обработка*. При этом приложения обращаются к *очередям* при помощи прикладного программного интерфейса. *Сообщение*, предназначенное для другой программы, должно быть доставлено в *очередь* назначения. *Сообщение* может передаваться через распределенную систему серверов - *менеджеров очередей*. Каждый раз,

получая *сообщение, менеджер очередей* записывает *сообщение* в локальную *очередь*, затем передает *сообщения по* сети другому *менеджеру очередей*. При этом гарантируется, что *сообщение* не будет потеряно или получено дважды. *Программа*-адресат обращается к целевой *очереди* и получает *доступ* к *сообщению*.

Поддержка асинхронного взаимодействия приложений при использовании *очередей сообщений* позволяет гибко интегрировать разнородные прикладные системы, не меняя внутренней логики их работы. Существование промежуточного звена в виде *очередей сообщений* позволяет вставлять промежуточную обработку передаваемой информации для решения самых различных задач: обеспечение безопасности, трансформации данных в разных форматах, динамической маршрутизации, анализа содержания передаваемых данных и так далее. Промежуточное звено позволяет применять различные топологии вычислительной среды, соединяя *потоки данных* для централизованной унифицированной обработки и разделяя их для функционально разнородной обработки.

Средства *MOM* имеют своих предшественников в виде систем для передачи файлов типа *FTP*, решений типа экспорта-импорта данных с использование различных промежуточных хранителей: файлов, буферов памяти, баз данных. Находясь долгое время в виде вспомогательных и частных средств, *класс* подобного программного обеспечения стал бурно развиваться и стандартизироваться в связи с выходом на первый план задач *интеграции* прикладных систем между собой.

Системы очередей сообщений позволяют программам отправлять и получать данные, не соединяясь друг с другом напрямую. Приложения изолируются друг от друга транспортным слоем, состоящим из *менеджеров очередей*, берущих на себя задачи обеспечения коммуникаций. С помощью *очередей сообщений* (рис. 18) могут быть реализованы как традиционные модели взаимодействия программ (*клиент-сервер*), так и модели, которые реализуются только при помощи сервиса *сообщений*.

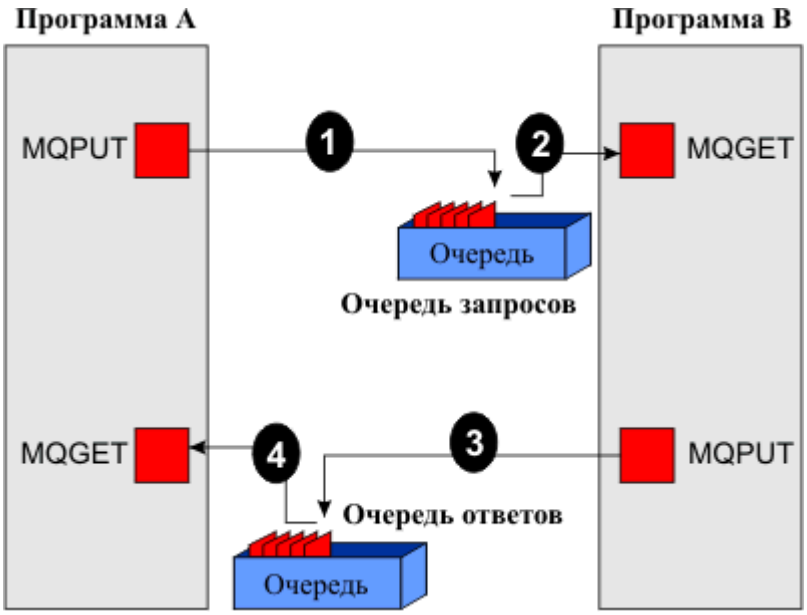


Рис. 18. Взаимодействие клиент-сервер через очередь сообщений

Модель клиент/сервер

Запросы клиента передаются в виде сообщений в серверную очередь. После обработки запроса приложение-сервер отправляет ответ в виде нового сообщения в очередь, указанную клиентом в сообщении -запросе.

Асинхронное взаимодействие

При использовании очередей сообщений нет необходимости, чтобы взаимодействующие приложения были активны одновременно. Программа может отправить сообщение другому приложению, которое обработает его, когда сочтет нужным. Отправив сообщение, программа может не ожидать ответа, а продолжать выполнение других задач.

Запуск программы

Сообщение, посланное одной прикладной программой, может инициировать старт другого приложения. В таком случае по команде программы-монитора приложение-адресат запускается и обрабатывает сообщение.

Параллельная и распределенная обработка

Исполнение прикладного процесса может быть распределено между несколькими прикладными программами. Старт, координация между

программами и консолидация результатов обработки на разных системах реализуются путем пересылки сообщений через очереди.

Адресация по принципу публикация-подписка

Прикладные программы-публикаторы посылают свою информацию с указанием темы в виде сообщения менеджеру очередей. Другие приложения - подписчики присылают заявки на информацию по темам. Присланные публикации распределяются между подписчиками через очереди сообщений специальной программой - брокером в соответствии с темами публикаций.

Основные компоненты и особенности работы системы очередей сообщений

Основными элементами системы очередей сообщений являются:

- сообщения, которые прикладные программы посылают друг другу;
- очереди, где хранятся сообщения;
- менеджеры очередей – серверные компоненты, которые управляют очередями и обработкой сообщений;
- каналы передачи сообщений, которые связывают менеджеры очередей между собой.

Введем следующие определения.

