**[https://docs.google.com/document/d/1QSK3ij--c5tx2khNpHjvXg58BBohwj7te\_W8y0E5yH0/edit#](https://docs.google.com/document/d/1QSK3ij--c5tx2khNpHjvXg58BBohwj7te_W8y0E5yH0/edit)**

**file:///C:/Users/%D0%9B%D0%B5%D0%BE%D0%BD%D0%B8%D0%B4/Downloads/Telegram%20Desktop/%D0%9E%D0%BF%D0%98%20%D0%BF%D0%BE%D0%B4%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%BA%D0%B0%20(1).pdf**

**1. ISO/IEC 12207:2010: Жизненный цикл ПО. Группы процессов ЖЦ.**

Жизненный цикл ПО – время от идеи до вывода из эксплуатации.

* Разработка требований
* Проектирование
* Тестирование
* Эксплуатация
* Анализ
* Разработка
* Внедрение
* Вывод из эксплуатации

Процесс ЖЦ – входные данные и ресурсы -> совокупность действий -> Выходные данные и ресурсы

Группы процессов ЖЦ

* Согласование
* Орг. Обеспечение
* Проектирование
* Процессы программных средств
* Реализации программных средств
* Поддержка программных средств
* Повторные испытания программных средств

**2. Модели ЖЦ (последовательная, инкрементная, эволюционная).**

Модель ЖЦ ПО – структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.

Последовательная (однократная) – определены все требования, разработка проходит за один этап

Инкрементная – определены все требования, но есть возможность разбить работу на несколько этапов. Процедура разработки по инкрементной модели предполагает выпуск на первом большом этапе продукта в базовой функциональности, а затем уже последовательное добавление новых функций, так называемых «инкрементов».

Эволюционная – требования определены не до конца, и доопределяются со временем.

**3. Водопадная (каскадная) модель.**

Водопадная модель **-** подразумевает последовательное прохождение стадий, каждая из которых должна завершиться полностью до начала следующей

* Определяются системные требования
* Определяются требования к ПО
* Требования анализируется
* Проектируется программа
* Пишется код
* Тестирование
* Ввод в эксплуатацию

В своей критике на данную модель Ройс написал, что у нас есть две возможности откатится назад. Из тестирования на проектирование и из проектирования на определение требований.

**4. Методология Ройса.**

Основывается на 5 нововведениях

* Предварительный дизайн – сделать обзор будущей системы.
* Документирование – документирование всех важных этапов проектирования ПО
* Do it twice - сделать небольшой прототип для проверки продукта
* Тестирование
* Подключение пользователя – на любой стадии

**5. Традиционная V-chart model J.Munson, B.Boehm.**

В основе лежит каскадная модель, но дня каждого этапа предусмотрено тестирование. Так в начале мы спускаемся по каскадной модели до момента кодирования, а после начинаем подниматься и проверять все предшествующие стадии на корректность.

**6. Многопроходная модель (Incremental model).**

По сути является V образной моделью с инкрементальным подходом. Т.е этап разработки проходит в несколько этапов.

**7. Модель прототипирования (80-е).**

Для того чтобы не отставать от трендов данная методика предполагает создание тестовых моделей итогового ПО. Разработчики делают базовые модели с базовым функционалом, заказчик говорит нравится ему данный вариант или нет. Если нет то создается новая модель, если да, то начинаем дорабатывать данную версию. Таким образом, обеспечивается обратная связь между пользователями и разработчиками, которая используется для изменения или корректировки спецификации требований к программному продукту.

**8. RAD методология.**

Основная идея – вовлечь пользователей в разработку ПО. Программисты реализуют множество решений для типовых и конкретных задач, заказчик выбирает то, что ему нравится больше всего (как будто сам строит дом из красивых кирпичиков).

RAD - разновидность инкрементной модели, популярная в данный момент. Пользователь принимает непосредственное участие в разработке. При Помощи интерфейса пользователь способен создавать простейшие функции. В данной модели компоненты или функции разрабатываются несколькими командами параллельно, как несколько мини-проектов. Временные рамки одного цикла жестко ограничены. Созданные модули затем объединяются в один рабочий прототип. Используют case системы

**9. Спиральная модель.**

«Спиральная модель» похожа на инкрементную, но с акцентом на анализ рисков. Она хорошо работает для решения критически важных бизнес-задач, когда неудача несовместима с деятельностью компании, в условиях выпуска новых продуктовых линеек, при необходимости научных исследований и практической апробации.

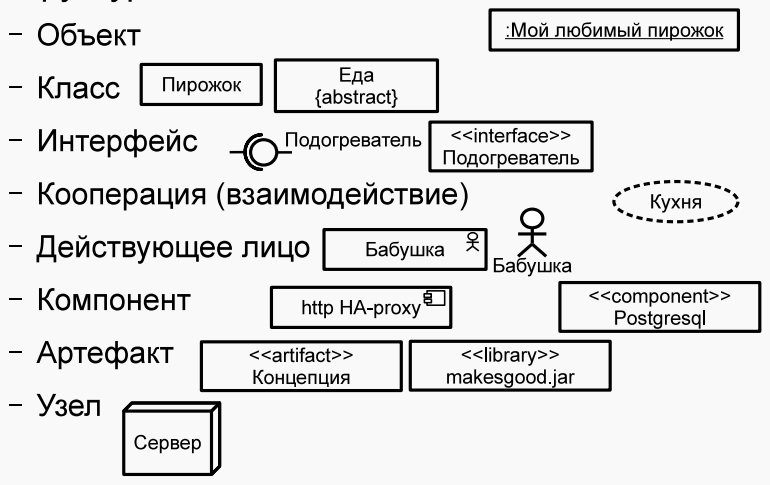
Спиральная модель предполагает 4 этапа для каждого витка:

1. планирование;
2. анализ рисков;
3. конструирование;
4. оценка результата и при удовлетворительном качестве переход к новому витку.

**10. UML Диаграммы: Структурные и поведенческие.**

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

Структурные сущности – структура программы, которая задается языком программирования.