Сообщения (Message) представляют собой структуру данных, состоящую из заголовка сообщения (MQMessageDescriptor) и прикладных данных (рис.1.3). Заголовок содержит контрольную и адресную информацию о сообщении и его характеристиках. С помощью этой информации менеджер очередей решает, каким образом обрабатывать и куда надо передавать сообщение.

В создании и изменении заголовочной информации принимают участие, как приложения - отправители, так и системные компоненты менеджера очередей. Содержание контрольной информации важно для безопасности системы передачи сообщений, и поэтому для изменения и перекопирования контрольной информации в новые сообщения определяются специальные права.

Прикладная часть сообщения может включать данные в специальных предопределенных форматах или данные в форматах пользователя. WebSphereMQ поддерживает произвольные форматы данных, однако бывают системы очередей сообщений, которые разрешают только свои специальные форматы, например XML специального вида. Поскольку в распределенной среде для приложений, функционирующих под управлением различных ОС, существуют различные кодовые страницы и методы представления данных, система очередей должна поддерживать методы конвертации передаваемых данных.

Очередь сообщений (Queue) - основное место хранения и обработки сообщений. Физическое управление очередями полностью скрыто от прикладных программ. Приложения имеют доступ к очередям через программный интерфейс MQI (Message Queue Interface). Для передачи критически важной информации используются "постоянные" (persistence) сообщения, которые журналируются и восстанавливаются после рестарта менеджера сообщений. Для повышения производительности системы очередей сообщений поддерживает также и "непостоянные" сообщения, которые содержатся в оперативной памяти, не журналируются и могут быть потеряны в результате системного или аппаратного сбоя.

Менеджер очередей (Queue Manager) является главным серверным компонентом и отвечает за управление очередями сообщений и прием вызовов от прикладных программ. Внутренняя реализация менеджеров очередей для каждой операционной системы своя. Однако с функциональной точки зрения менеджеры очередей поддерживают единый программный интерфейс, единую систему адресации и обмена данных через каналы, и единую систему администрирования.

Прикладной программный интерфейс

Прикладные программы взаимодействуют с WebSphereMQ через программный интерфейс MQI (Message Queue Interface). MQI имеет единую структуру на всех платформах и основан на простой системе из десятка команд:

- команда подключения к менеджеру очередей MQCONN
- команда открытия очереди MQOPEN
- команда помещения сообщения в очередь MQPUT
- команда выборки сообщений из очереди MQGET
- вспомогательные команды запроса и установки атрибутов очередей MQINQ и MQSET
- команды успешного завершения транзакции MQCMIT
- команда отката транзакции назад MQBACK
- команда закрытия очереди MQCLOSE
- команда отсоединения приложения от менеджера очередей MQDISC.

При создании приложений обеспечивается поддержка интерфейса MQI для языков программирования: C/C++, Java, SmallTalk, Cobol, PL/1, Lotus LSX, Basic. Надо отметить, что разработка приложений для систем очередей сообщений имеет свои особенности, приведенные в последующих лекциях.

Распределенная передача сообщений: Как сообщение попадает в очередь, если очередь находится на другой системе?

Пользовательские приложения не обязаны знать внутреннюю структуру системы очередей сообщений, где физически размещены очереди, какие коммуникации существуют между менеджерами очередей. Приложение, обращаясь к менеджеру очередей, всегда получает доступ только к локальным очередям сообщений. Когда приложение посылает сообщение в очередь, расположенную на удаленной системе, то сообщение записывается в специальную транспортную очередь (transmission queue), что обеспечивает надежное сохранение сообщения, а уже затем сообщение переправляется по каналу передачи сообщений другому менеджеру на удаленную систему.

На рис.12.4 показаны основные элементы, участвующие в передаче сообщения - от приложения к менеджеру очередей A и затем в удаленную очередь на менеджере очередей B.

Каналы передачи сообщений: Как сообщения пересылаются по сети?

В распределенной среде за передачу *сообщений* отвечают сетевые компоненты *системы очередей сообщений*. В IBM *WebSphere MQ* используется система *каналов* передачи *сообщений*, обеспечивающая *гарантированную передачу сообщений* поверх разнообразных сетевых соединений. При использовании *протокола гарантированной передачи WebSphere MQ MCP* (*Message Channel Protocol*) посылаемое *сообщение* не будет удалено из *очереди* на сервере посылки до тех пор, пока это *сообщение* не будет полностью принято на сервере - адресате, то есть реализуется сетевая транзакция при передаче *сообщений*.

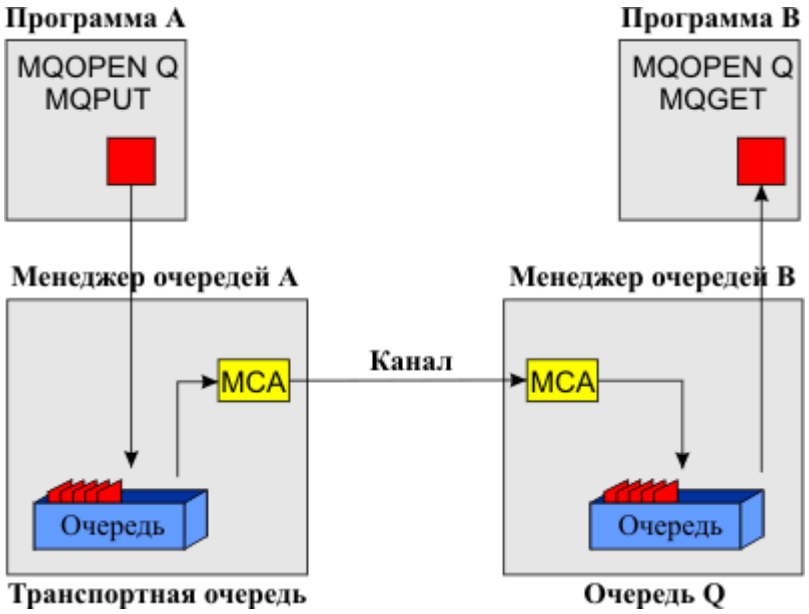


Рис. 19. Передача сообщений между системами

Каналы соединяют *менеджеры очередей* и позволяют осуществлять направленную посылку *сообщений*. В *WebSphere MQ каналы* являются однонаправленными и состоят из пары взаимодействующих канальных агентов (Message Channel Agent - *MCA*). Для двусторонней связи необходимо определить два *канала*. Существует несколько типов *каналов*. Типы *каналов* различаются тем, какая сторона в *канале* является инициатором установки связи, а какая источником *сообщений*. В комбинации *каналов* типа Sender-Receiver одна сторона Sender является инициатором связи и источником *сообщений*, вторая сторона Receiver только откликается на

инициирующий запрос и принимает поток *сообщений*, в другой комбинации *канал* Requestor-Server, инициирующая соединение сторона Requestor принимает *сообщения* от стороны Server.

После установки связи из транспортной *очереди* в *канале* начинается передача *сообщений*. *Сообщения* удаляются из транспортной *очереди* передающего *менеджера*, только после подтверждения доставки *сообщения* другим *менеджером*. Использование в протоколе *MCP* контрольных точек позволяет концам *канала* синхронизировать передачу числа *сообщения* в случае системного или сетевого сбоя. Если линия связи недоступна, *MQSeries* может автоматически совершать повторные попытки передачи после восстановления связи. Протокол *MCP* используется при передаче *сообщений* поверх транспортных протоколов более низкого уровня TCP/IP, LU6.2, *DECnet*, NetBios, IPX/SPX.

Адресация и маршрутизация сообщений: Как сообщение находит очередь назначения в распределенной среде?

Сервер *системы очередей сообщений* отправляет *сообщения* различным адресатам, пользуясь информацией из заголовка каждого *сообщения*: имя *менеджера очередей* для идентификации узла и имя *очереди* для идентификации самой *очереди*.

В стандартной двойной структуре имен, лежащей в основе системы маршрутизации *MQSeries - WebSphere MQ*, предусмотрены дополнительные правила, расширяющие возможности идентификации *очередей*. Кроме прямого использования полных имен *очередей* реализован алгоритм разрешения имен, позволяющий указывать адресатов при помощи псевдонимов *очередей* и определений удаленных *очередей*. Это дает возможность привязывать имена *очередей*, заложенные разработчиками приложений в процессе кодирования программы к реальной *системе очередей*. В частности, при изменении физической конфигурации системы администратор сети при помощи административных команд может переопределить

маршрутизацию *сообщения* к новому местоположению *очереди* без изменений кода приложения.

Механизм разрешения имен *системы очередей сообщений* используется для организации многошаговой маршрутизации *сообщений* через произвольное число промежуточных *менеджеров очередей*.

Для обработки *сообщения* важным является адрес *очереди* ответа, то есть хранящаяся в заголовке *сообщения* информация о том, в какую *очередь* должен быть отправлен ответ на это *сообщение*. Имя *очереди* ответа используется для организации связи типа запрос-ответ, а также для организации отправки многочисленных *сообщений* -отчетов, информирующих о системных событиях, например, о доставке *сообщения* в *очередь* назначения или о выборке этого *сообщения* приложением - адресатом.

Для отслеживания и исправления ошибок у *менеджера очередей* имеется специальная *очередь* DLQ (Dead-Letter Queue) для хранения недоставленных *сообщений*. Чаще всего *сообщения* попадают в DLQ, когда *очередь*, указанная в заголовке *сообщения*, не существует или когда *очередь* назначения оказывается полной. *Очереди* недоставленных *сообщений* позволяют разгрузить транспортные *очереди* от ошибочных *сообщений* и освободить *каналы* от непрерывных повторных обработок. При попадании *сообщения* в *очередь* недоставленных *сообщений* в его тело вставляется специальный информационный подзаголовок, позволяющий анализировать причины попадания в DLQ. *MQSeries* обладает механизмом задания правил для автоматической обработки недоставленных *сообщений*.

Транзакционные свойства: Как обеспечивается целостность данных и синхронизация изменения в сообщениях?

Корпоративная *система очередей сообщений* должна обязательно включать в себя механизмы транзакций. Прикладная программа помечает часть своих получаемых и отправляемых *сообщений* специальной опцией - участвующие в транзакции. До выполнения приложением команды на

завершение транзакции, посланные этим приложением *сообщения* являются фактически "невидимыми" для других приложений, а полученные приложением *сообщения* реально не удаляются из *очередей*. В случае выполнения приложением команды на откат транзакции, *сообщения* в *очереди* восстанавливаются к состоянию на начало транзакции.

WebSphereMQ обладает своим внутренним *менеджером ресурсов*, который, кроме того, поддерживает внешний XA интерфейс, и может участвовать в распределенной транзакции под управлением таких мониторов транзакций как *CICS*, Encina, Tuxedo. Сами по себе сервера *WebSphereMQ*, начиная с *версии* 5, могут быть координаторами распределенных транзакций с двухфазной фиксацией.

Триггерные возможности: Как активизировать программу обработки?