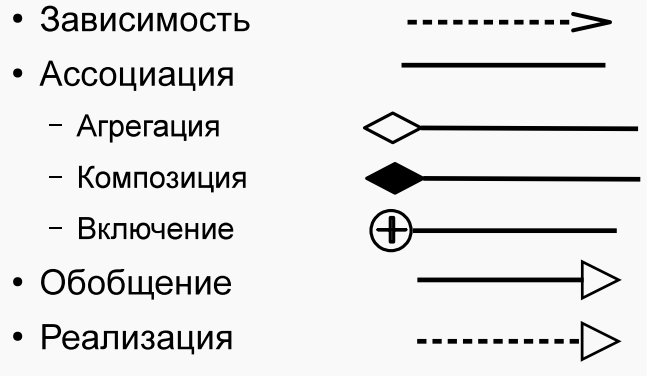




**11. UML:** **Use-case модель.**

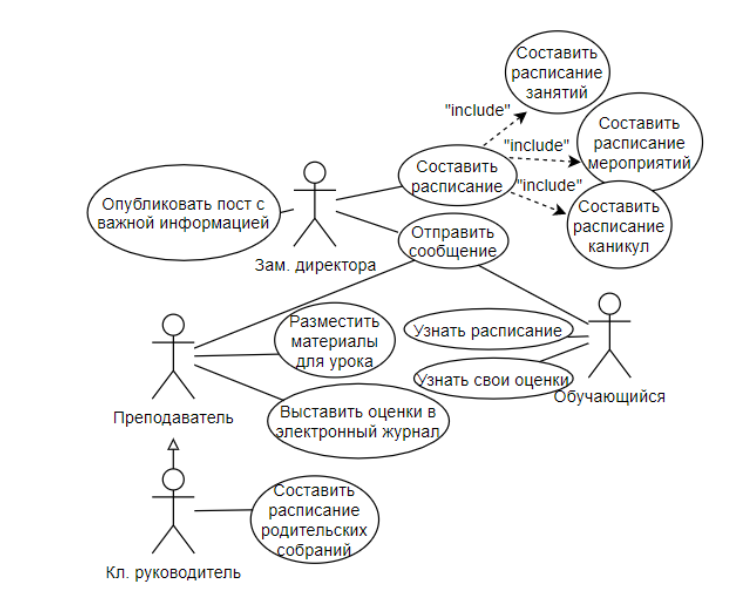
Use-case модель - диаграмма, описывающая, какой функционал разрабатываемой программной системы доступен каждой группе пользователей. На диаграммах UML для связывания элементов используются различные соединительные линии, которые называются отношениями. Каждое такое отношение имеет собственное название и используется для достижения определённой цели.

Отношения UML



* Зависимость - необходимо показать что исходный элемент зависит от целевого элемента и изменение целевого аргумента может повлиять на исходный.
* Ассоциация - Отношение ассоциации предназначено **только** для соединения актёров и вариантов использования.
  + Агрегация – целевой элемент является частью исходного элемента и может существовать как отдельный. Пример – машина и колесо.
  + Композиция – форма агрегирования, где часть не может существовать без целого. Пример - стул
  + Включение – исходный элемент содержит целевой. Пример – пакет
* Обобщение - Отношение обобщения означает, что некоторый актёр (вариант использования) может быть обобщён до другого актёра (варианта использования).
* Реализация – объект реализует данный интерфейс.

Пример



**12. UML: Диаграмма классов.**

Диаграмма классов UML иллюстрирует структуру системы, описывая классы, их атрибуты, методы и отношения между объектами.

Три наиболее важных типа отношений в диаграммах классов, это:

Ассоциация, которая представляет отношения между экземплярами типов, к примеру, человек работает на компанию, у компании есть несколько офисов.

Наследование, которое имеет непосредственное соответствие наследованию в Объектно-Ориентированном дизайне.

Агрегация, которая представляет из себя форму композиции объектов в объектно-ориентированном дизайне.

**13. UML: Диаграмма последовательности**

Диаграмма последовательности моделирует взаимодействие объектов на основе временной последовательности. Она показывает, как одни объекты взаимодействуют с другими в конкретном прецеденте.

Диаграммы последовательностей используются для уточнения диаграммы прецедентов. Они обычно содержат объекты, которые взаимодействуют в рамках сценария, сообщения, которыми они обмениваются, и возвращаемые результаты, связанные с сообщениями.

Объекты Обозначаются прямоугольниками с подчеркнутыми именами.

Сообщения(вызовы методов) - линиями со стрелками.

Возвращаемые Результаты- пунктирными линиями со стрелками.

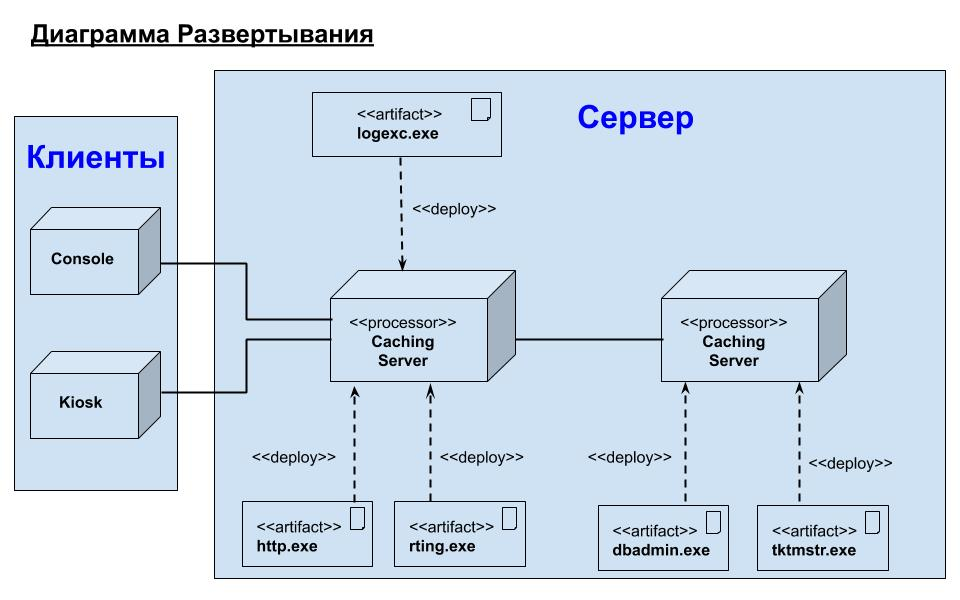
Прямоугольники на вертикальных линиях под каждым из объектов показывают “время жизни” (фокус) объектов.

**14. UML: Диаграмма размещения**

Диаграмма размещения помогает моделировать физический аспект объектно-ориентированной программной системы. Это структурная схема, которая показывает архитектуру системы, как развертывание программных артефактов.

Артефакты представляют собой конкретные элементы в физическом мире.

Диаграмма размещения показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения. Графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего зависит в том числе и производительность системы. Такая диаграмма может помочь решить множество вспомогательных задач. Это единственная диаграмма, на которой применяются трехмерные обозначения: узлы системы обозначаются кубиками.