Асинхронный характер работы *системы очередей сообщений* требует специального механизма внешней активизации для старта прикладных и системных компонентов. В *WebSphereMQ* такой механизм - "триггеринг" привязан непосредственно к *очередям сообщений*. В качестве триггерных событий могут выступать, например, появление в *очереди* нового *сообщения* или энного *сообщения* с приоритетом выше определенного порогового значения.

Чтобы определить триггерное событие, для прикладной *очереди* при помощи административных команд устанавливаются опции, разрешающие триггеринг и условия *триггерного события*. Кроме того, администратором системы создается специальное описание обработки *триггерного события*. В этом описании содержится информация о приложении, которое будет вызвано при наступлении *триггерного события*. В случае возникновения такого события, например появления нового *сообщения* в *очереди*, *менеджер очередей* автоматически генерирует специальное триггерное *сообщение*, которое помещается в специальную *очередь* инициации (initiation queue). В триггерном *сообщении* содержатся данные о событии и вызываемом процессе.

Все *очереди* инициации отслеживаются триггерным монитором, который читает триггерное *сообщение* и вызывает внешнюю программу обработки (рис. 20).

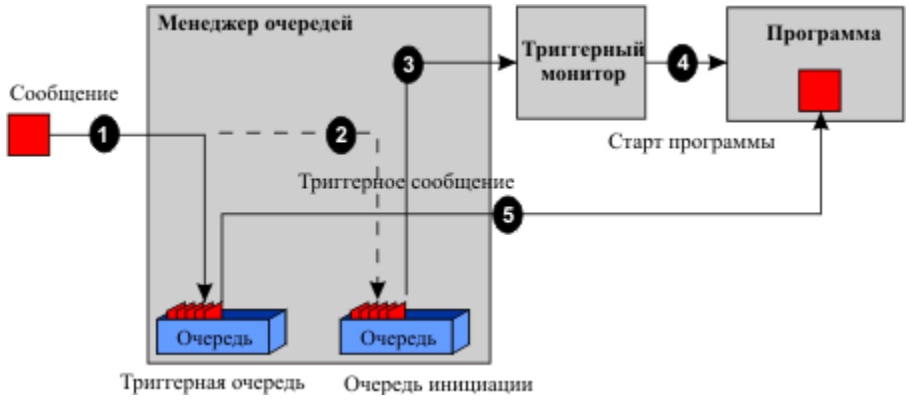


Рис. 20. Обработка событий при помощи триггеринга

Триггеринг достаточно часто применяется в качестве стандартного метода автоматического старта *каналов* между *менеджерами очередей*. Для этого достаточно в качестве триггерных *очередей* объявить транспортные *очереди*, содержащие *сообщения* для передачи удаленному *менеджеру очередей*.

Применение системы очередей сообщений

Программное обеспечение *систем очередей сообщений* является одной из наиболее перспективных технологий для *интеграции* систем. Среди первостепенных достоинств *систем очередей сообщений* как решения для *интеграции*, можно указать простоту, надежность и универсальность данного подхода.

Достоинства архитектуры *очередей сообщений* позволяют использовать ее, как для задач *интеграции* готовых приложений, так и разработки совершенно новых систем, базирующихся на принципах обработки событий и передачи *сообщений* и в полной мере использующих все преимущества новых подходов. Можно указать на следующие типичные задачи *интеграции* систем:

- *Интеграция* приложений в распределенной гетерогенной среде;
- Организация взаимодействия независимо работающих приложений или приложений, работа которых разделена во времени;

- Сложные распределенные и/или распараллеленные процессы обработки;
- Пересылка файлов, передача больших объемов данных;
- Синхронизация и репликация данных;
- Поддержка мобильных клиентов.

Достаточно частыми являются случаи кроссплатформенной *интеграции*, например, серверное приложение, работающее на рабочей станции под управлением операционной системы UNIX, должно посылать данные об обновлениях базы данных и проведенных транзакциях приложению на сервер системы iSeries или мейнфрейм.

Между разными приложениями, работающими на одном мощном сервере, может быть организовано межсистемное взаимодействие, например, между подсистемами OS/390, когда другие технологии межсистемного взаимодействия оказываются более сложными.

Многие сервисные организации, в первую очередь в области финансовых услуг, используют *очередь сообщений* как базовый транспорт, с помощью которого они предоставляют заказчикам свои услуги. Здесь важнейшими факторами являются *гарантированная доставка сообщений*, многочисленность аппаратных платформ, а также открытость решения, позволяющая подключать к *системе очередей сообщений* собственные прикладные компоненты, а также корпоративные и отраслевые средства обеспечения безопасности.

Расширяемость системы. Важной характеристикой *системы очередей сообщений* является возможность расширять функциональность системы. Существование открытых интерфейсов для встраиваемых модулей (канальные и интерфейсные user exits) и документированных интерфейсов для базовых сервисных компонент позволяет подключать к *менеджеру очередей* дополнительные модули, готовые или написанные пользователем, и использовать их для взаимного опознания систем, кодирования *сообщений*, компрессии данных и других задач.

Лекция 12. Метрики

Система измерения включает метрики и модели измерений, которые используются для количественной оценки качества ПО.

При определении требований к ПО задаются соответствующие им внешние характеристики и их атрибуты (подхарактеристики), определяющие разные стороны управления продуктом в заданной среде. Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

В соответствии со стандартом ISO/IEC 9126, Information Technology – Software quality characteristics and metrics (Part 1–4) метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.

Типы метрик.