**15. \*UP методологии (90-е). RUP: основы процесса.**

Три человека решили объединить свои методики и создали RUP. Rational Unified Process – это методология создания программного обеспечения, оформленная в виде размещаемой на Web базы знаний, которая снабжена поисковой системой. Предоставляя каждому члену группы доступ к той же самой базе знаний, вне зависимости от того, разрабатывает ли он требования, проектирует, выполняет тестирование или управляет проектом - RUP гарантирует, что все члены группы используют общий язык моделирования, процесс, имеют согласованное видение того, как создавать ПО. В качестве языка моделирования в общей базе знаний используется Unified Modeling Language (UML), являющийся международным стандартом.

Продукт Rational Unified Process (RUP) разработан и поддерживается Rational Software. Если представить его на графике, то по оси абсцисс будет идти время (фазы), а по оси ординат – дисциплины – анализ, дизайн, тестирование… В данной методологии четко дали описание всем её составляющим.

**16. RUP: Фаза «Начало».**

Цель: возможно ли сделать этот проект с учетом архитектуры времени и денег, которые у нас есть. Также оцениваются риски и строиться примерный график работ.

Веха: Пришли ли заинтересованные стороны к согласию в оценке стоимости, срокам, технологиям. Риски должны быть обсуждены и проверены.

**17. RUP: Фаза «Проектирование».**

Цели: разработка и тестирование стабильной и неизменной системы, создание одного или нескольких прототипов системы, которые определяют исполняемую архитектуру. Исполняемая архитектура – полностью законченные на базе выбранных технологий несколько характерных функций разрабатываемой системы. На основе прототипов и результатов тестирования уточняются планы разработки и производиться переоценка, контроль рисков, уточняются сроки и стоимость системы.

Веха: Убедится в том что концепция требования и архитектура стабильны, протопипы надежны и показывают отсутствие рисков, Договоренность по времени разработки и цене.

**18. RUP: Фаза «Построение».**

Цели: экономически эффективно, с надлежащим качеством, так быстро как возможно разработать продукт. Экономически эффективно – изменений в архитектуру вноситься уже не должно, ресурсы команды не должны тратится на не основанную работу. Проводится анализ проектирование, разработка и тестирование продукта – создаются выпуски. Подготавливаются продукт, места установки, и пользователи к использованию.

Веха: Готовы ли все стороны к передаче продукта пользователям? Продукт должен быть стабилен.

**19. RUP: Фаза «Внедрение».**

Цели: данная фаза предназначена для запуска продукта в использование и подтверждение пользователями пригодности продукта к для их нужд. Проводится бета тестирование, обучение пользователей, запускается маркетинг и продажи. Пользователи должны самостоятельно пользоваться ПО и поддержку.

Веха: Удовлетворены ли пользователи?

**20. Манифест Agile (2001).**

Главным минусом rup считалось то что очень сложно как то менять продукт заказчиками со временем. Требовался подход к разработке, который позволил бы быстро и гибко на их изменения. Именно это было заложено в Agile манифест.

Во главу ставится требования заказчика, и реакция на них. Данный подход практически невозможен если на проект выделен фиксированный бюджет. Разработчики должны постоянно демонстрировать результат и общаться с заказчиком. Ещё один принцип – максимально сокращение расходов разработчиков, не относящихся к созданию кода.

* Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.
* Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
* Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
* На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
* Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
* Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
* Работающий продукт - основной показатель прогресса.
* Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.
* Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
* Простота - искусство минимизации лишней работы - крайне необходима.
* Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
* Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

**21. Scrum.**

Основной и единственный служебный артефакт Scrum бэклог. Упорядоченный по приоритетам список требований с оценкой трудоемкости разработки. Существуют Бэклог продукта, обычно более общий и состоящий из бизнес-требований, и бэклог спринта-более детальный, с учетом технических особенностей реализации.

Спринт- время от двух до четырех недель, за которые разработчики реализуют выбранный набор требований. Каждый спринт заканчивается демонстрацией результатов заказчику. Также через несколько спринтов проводятся ретроспективы, в рамках которых может проводиться работа над ошибками и перераспределение обязанностей.

Команда в Scrum небольшая, от 3 до 10 человек. Также Выделяется Особая роль -Product Owner- это человек, определяющий порядок разработки требований из бэклога. Кроме этого, отдельно выделяется Скрам-мастер, ответственный за проведение скрам-митинга, который помогает команде планировать спринт и следит за внутренними отношениями в команде. Ежедневно утром производится скрам-митинг, где каждый отчитывается о проделанной работе, возникших проблемах, и том, что он собирается сделать к следующей встрече.

Достоинства Scrum - простота, минимум административной работы и документов и максимальная концентрация на работоспособном коде. Scrum лучше всего подходит для проектов с небольшой командой разработки.

**22. Disciplined Agile 2.X (2013).**

В Disciplined Agile 2.Х подход к делению на фазы и дисциплины похож на RUP, но основной цикл разработки построен на базе гибких методов.

Помимо деления процесса на фазы, которых предлагается три (Начало,Построение и Внедрение), и описания каждой роли в разработке, рассматривает процессы, выходящие за рамки собственно процесса разработки.

К ним относятся:

●Управление архитектурой и повторным использованием кода

●Управление персоналом, служба поддержки и текущих операций компании-разработчика

●Управление портфолио компетенций

●Непрерывное улучшение процессов разработки и вспомогательных процессов В последнее время отдельное внимание уделяется сбору и анализу большого объема данных.

**23. Требования. Иерархия требований.**

Требования – условия или возможности, которым должна соответствовать система.

Требование – подробное описание того, что должно быть реализовано, но не как должно быть реализовано.

Иерархия требований

* Потребности – пожелания для решения конкретной задачи
* Функции – обобщение на формальный язык, уточнение потребность
* Требования ПО – формулирование конкретных, финальных деталей реализации.

**24. Свойства и типы требований (FURPS+).**

Требования должны быть однозначными и полными. Они не должны противоречить друг другу.



FURPS+ - модель описания требований в RUP

* Функциональные – определяют, что должна делать система
* Нефункциональные – описывают характеристики и ограничения, которые накладываются на систему.
  + Usability – требования к ПО
  + Reliability – требования к надежности
  + Performance – требования к производительности
  + Supportability – требования к условиям поддержки
* Дополнительные – требования к реализации, физ. Требования.