Существует три типа метрик:

- метрики программного продукта, которые используются при измерении его характеристик - свойств;
- метрики процесса, которые используются при измерении свойства процесса ЖЦ создания продукта.
- метрики использования.

Метрики программного продукта включают:

- внешние метрики, обозначающие свойства продукта, видимые пользователю;
- внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта — это метрики:

- надежности продукта, которые служат для определения числа дефектов;
- функциональности, с помощью которых устанавливаются наличие и правильность реализации функций в продукте;
- сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда); применимости продукта, которые способствуют определению степени доступности для изучения и использования;
- стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают:

- метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
- метрики сложности, необходимые для определения сложности продукта;
- метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования и управления достижением качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств, значений модели качества или прогнозирования. Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта. Общая мера - степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев типа UML и оценкой количества:

- требований;
- сценариев и действующих лиц;

- объектов, включенных в сценарий, и локализация требований к каждому сценарию;
- параметров и операций объекта и др.

Стандарт ISO/IEC 9126-2 определяет следующие типы мер:

- мера размера ПО в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);
- мера времени (функционирования системы, выполнения компонента и др.);
- мера усилий (производительность труда, трудоемкость и др.);
- мера учета (количество ошибок, число отказов, ответов системы и др.).

Специальной мерой может служить уровень использования повторных компонентов и измеряется как отношение размера продукта, изготовленного из готовых компонентов, к размеру системы в целом. Данная мера используется также при определении стоимости и качества ПО. Примеры метрик:

- общее число объектов и число повторно используемых;
- общее число операций, повторно используемых и новых операций;
- число классов, наследующих специфические операции;
- число классов, от которых зависит данный класс;
- число пользователей класса или операций и др.

При оценке общего количества некоторых величин часто используются среднестатистические метрики (среднее число операций в классе, наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются метрики Холстеда — это характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений наиболее часто встречающихся операндов и операторов;

длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракции средств языка и ориентация на проблему) и др.

В качестве метрик процесса могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:

- общее время разработки и отдельно время для каждой стадии;
- время модификации моделей;
- время выполнения работ на процессе;
- число найденных ошибок при инспектировании;
- стоимость проверки качества;
- стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить не свойства самой программы, а результаты ее эксплуатации - эксплуатационное качество. Примером может служить - точность и полнота реализации задач пользователя, а такжезатраченные ресурсы (трудозатраты, производительность и др.) на эффективное решение задач пользователя. Оценка требований пользователя проводится с помощью внешних метрик.

Лекция 13. Тестирование

Тестирование программного обеспечения (Software Testing) — проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определённым образом.

Цель тестирования — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы.

Для чего проводится тестирование ПО:

1. Для проверки соответствия требованиям.
2. Для обнаружения проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта.
3. Обнаружение вариантов использования, которые не были предусмотрены при разработке. А также взгляд на продукт со стороны пользователя.
4. Повышение лояльности к компании и продукту, т.к. любой обнаруженный дефект негативно влияет на доверие пользователей.

Принципы тестирования

- Принцип 1 — Тестирование демонстрирует наличие дефектов (Testing shows presence of defects).
Тестирование только снижает вероятность наличия дефектов, которые находятся в программном обеспечении, но не гарантирует их отсутствия.
- Принцип 2 — Исчерпывающее тестирование невозможно (Exhaustive testing is impossible).
Полное тестирование с использованием всех входных комбинаций данных, результатов и предусловий физически невыполнимо (исключение — тривиальные случаи).

- Принцип 3 — Раннее тестирование (Early testing). Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше.
- Принцип 4 — Скопление дефектов (Defects clustering). Большая часть дефектов находится в ограниченном количестве модулей.
- Принцип 5 — Парадокс пестицида (Pesticide paradox). Если повторять те же тестовые сценарии снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты.
- Принцип 6 — Тестирование зависит от контекста (Testing is context depending). Тестирование проводится по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем новостной портал.
- Принцип 7 — Заблуждение об отсутствии ошибок (Absence-of-errors fallacy). Отсутствие найденных дефектов при тестировании не всегда означает готовность продукта к релизу. Система должна быть удобна пользователю в использовании и удовлетворять его ожиданиям и потребностям.

Обеспечение качества (QA — Quality Assurance) и контроль качества (QC — Quality Control) — эти термины похожи на взаимозаменяемые, но разница между обеспечением качества и контролем качества все-таки есть, хоть на практике процессы и имеют некоторую схожесть.

QC (Quality Control) — Контроль качества продукта — анализ результатов тестирования и качества новых версий выпускаемого продукта.

К задачам контроля качества относятся:

- проверка готовности ПО к релизу;
- проверка соответствия требований и качества данного проекта.

QA (Quality Assurance) — Обеспечение качества продукта — изучение возможностей по изменению и улучшению процесса разработки, улучшению коммуникаций в команде, где тестирование является только одним из аспектов обеспечения качества. К задачам обеспечения качества относятся:

- проверка технических характеристик и требований к ПО;
- оценка рисков;
- планирование задач для улучшения качества продукции;
- подготовка документации, тестового окружения и данных;
- тестирование;
- анализ результатов тестирования, а также составление отчетов и других документов.

Верификация и валидация — два понятия тесно связаны с процессами тестирования и обеспечения качества.

Верификация (verification) — это процесс оценки системы, чтобы понять, удовлетворяют ли результаты текущего этапа разработки условиям, которые были сформулированы в его начале.

Валидация (validation) — это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе.

Документацию, которая используется на проектах по разработке ПО, можно условно разделить на две группы:

– Проектная документация — включает в себя всё, что относится к проекту в целом.

– Продуктовая документация — часть проектной документации, выделяемая отдельно, которая относится непосредственно к разрабатываемому приложению или системе.

Этапы тестирования:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии тестирования и планирование процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование

7. Стабилизация

8. Эксплуатация

Стадии разработки ПО — этапы, которые проходят команды разработчиков ПО, прежде чем программа станет доступной для широкого круга пользователей.

Программный продукт проходит следующие стадии:

1. анализ требований к проекту;
2. проектирование;
3. реализация;
4. тестирование продукта;
5. внедрение и поддержка.

Требования — это спецификация (описание) того, что должно быть реализовано. Требования описывают то, что необходимо реализовать, без детализации технической стороны решения. Атрибуты требований:

1. Корректность — точное описание разрабатываемого функционала.
2. Проверяемость — формулировка требований таким образом, чтобы можно было выставить однозначный вердикт, выполнено все в соответствии с требованиями или нет.
3. Полнота — в требовании должна содержаться вся необходимая для реализации функциональности информация.
4. Недвусмысленность — требование должно содержать однозначные формулировки.
5. Непротиворечивость — требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.
6. Приоритетность — у каждого требования должен быть приоритет (количественная оценка степени значимости требования). Этот атрибут позволит грамотно управлять ресурсами на проекте.
7. Атомарность — требование нельзя разбить на отдельные части без потери деталей.

8. Модифицируемость — в каждое требование можно внести изменение.

9. Прослеживаемость — каждое требование должно иметь уникальный идентификатор, по которому на него можно сослаться.

Дефект (bug) — отклонение фактического результата от ожидаемого. Отчёт о дефекте (bug report) — документ, который содержит отчет о любом недостатке в компоненте или системе, который потенциально может привести компонент или систему к невозможности выполнить требуемую функцию. Атрибуты отчета о дефекте:

1. Уникальный идентификатор (ID) — присваивается автоматически системой при создании баг-репорта.

2. Тема (краткое описание, Summary) — кратко сформулированный смысл дефекта, отвечающий на вопросы: Что? Где? Когда (при каких условиях)?

3. Подробное описание (Description) — более широкое описание дефекта (указывается опционально).

4. Шаги для воспроизведения (Steps To Reproduce) — описание четкой последовательности действий, которая привела к выявлению дефекта. В шагах воспроизведения должен быть описан каждый шаг, вплоть до конкретных вводимых значений, если они играют роль в воспроизведении дефекта.

5. Фактический результат (Actual result) — описывается поведение системы на момент обнаружения дефекта в ней. чаще всего, содержит краткое описание некорректного поведения (может совпадать с темой отчета о дефекте).

6. Ожидаемый результат (Expected result) — описание того, как именно должна работать система в соответствии с документацией.

7. Вложения (Attachments) — скриншоты, видео или лог-файлы.

8. Серьёзность дефекта (важность, Severity) — характеризует влияние дефекта на работоспособность приложения.

9. Приоритет дефекта (срочность, Priority) — указывает на очерёдность выполнения задачи или устранения дефекта.

10. Статус (Status) — определяет текущее состояние дефекта. Статусы дефектов могут быть разными в разных баг-трекинговых системах.

11. Окружение (Environment) – окружение, на котором воспроизвелся баг.

Серьёзность бага (severity) показывает степень ущерба, который наносится проекту существованием дефекта. Severity выставляется тестировщиком.

Градация Серьезности дефекта (Severity):

- Блокирующий (S1 – Blocker) тестирование значительной части функциональности вообще недоступно. Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
- Критический (S2 – Critical) критическая ошибка, неправильно работающая ключевая бизнес-логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, то есть не работает важная часть одной какой-либо функции либо не работает значительная часть, но имеется workaround (обходной путь/другие входные точки), позволяющий продолжить тестирование.
- Значительный (S3 – Major) не работает важная часть одной какой-либо функции/бизнес-логики, но при выполнении специфических условий, либо есть workaround, позволяющий продолжить ее тестирование либо не работает не очень значительная часть какой-либо функции. Также относится к дефектам с высокими visibility – обычно не сильно влияющие на функциональность дефекты дизайна, которые, однако, сразу бросаются в глаза.
- Незначительный (S4 – Minor) часто ошибки GUI, которые не влияют на функциональность, но портят юзабилити или внешний вид. Также

незначительные функциональные дефекты, либо которые воспроизводятся на определенном устройстве.

- Тривиальный (S5 – Trivial) почти всегда дефекты на GUI — опечатки в тексте, несоответствие шрифта и оттенка и т.п., либо плохо воспроизводимая ошибка, не касающаяся бизнес-логики, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Срочность (priority) показывает, как быстро дефект должен быть устранён. Priority выставляется менеджером, тимлидом или заказчиком.

Градация Приоритета дефекта (Priority):

- P1 Высокий (High) Критическая для проекта ошибка. Должна быть исправлена как можно быстрее.
- P2 Средний (Medium) Не критичная для проекта ошибка, однако требует обязательного решения.
- P3 Низкий (Low) Наличие данной ошибки не является критичным и не требует срочного решения. Может быть исправлена, когда у команды появится время на ее устранение.