**25. Формулирование требований. Функциональные требования.**



Функциональные требования определяют

* Набор функциональных требований – набор свойств продукта, необходимой для выполнения конкретной деятельности
* Возможности ПО
* Требования к безопасности – обычно включает в себя метод аутентификации, список ролей итд…

**26. Требования к удобству использования и надежности.**

Требования к удобству использованияотвечают на вопрос как?

* Учет особенностей пользователя (для пенсионеров…)
* Эстетические требования
* Согласованность пользовательского интерфейса
* Требования к справочной подсистеме
* Требования к документации
* Требования к методичкам

Требования к надежности – предназначены для фиксирования способности ПО безоткатно функционировать в течение определенного периода времени.

* Частота и обработка отказов
* Способность системы восстанавливать продуктивное функционирование
* Предсказуемость поведения
* Точность
* Среднее время между отказами

**27. Требования к производительности и Поддерживаемость.**

Требования к производительности включает в себя большой различных требований – они взаимосвязаны.

* Скорость решения задач
* Эффективность
* Готовность системы к решению задач
* Пропускная способность
* Время отклика
* Время восстановления
* Использование ресурсов

Требования, обеспечивающие поддержку системы:

* Расширяемость
* Адаптируемость
* Поддерживаемость
* Совместимость
* Способность задавать конкретную конфигурацию
* Возможность проведения профилактик и обслуживания
* Требования к установке на разных системах
* Локализуемость для разных языков

**28. Атрибуты требований.**

Требования могут иметь различные атрибуты, котрорые помогают менеджерам сортировать их. Один из подходов – метод MuSCoW

* Приоритет
  + Must have – фундаментальные системные требования, без которых финальная реализация системы не имеет смысла
  + Should have – Важные требования, которые следует реализовать при наличии времени в процессе разработки
  + Could have – потенциально возможные
  + Won’t have – могут быть реализованы в следующих версиях системы.
* Статус
  + Предложенные
  + Одобренные
  + Отклоненные
  + Включенные
* Трудоемкость – на сколько сложна реализация
  + Человеко – часы
  + Функциональные точки
* Риск
* Стабильность
  + Высокая
  + Средняя
  + Низкая

**29. Описание прецедента.**

Изображение выглядит как текст, стол

Автоматически созданное описание

**30. Риски. Типы Рисков.**

Риск – потенциально опасный для объекта фактор.

Риск – сочетание вероятности события и его последствий

Типы рисков

* Прямые и непрямые – Можем управлять этим риском или нет
* Ресурсные – организационные, финансовые, люди, время
* Бизнес риски – появляются из за взаимодействия с другими организациями(конкуренция, подрядчики, плохие решения)
* Технические риски – находятся в компетенции разработчиков (границы проекта, технологические)
* Политические риски – связаны с изменениями сфер влияния заказчика
* Форс-мажоры- нельзя предугадать

**31. Управления рисками. Деятельности, связанные с оценкой.**

В процессе оценки риска специалист по работе с риском анализирует его, определяет степень его серьезности и продумывает план работы с ним. Оценка может производиться по разным методикам, например, по списку заранее подготовленных вопросов. Все источники рисков в компании-разработчике распределяются по иерархической структуре. Первым ее уровнем являются классы рисков:

● риски, связанные с самой разработкой (Product Engineering);

● риски, связанные с окружением, где осуществляется разработка (Development Environment);

● риски, связанные с программным обеспечением (Program Constraints).

Риск, который просто идентифицировать и определить место его возникновения в компании, называется известным.

Место возникновения неизвестного риска можно предположить, но данная команда разработки еще с ним не сталкивались.

Место возникновения непознаваемых рисков невозможно предугадать, а способ борьбы разработать заранее.

После идентификации рисков необходимо провести их всесторонний анализ. Анализ рисков связан с выявлением скрытых взаимосвязей неопределённых ситуаций и источников рисков.

Два основных параметра риска - вероятность его наступления и масштаб (магнитуда) возможных потерь.

Экспозиция риска - произведением вероятности наступления риска и величины денежных потерь.

**32. Управления рисками. Деятельности, связанные контролем и управлением.**

Способы реакции на риск:

● Избегание - разработать комплекс мероприятий, которые помогут отсрочить или исключить вероятность его наступления.

● Перенос - можно сделать так, чтобы под него попал кто-то другой.

● Прием - мы соглашаемся с тем, что риск может наступить, заранее составляются планы действий, какие действия необходимо предпринимать в случае его наступления.

Способы разрешения рисков:

● Построение как можно большего количества прототипов.

● Построение множества различных моделей функционирования.

● Аналитическая работа над ошибками.

● Подбор квалифицированного персонала.

**33. Изменение. Общая модель управления изменениями**

Изменение – контролируемое, журналируемое обновление системы.

У изменений могут быть атрибуты, таки как идентификатор, дата, ответственный, описание…

Общая модель изменений

Заказчик хочет добавить новый функционал в программу или исправить баг. Любая из этих деятельностей приводит к созданию документов: требования к системе (requirements) или отчет о проблеме (problem report). Далее формируется запрос на изменения (change request).

Далее запрос поступает к менеджеру проекта, который должен определить техническую необходимость и стоимость данного изменения. Оба фактора бумажно фиксируются. Далее запрос поступает к комитету по изменениям, который оценив все за и против выносит решение и меняет статус в change request. Если было принято решение принять изменение, то организуется планирование изменения в системе(определяются ресурсы, время, график).

Затем изменения передаются на реализацию, где после внесенных изменений формируются документы – отчет о тестировании, документация и выпуск системы. В конце изменения передаются менеджеру, проверяются и утверждаются.

**34. Системы контроля версий. Одновременная модификация файлов.**

Система контроля версий – управляет изменениями в программном коде.

Существуют три типа СКВ:

● На основе файловой системы. Устаревшая система, разработчики пользовались одной машиной с общим доступом к файлам. Система создавала файлы слежения за директорией и позволяла следить в рамках одной файловой системы.

● Централизованная, с единым репозиторием. – хранилище исходных кодов на сервере и удаленный доступ к ним по протоколам. Например, Apache Subversion.

● Распределенная. В ней существует центральный репозиторий, из которого пользователи скачивают данные на свои локальные репозитории. Обратно эти данные попадают после локальных проверок. Например, Git.

Главной проблемой при работе с СКВ является необходимость одновременной работы с одним и тем же файлом. Одновременное редактирование приводило к конфликтам. Есть два подхода к решению данной проблемы

1. Lock-modify-unlock: при работе пользователя с файлом он блокируется для других. Это замедляет работу. Данный подход характерен для систем с общей файловой системой

2. Copy-modify-merge: каждый пользователь копирует себе весь репозиторий работает с ним, и изменения сливаются.

**35. Subversion. Архитектура системы и репозиторий.**

Уровень хранения может иметь две технические реализации: размещать в БД Berkeley DB, либо в файловой системе FSFS.