Существует шесть базовых типов задач:

- Эпик (epic) — большая задача, на решение которой команде нужно несколько спринтов.
- Требование (requirement) — задача, содержащая в себе описание реализации той или иной фичи.
- История (story) — часть большой задачи (эпика), которую команда может решить за 1 спринт.
- Задача (task) — техническая задача, которую делает один из членов команды.
- Подзадача (sub-task) — часть истории / задачи, которая описывает минимальный объем работы члена команды.
- Баг (bug) — задача, которая описывает ошибку в системе.

Тестовые среды

- Среда разработки (Development Env) – за данную среду отвечают разработчики, в ней они пишут код, проводят отладку, исправляют ошибки
- Среда тестирования (Test Env) – среда, в которой работают тестировщики (проверяют функционал, проводят smoke и регрессионные тесты, воспроизводят.
- Интеграционная среда (Integration Env) – среда, в которой проводят тестирование взаимодействующих друг с другом модулей, систем, продуктов.
- Предпрод (Preprod Env) – среда, которая максимально приближена к продакшену. Здесь проводится заключительное тестирование функционала.
- Продакшн среда (Production Env) – среда, в которой работают пользователи.

Основные фазы тестирования

- Pre-Alpha: прототип, в котором всё ещё присутствует много ошибок и наверняка неполный функционал. Необходим для ознакомления с будущими возможностями программ.
- Alpha: является ранней версией программного продукта, тестирование которой проводится внутри фирмы-разработчика.
- Beta: практически готовый продукт, который разработан в первую очередь для тестирования конечными пользователями.
- Release Candidate (RC): возможные ошибки в каждой из фичей уже устранены, и разработчики выпускают версию, на которой проводится регрессионное тестирование.
- Release: финальная версия программы, которая готова к использованию.

Основные виды тестирования ПО

Вид тестирования — это совокупность активностей, направленных на тестирование заданных характеристик системы или её части, основанная на конкретных целях.

1. Классификация по запуску кода на исполнение:

- Статическое тестирование — процесс тестирования, который проводится для верификации практически любого артефакта разработки: программного кода компонент, требований, системных спецификаций, функциональных спецификаций, документов проектирования и архитектуры программных систем и их компонентов.
- Динамическое тестирование — тестирование проводится на работающей системе, не может быть осуществлено без запуска программного кода приложения.

2. Классификация по доступу к коду и архитектуре:

- Тестирование белого ящика — метод тестирования ПО, который предполагает полный доступ к коду проекта.
- Тестирование серого ящика — метод тестирования ПО, который предполагает частичный доступ к коду проекта (комбинация White Box и Black Box методов).
- Тестирование чёрного ящика — метод тестирования ПО, который не предполагает доступа (полного или частичного) к системе. Основывается на работе исключительно с внешним интерфейсом тестируемой системы.

3. Классификация по уровню детализации приложения:

- Модульное тестирование — проводится для тестирования какого-либо одного логически выделенного и изолированного элемента (модуля) системы в коде. Проводится самими разработчиками, так как предполагает полный доступ к коду.
- Интеграционное тестирование — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.
- Системное тестирование — процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности.

- Приёмочное тестирование — проверяет соответствие системы потребностям, требованиям и бизнес-процессам пользователя.

4. Классификация по степени автоматизации:

- Ручное тестирование.
- Автоматизированное тестирование.

5. Классификация по принципам работы с приложением

- Позитивное тестирование — тестирование, при котором используются только корректные данные.
- Негативное тестирование — тестирование приложения, при котором используются некорректные данные и выполняются некорректные операции.

6. Классификация по уровню функционального тестирования:

- Дымовое тестирование (smoke test) — тестирование, выполняемое на новой сборке, с целью подтверждения того, что программное обеспечение стартует и выполняет основные для бизнеса функции.
- Тестирование критического пути (critical path) — направлено для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности.
- Расширенное тестирование (extended) — направлено на исследование всей заявленной в требованиях функциональности.

7. Классификация в зависимости от исполнителей:

- Альфа-тестирование — является ранней версией программного продукта. Может выполняться внутри организации-разработчика с возможным частичным привлечением конечных пользователей.
- Бета-тестирование — программное обеспечение, выпускаемое для ограниченного количества пользователей. Главная цель — получить отзывы клиентов о продукте и внести соответствующие изменения.

8. Классификация в зависимости от целей тестирования:

- Функциональное тестирование (functional testing) — направлено на проверку корректности работы функциональности приложения.

- Нефункциональное тестирование (non-functional testing) — тестирование атрибутов компонента или системы, не относящихся к функциональности.

1. Тестирование производительности (performance testing) — определение стабильности и потребления ресурсов в условиях различных сценариев использования и нагрузок.
2. Нагрузочное тестирование (load testing) — определение или сбор показателей производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).
3. Тестирование масштабируемости (scalability testing) — тестирование, которое измеряет производительность сети или системы, когда количество пользовательских запросов увеличивается или уменьшается.
4. Объёмное тестирование (volume testing) — это тип тестирования программного обеспечения, которое проводится для тестирования программного приложения с определенным объемом данных.
5. Стрессовое тестирование (stress testing) — тип тестирования направленный для проверки, как система обращается с нарастающей нагрузкой (количеством одновременных пользователей).
6. Инсталляционное тестирование (installation testing) — тестирование, направленное на проверку успешной установки и настройки, обновления или удаления приложения.
7. Тестирование интерфейса (GUI/UI testing) — проверка требований к пользовательскому интерфейсу.
8. Тестирование удобства использования (usability testing) — это метод тестирования, направленный на установление степени удобства использования, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

9. Тестирование локализации (localization testing) — проверка адаптации программного обеспечения для определенной аудитории в соответствии с ее культурными особенностями.

10. Тестирование безопасности (security testing) — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

11. Тестирование надёжности (reliability testing) — один из видов нефункционального тестирования ПО, целью которого является проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

12. Регрессионное тестирование (regression testing) — тестирование уже проверенной ранее функциональности после внесения изменений в код приложения, для уверенности в том, что эти изменения не внесли ошибки в областях, которые не подверглись изменениям.

13. Повторное/подтверждающее тестирование (re-testing/confirmation testing) — тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.

Тест-дизайн — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы).

Техники тест-дизайна

Автор книги "A Practitioner’s Guide to Software Test Design", Lee Copeland, выделяет следующие техники тест-дизайна:

1. Тестирование на основе классов эквивалентности (equivalence partitioning) — это техника, основанная на методе чёрного ящика, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений.

2. Техника анализа граничных значений (boundary value testing) — это техника проверки поведения продукта на крайних (граничных) значениях входных данных.

3. Попарное тестирование (pairwise testing) — это техника формирования наборов тестовых данных из полного набора входных данных в системе, которая позволяет существенно сократить количество тест-кейсов.

4. Тестирование на основе состояний и переходов (State-Transition Testing) — применяется для фиксирования требований и описания дизайна приложения.

5. Таблицы принятия решений (Decision Table Testing) — техника тестирования, основанная на методе чёрного ящика, которая применяется для систем со сложной логикой.

6. Доменный анализ (Domain Analysis Testing) — это техника основана на разбиении диапазона возможных значений переменной на поддиапазоны, с последующим выбором одного или нескольких значений из каждого домена для тестирования.

7. Сценарий использования (Use Case Testing) — Use Case описывает сценарий взаимодействия двух и более участников (как правило — пользователя и системы).

Методы тестирования:

1. Тестирование белого ящика — метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику.Согласно ISTQB, тестирование белого ящика — это:

- тестирование, основанное на анализе внутренней структуры компонента или системы;
- тест-дизайн, основанный на технике белого ящика — процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.
- Почему «белый ящик»? Тестируемая программа для тестировщика — прозрачный ящик, содержимое которого он прекрасно видит.

2. Тестирование серого ящика — метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично.

3. Тестирование чёрного ящика — также известное как тестирование, основанное на спецификации или тестирование поведения — техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.

Согласно ISTQB, тестирование черного ящика — это:

- тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы;
- тест-дизайн, основанный на технике черного ящика — процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

Тестовая документация:

Тест план (Test Plan) — это документ, который описывает весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков. Тест план должен отвечать на следующие вопросы:

- Что необходимо протестировать?
- Как будет проводиться тестирование?
- Когда будет проводиться тестирование?
- Критерии начала тестирования.
- Критерии окончания тестирования.

Основные пункты тест плана:

1. Идентификатор тест плана (Test plan identifier);
2. Введение (Introduction);
3. Объект тестирования (Test items);
4. Функции, которые будут протестированы (Features to be tested;)

5. Функции, которые не будут протестированы (Features not to be tested);

6. Тестовые подходы (Approach);

7. Критерии прохождения тестирования (Item pass/fail criteria);

8. Критерии приостановления и возобновления тестирования (Suspension criteria and resumption requirements);

9. Результаты тестирования (Test deliverables);

10. Задачи тестирования (Testing tasks);

11. Ресурсы системы (Environmental needs);

12. Обязанности (Responsibilities);

13. Роли и ответственность (Staffing and training needs);

14. Расписание (Schedule);

15. Оценка рисков (Risks and contingencies);

16. Согласования (Approvals).

Чек-лист (check list) — это документ, который описывает что должно быть протестировано. Чек-лист может быть абсолютно разного уровня детализации.

Чаще всего чек-лист содержит только действия, без ожидаемого результата. Чек-лист менее формализован.

Тестовый сценарий (test case) — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Атрибуты тест кейса:

- Предусловия (PreConditions) — список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
- Шаги (Steps) — список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод об удовлетворении реализации, поставленным требованиям.

- Ожидаемый результат (Expected result) — что по факту должны получить.

Рекомендуемые источники для изучения дисциплины

1. Вигерс Карл, Битти Джой. Разработка требований к программному обеспечению. 3-е изд., дополненное / Пер. с англ. — М. : Издательство «Русская редакция» ; СПб. : БХВ-Петербург, 2014. — 736 стр. : ил. ISBN 978-5-7502-0433-5 («Русская редакция») ISBN 978-5-9775-3348-5 («БХВ-Петербург»)
2. Бобби Вульф, Грегор Хоп. Шаблоны интеграций корпоративных приложений. Пер. с англ. — М. : ООО «И.Д.Вильямс»-2019. – 672 с. : ил. – Парал. тит.англ. ISBN 978-5-8459-1146-9 (рус.)
3. Алистер Коберн. Современные методы описания функциональных требований к системам. Издательство: Лори; 2012. – 264 с.
4. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия.

Процессы жизненного цикла программных средств; 2012. – 160с.
5. Федеральный профессиональный стандарт РФ "Системный аналитик"; 2014. – 44с.
6. Кен Швабер и Джефф Сазерленд. Руководство по Скраму.2017. – 26 с.
7. Новиков Ф.А., Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. – СПб.: Профессиональная литература, Наука и Техника, 2010". - 640с.
8. Ньюмен Сэм. От монолита к микросервисам: Пер. с англ.- БХВ-Петербург, 2021 – 272 с.: ил. ISBN 978-5-9775-6723-7