Доступом к репозиторию управляет демон svnserve или сервер Apache.

Удаленный доступ может осуществляться по нескольким протоколам svn+https и ssh+svn.

Клиент SVN не только осуществляет передачу данных с репозитория, но и управляет локальной копией файлов.

Репозиторий – набор файлов проекта, организованный определенным иерархическим образом.

* Организация файлов такова: в каталоге trunk происходит основной процесс разработки.
* Стабильные копии за некий момент времени копируются в branch (хранение модификаций продукта)
  + Releases - значимые версии продукта
  + Features – работа над версиями без влияния на trunk
* tag (функционально целостные изменения) – исправление дефектов

В SVN каждый коммит повышает ревизию репозитория на 1.

**36. Subversion: Основной цикл разработчика. Команды.**

В начале работы в каталоге существует локальная версия предыдущего дня. Для начала работы надо скачать с сервера обновление локальной копии. Скачиваются не только данные, но и метаданные. Дальше работаем с данными.

1. svn update - обновление рабочей копии

2. svn add, delete, copy, move, mkdir - добавить, удалить, копировать, переместить, создать директорию

3. svn revert - откат изменений

4. svn commit - фиксация изменений

**37. Subversion: Конфликты. Слияние изменений.**

Конфликты бывают: содержимого файлов, структуры файлов

При возникновении конфликта можно изменить файл в редакторе, посмотреть несовпадения, подтвердить, что конфликт решен текущей версией файла, посмотреть все конфликты, подтвердить мои или чужие версии, отложить или запустить внешнее средство устранения конфликта.

Чаще всего конфликт откладывают и идут решать проблемы с автором конфликтных изменений. Тогда создаются три файла: текущий, файл до правок и с правками.

Обычно в средствах разработки есть средства Diff для сравнения версий. Существует конфликт структуры, когда не совпадает структуры репозиториев.

При работе над параллельными ветками часто необходимо изменения из одной ветки сливать с изменениями в другой, образуя одну общую ветвь. Для этого предназначена команда svn merge. То есть мы берем из нашей ветки и все изменения, что мы проводили с ней, и присваиваем их другой, второй.

Процесс очень похож на один большой коммит.

**38. GIT: Архитектура и команды.**

В Git существует origin - центральный удаленный репозиторий, который хранится на облачных сервисах. У каждого разработчика есть локальный репозиторий, который является клоном центрального. Разработчик выбирает необходимую ветвь и получает копию файлов, которую он может изменить. Во время работы с копией разработчик указывает какие файлы включать в фиксацию изменений. Они помещаются в Stage Area - область, из которой файлы будут включены в коммит. Для помещения в центральный репозиторий разработчик использует команду git push. Другой способ фиксации в удаленном репозитории - конфигурация запросов на фиксацию изменений - pull request.

Команды:

● branch – вывод всех локальных веток

● status – показывает состояние текущих файлов

● checkout – переключение между ветками

● rebase - создать новую ветвь

● diff – показать все изменения

● clone - скопировать удаленный в локальный

● init - создать репозиторий

● add - добавить файлы в stage area

● commit - зафиксировать изменения на локальном репозитории

● push - отправить изменения на удаленный репозиторий

● fetch - загрузить изменения с удаленного репозитория

● merge - слить изменения

● pull - то же самое что fetch+merge

● fork - создать ответвление существующего проекта из удаленного репозитория

**39. GIT: Организация ветвей репозитория.**

В ветви master находятся версии, готовые к поставке заказчику.

Функционал там определен, дефекты известны. Основная разработка происходит в develop.

Feature - ветви разработки отдельных функциональных требований.

Release - ветвь подготовки новых продуктивных версий для заказчика.

Hotfix - ветвь для срочного исправления критических ошибок.

**40. GIT: Плагин git-flow**

Каждое проделывают в git действие над рабочей копией требует большого количества команд. Для сокращения времени был создан плагин git flow, позволяющий работать с версиями именно в терминах версий, а не операций, без точного знания команд и последовательностей действий.

Для создания новой feature в git flow используется команда git flow feature start FEATURE NAME. Когда надо указать на окончание изменений, используется git flow feature finish FEATURE\_NAME. Остальные команды формируются аналогичным образом. Все это позволяет пользоваться репозиторием на более высоком, независимом от синтаксиса git уровне.

**41. Системы автоматической сборки: предпосылки появления**

При сборке проекта приходится постоянно писать одни и теже команды в терминале, их объем может быть довольно большим, и программисты захотели автоматизировать процесс сборки своих проектов.

Первое логичное решение было создание скриптов для операционной системы, но если задуматься, то и у этого подхода есть ряд минусов:

* При расширении проекта, приходится постоянно дописывать скрипт, и держать его логику в голове.
* Весь скрипт нужно писать руками
* Данные скрипты платформо-зависимы

Если ваш продукт будет собираться на стороне пользователя, то у него может быть совершенно иное аппаратное окружение, архитектура, операционная система, и все это нужно учитывать.

Также было желание попытаться ускорить процесс сборки, так как некоторые большие проекты приходилось компилировать днями. Возможно добавить параллелизма.

Так и появились системы автоматической сборки, которые:

* Используют специальные языки для организации компиляции
* Позволяют автоматически определять архитектуру
* Могут организовать автоматическую сбору
* Поддерживают параллельный режим работы

**42. Системы сборки: Make и Makefile.**

Make – система автоматической сборки низкого уровня, разработанная для UNIX систем. У него есть свой язык. Который в неявном виде позволяет определить последовательность действий при сборке, и шаги которые необходимо предпринять к сборке. Данная система является императивной – сообщает что сделать (скомпилировать эти файлы, скопировать их в эту папку а затем закинуть все это в архив).

* Для начала нам нужно скачать утилиту make
* В рабочей директории создаем makefile и прописываем там правила сборки нашего проекта

Также мы можем использовать переменные: объявляем переменную в начале makefile: path=/tmp, а потом используем её в нужном месте: ${path}

Пример: пусть у нас есть файл main.c, и как только мы будем его менять, он сразу будет перекомпилироваться

Main: main.c

gcc -o Main main.c

Данный makefile состоит из одного правила, которое состоит из цели Main, реквизита(то что меняется) main.c и команды gcc -o Main main.c

Теперь если нам необходимо перекомпилировать проект, мы просто пишем make

**43. Системы сборки: Ant. Команды Ant.**

Так как make создавался для работы с файлами c, ant создавался для сборки проектов на джава. Он также как и make относится к императивным системам сборки, однако все правила и зависимости прописываются на xml. Плюсом данного подхода является то, что xml очень удобно проверять на корректность синтаксиса, как минус он очень громоздкий и не удобен для чтения человека. Можем использовать константы задавая их:

* Значением: <property name=”build.dir” value=”/tmp”/>
* Переменной окружения: ${env.PATH}
* Загружаться из файла: <property file=”build.properties”/>

Самые базовые команды:

* Echo – вывод сообщения в консоль
* Mkdir – создать директорию
* Delete – удалить файл или диркторию
* Javac – компиляция java кода
* Java – запуск java проекта
* Jar – сздание jar архива
* Junit – запуск тестов

В ходе написания сценария мы будем писать несколько целей, для каждой свой сценарий, и в конце сможем вызвать как дефолтную так и определенную обратившись по имени. Пример сценария на ant: у нас есть файл HelloWorld.java

<project name="HelloWorld" default="run">

<target name="compile">

<mkdir dir="build/classes"/>

<javac destdir="build/classes" includeantruntime="false">

<src path="src"/>

</javac>

</target>

<target name="run" depends="compile">

<java classname="HelloWorld" classpath="build/classes"/>

</target>

<target name="clean">

<delete dir="build"/>

</target>

</project>

Теперь сценарий содержит три target (команды): compile (компиляция файла(ов) .java), run (запуск файла .class), clean (удаление папок с результатами компиляции). При этом compile содержит два tasks – mkdir и javac. Обратите внимание на зависимость: target run предварительно вызовет compile. Кроме того run – это target по умолчанию для проекта.

**44. Системы сборки: Ant-ivy.**

Один из главных минусов ant – ручное подключение библиотек, их скачивание и поддержка. Для того чтобы упростить этот процесс был придуман ivy – менеджер зависимостей для Ant. Он может работать с репозиторием Maven и самостоятельно качать необходимые зависимости.

<project name="ivyWithAnt"default="default" xmlns:ivy="antlib:org.apache.ivy.ant">

<property name="libdir" value="../lib"></property>

<target name="default" description="description">

<ivy:retrieve />

</target>

<target name="clearIvyCache" description="description">

<ivy:cleancache/>

</target>

</project>

**45. Системы сборки: Maven. POM. Репозитории и зависимости.**

Со временем проекты становились все больше и сложнее и постоянно поддерживать команды, правила их выполнения и их корректность становилось все сложнее. Так на свет начали появляться декларативные средства сборки например Maven. Декларативный подход заключается в том, что мы говорим не то как собирать а то что мы хотим собрать, и наша система сама решает как выполнить данную задачу.

Maven управляется на базе POM – Project Object Model, это XML файл, состоящий из нескольких частей. Он содержит:

* Имя, версия, тип
* Местонахождение кода
* Зависимости
* Плагины
* Профили

У Maven есть несколько целей, базовая – install. Она заключается в том, чтобы собрать на локальной машине продукт. Обычно он помещается в папку target. Алгоритм следующий:

* Maven читает зависимости из pom.xml и определяет зависимости от внешних продуктов
* Далее все отсутствующие зависимости он скачивает с удаленного репозитория на локальный, и сохраняет их там
* Затем осуществляется сборка
* Собранный продукт помещается в target

Если нам необходима какая-то зависимость то maven начинает последовательно проверять репозитории:

* Наш, локальный репозиторий
* Центральный репозиторий maven
* Удаленный репозиторий самих разработчиков, создавших библиотеку

Также у зависимостей существует scope – время их применения (compile, provided, test).

**46. Maven: Структура проекта. GAV.**

В Maven в отличии от Ant существует система расположения каталогов по умолчанию.

* Target – папка для артефактов
* Src/main – основные исходные файлы
  + Src/main/java – исходники java
  + Src/main/webapp – web-страницы, jsp, js, css
  + Src/main/resources – файлы не нуждающиеся в компиляции
* Src/test – исходные данные тестов
  + Src/test/java – тесты на java
  + Src/test/resources – ресурсы для тестов

Система наименования модулей в Maven строится по принципу GAV – groupId:artifactID:version. Любая внешняя зависимость описывается при помощи данной системы наименования. Пример

<project>

<modelVersion>1.0.0</modelVersion>

<groupId>javax.portlet</groupId>

<artifactId>portlet-api</artifactId>

<version>2.0</version>

</project>

**47. Maven: Зависимости. Жизненный цикл сборки. Плагины.**

Зависимости также имеют GAV синтаксис, scope, и тип. Пример

<project>

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>//provided = во время развёртывания в веб контейнере

<dependency>

<dependencies>

</project>

Жизненный цикл Maven – последовательность сборки.

* Генерируются исходные коды и ресурсы
* Компиляция исходников
* Компиляция тестов
* Происходит тестирование
* Формируется архив
* Происходит интеграционное тестирование
* Загрузка в target
* Deploy нашего проекта

Плагины выполняют все операции в Maven, именно они содержат всю логику сборки проекта. Каждый плагин имеет точку входа – цель связанную с ЖЦ сборки.

**48. Системы сборки: Gradle. Преимущества и файл сборки.**

Проблемой сборки на Maven является то, что мы используем уже написанные плагины. Если вдруг нам понадобится плагин, которого нет в базе Maven, то нам придется самим его создавать. Также многим не нравится писать в xml файле. Так был создан Gradle – система декларативной сборки, которая:

* Нейтральна к языкам программирования
* Может работать с различными репозиториями зависимостей (в.т.ч. Maven)
* Использует язык dsl для описания сборки
* Экономит время сборки проекта

Пример файла сборки build.gradle

Apply plugin: “java”

Apply plugin: “application”

mainClassName= “ru.ifmo.cs.Bcomp-NG”

repositories{

mavenCentral()

}

Dependencies{

Compile “log4j”:log4j-core:2.12.1

}

Jar{

Manifest.attributes(“Main-Class”: Main);

}

**49. Системы сборки: GNU autotools. Создание конфигурации проекта.**

GNU autotools основывается на микропроцессоре общего назначения m4. Макропроцессор - это программа, преобразующая входной текст в выходной, руководствуясь правилами замены последовательностей символов, называемых правилами макроподстановки.

Основные команды: ./configure; make; sudo make install. Их последовательного выполнения достаточно для установки собранного продукта.

GNU autotools полностью платформонезависимый. Процесс создания конфигурации проекта:

1. С помощью утилиты autoscan последовательно сканируется существующий исходный код. Выделяются участки кода, которые могут зависеть от особенностей платформы. Формируется шаблон конфигурационного файла, который затем редактируется вручную. В результате получается файл configure.ac.
2. Вручную создается Makefile.am. В нем в явном виде указываются названия исполняемых программ, из каких исходных файлов они должны быть собраны, и другие зависимости. Такой файл создается в каждом из подкаталогов исходных файлов.
3. autoheader создает шаблон config.h.in для файла config.h
4. aclocal проверяет, что установлено на локальной системе разработчика, и должно быть использовано в проекте.
5. Затем запускаются утилиты automake и autoconf, в результате чего в дистрибутиве появляются файлы Makefile.in и configure, которые позже используются при определении текущей конфигурации на целевой системе, где будет собираться ПО.

**50. Системы сборки: GNU autotools. Конфигурация и сборка проекта.**

GNU autotools основывается на микропроцессоре общего назначения m4. Макропроцессор - это программа, преобразующая входной текст в выходной, руководствуясь правилами замены последовательностей символов, называемых правилами макроподстановки.

Основные команды: ./configure; make; sudo make install. Их последовательного выполнения достаточно для установки собранного продукта.

Процесс конфигурации и сборки проекта:

1. После скачивания ПО запускается команда ./configure, и на основании текущей конфигурации создается файл config.h и необходимые платформо зависимые файлы.

2. После этого используется команда make, и все собранные файлы при помощи команды make install распределяются в необходимые системные каталоги.

**51. Сервера сборки/непрерывной интеграции**

Для выполнения сборки в автоматическом режиме существуют отдельные средства, называемые серверами сборки или серверами непрерывной интеграции. Их основное назначение - сборка новой версии продукта при наступлении заданных администратором или разработчиком условий (например, изменение исходного кода или наступление определенного времени - например, ночная сборка).

Помимо сборки, билд-сервер может проводить тесты, снимать метрики с программного кода, производить автозапуск и т. д.

Сервер сборки также предоставляет доступ ко всем существующим собранным версиям продукта для скачивания и немедленной установки у пользователя.

**52. Основные понятия тестирования. Цели тестирования.**

Mistake – ошибка, просчет человека

Fault – дефект, изъян, который произошел в результате ошибки

Failure – неисправность, отказ, сбой (внешнее проявление дефекта)

Error – невозможность выполнить задачу вследствие отказа

BUG – может означать все что угодно)

Цели тестирования

* Обнаружение дефектов
* Повышение уверенности в уровне качества
* Предоставление отчета о системе
* Предотвращение дефектов

**53. Понятие полного тестового покрытия и его достижимости. Пример.**

Тестовое покрытие – насколько код приложения покрыт тестами, которые могут находить известные и потенциальные дефекты.

Полное тестовое покрытие – покрытие тестами всего кода и всех возможных вариантов развития событий, в данном коде.

Считается, что невозможно обеспечить полное тестовое покрытие программ. Например, если взять калькулятор, который умножает два 32разрядных числа друг на друга, то мы будем иметь вот что:

Для того чтобы зафиксировать всевозможные варианты развития событий – таблица тестирования, где нам нужно будет хранить два числа + ответ + флаг ответа, нам понадобится минимум (32+32+64+1) \* 2^64 бит. Это настолько много, что данный объем памяти мы сможем расположить только на самых передовых хранилищах.

Плюс к этому нам нужно будет как заранее посчитать все ответы(желательно на другой эталонной машине), так и провести само тестирование, и если использовать стандартный процессор с частотой 3ггц, то используя нехитрую формулу мы получим:

И это только тестирование одной функции умножения.

**54. Статическое и динамическое тестирование.**

Статическое тестирование – тестирование которое не связано с запуском тестов на реально коде. Оно подразумевает проверку на этапах спецификации, проектирование, требований. Его задача постараться найти такие места, которые или невозможно осуществить, или найдется более оптимальное решение. В будущем это сэкономит много денег и времени.

Динамическое тестирование – происходит после написания кода, и тестируется на нем. В него входит модульное тестирование, интеграционное тестирование, системное, системы в целом.

**55. Автоматизация тестов и ручное тестирование.**

Ручное тестирование – взаимодействие профессионального тестировщика и софта с целью поиска бага. Его

стоимость зависит от тестировщика.

|  |  |
| --- | --- |
| Плюсы   * Пользовательский фидбэк * UI-фидбэк, его можно протестировать только вручную * Дешевизна * Тестирование в реальном времени * Возможность исследовательского тестирования | Минусы   * Человеческий фактор * Трудоемкость повторного использования * Невозможность нагрузочного тестирования |

Автоматизированное тестирование – написание кода. Ожидаемые сценарии сравниваются с теми, что получает пользователь, указываются расхождения. Играет важную роль в тяжелых приложениях с большим количеством функций.

|  |  |
| --- | --- |
| Плюсы   * Возможность нагрузочного тестирования * Экономия времени * Возможность повторного использования | Минусы   * Дороговизна * UI тестирование не может быть проведено в полной мере * Отсутствие человеческого взгляда |

**56. Источники данных для тестирования. Роли и деятельности в тестировании.**

Источники данных для тестов

* Описания ПО – метод черного ящика (мы смотрим как наша программа справляется с тем, какие требования ей дали) мы не знаем как утроена наша программа
  + Спецификация, требования, дизайн
  + Запуск и сравнение с эталоном
* Исходный код – метод белого ящика – мы делаем декомпозицию нашего кода, и строим граф, в каких состояниях может находится наша программа. Далее мы начинаем покрывать наш граф тестами.
  + Переходы, утверждения, условия
  + Анализ путей, структуры
* Опыт
* Модели UML…

Виды тестировщиков

* Проектирование тестов – на основе формальных критериев, знаний, опыта
* Автоматизация тестов – знание средств, скриптов
* Исполнение тестов – обычные ребята
* Анализ результатов – знания предметной области

**57. Понятие тестового случая и сценария.**

Тестовый случай состоит из набора входных данных (данные, на которых разрабатываемое ПО должно вести себя определенным образом с точки зрения спецификации или других требований), предусловий выполнения, ожидаемых результатов (включают в себя входные данные, изменения в данных, и любые другие последствия теста) и постусловий.

Тестовые случаи должны быть повторяемыми, автоматизируемыми, при правильных действия выводить корректный результат, при неправильных – корректные сообщения об ошибках.

Тестовый сценарий – набор тестовых случаев

Изображение выглядит как стол

Автоматически созданное описание

**58. Выбор тестового покрытия и количества тестов. Анализ эквивалентности.**

При планировании тестирования следует соблюдать баланс, между качеством и скоростью вывода продукта в эксплуатацию. Чем больше тестов – тем больше качество, однако также растут время и ресурсы. Может произойти ситуация, что конкуренты, выйдут на рынок раньше вас, и захватят его хотя их продукт более сырой.

Есть несколько вариантов выбора тестового покрытия

* Эквивалентное разбиение(Анализ эквивалентности) – представляет из себя анализ разных значений, с которыми функция ведет себя одинаково. Она разбивается на части, для каждой из которых существует свой набор тестов. Также составляются тесты для пограничных значений. Если участков разбиения не много, то это позволит сильно сократить количество тестовых случаев.
* Таблица решений – составляется таблица, отображающая комбинации входных данных, с соответствующими выходными данными, которая используется для проектирования набора таких тестовых случаев.
* Таблицы переходов – выделяются явные состояния внутри системы, определяются переходы между этими состояниями, которые потом покрываются тестами.
* Сценарии использования – берутся требования, под них пишутся тесты и подставляются реальные значения.

**59. Модульное тестирование. Junit 4.**

Модульное тестирование - тестирование отдельных компонентов ПО. Для изолирования модулей применяется драйверы - компоненты, вызывающие модули и обеспечивающие последующее тестирование. Модуль – компонент, который необходимо протестировать отдельно от остального программного продукта. Модули определены в дизайне программы. Изолирование модулей предполагает замену вызывающих модулей драйверами, а зависимых заглушками.

Junit - простейший фреймворк, позволяющий создать модульные тесты и выполнить их в определенном окружении. Junit построен на аннотациях. Метод, организующий тестирование, помечается @Test. При этом фреймворк тестирования последовательно просматривает загружаемые классы при помощи рефлексии и ищет эту аннотацию. Внутри тестового метода проверяется тестовое покрытие на соответствие определенным условиям.

Результаты заносятся в журнал для последующего анализа. Для организации тестового окружения существуют аннотации, позволяющие выполнить код перед тестами. Последовательность тестов не регламентирована.

**60. Интеграционное тестирование. Стратегии интеграции.**

Интеграционное тестирование – одна из фаз тестирования ПО, при которой отдельные программные модули объединяются и тестируются в группе. Интеграционное тестирование проводится после модульного и предшествует системному. Смысл данного тестирования в проверке взаимодействия модулей на правильную последовательность вызовов и соответствия протоколов взаимодействия требуемой спецификации. Системное интеграционное тестирование проверяет взаимодействие между аппаратным обеспечением или системами и может быть вызвано после системного.

Стратегии:

* Сверху вниз – самая распространенная, применяется для бизнес-приложений. Сначала проверяется бизнес-логика с драйверами и заглушками, потом подключается UI, который выполняет запросы к блоку кода бизнес-логики, последними подключаются блоки хранения данных. Быстро появляется осязаемое приложение, но много заглушек.
* Снизу вверх используется, когда приложение сильно связано с аппаратной архитектурой, поэтому подключаются с нижнего уровня.
* Функциональная (по 1 функции, производится сборка, отладка и тестирование по 1 пользовательскому сценарию).
* Ядро - формируется минимальный работоспособный функционал, а потом добавляются остальные функции.
* Большой взрыв - собираем все. Это крайне рискованный шаг, поскольку если не работает один компонент, то программа будет работать некорректно, либо вообще не запустится.

**61. Функциональное тестирование. Selenium.**

Функциональное тестирование обычно рассматривается как разновидность интеграционного. В функциональном тестирование проверяется функционал, заложенный в программу. Данное тестирование осуществляется на базе сценариев использования, где в явном виде описаны действия пользователя в системе.

Следует отметить, что обычно проверяются бизнес-процессы целиком, которые могут включать использование различных ролей, последовательно выполняющих отдельные элементы бизнес-процесса.

Основным элементом управления при ФТ является графический интерфейс. ФТ могут выполняться как ручными тестировщиками, так и при помощи автоматических средств. Разработка тестируемого ПО при этом полностью завершена.

Для ФТ разработано большое кол-во средств автоматизации, например дополнение к Firefox – Selenium. Оно позволяет записать тестовую последовательность использования интерфейса, и сохранить ее в виде тестовой программы, способной исполняться и в других браузерах.чё

**62. Техники статического тестирования. Статический анализ кода.**

Статическое тестирование – это тестирование, не связанное с запуском наборов тестов, разработанных для ПО. Включает в себя методику по рецензии и инспекции кода без его запуска. На ранних стадиях включает проверку спецификаций и архитектурных принципов и требований.

Способствует раннему нахождению, что экономит время и средства. В отличие от динамического статическое находят причины сбоя, а не сам факт.

Рецензирование может проводиться вручную или с помощью средств, главной составляющей ручного тестирования является исследование и комментирование продукта.

Одной из распространенных техник является рецензия коллегой. Глаза человека замыливаются, и мы можем не замечать очевидных ошибок. Однако коллега может ошибаться или быть субъективным.

Есть несколько техник:

● Технический анализ: проводится анализ под руководством лидера проекта.

● Сквозной контроль: просмотр специальным экспертом, который фиксирует недочеты и дефекты.

● Инспекции: много инспекторов, у каждого из которых есть 2 собственные роли.

Преимущества статического тестирования: средства статического анализа проводят проверку кода на неопределенное поведение (не инициализирована переменная), нарушение алгоритмов, разрушение кроссплатформенности и тд.

**63. Тестирование системы в целом. Системное тестирование. Тестирование производительности.**

Тестирование системы в целом начинается после окончания интеграции. На этом этапе нужно провести проверку заявленных характеристик. Состоит из нескольких частей:

● Системное тестирование. Выполняется внутри организации разработчика.

● Альфа- и Бета-тестирование. Выполняется пользователем под контролем разработчика. Альфа - на окружении разработчика, Бета - в реальном пользовательском окружении.

● Приемочное тестирование. Выполняется пользователем в его окружении без контроля разработчика.

Системное тестирование обычно производится от простых сценариев к сложным.

Первыми тестируются заявленные возможности ПО. Обычно на данном этапе используются те же сценарии, что и в функциональном тестировании, но на корректность, реализации функций тестируется вся система целиком.

Затем проверяется стабильность работы системы в таких ситуациях, как одновременное обращение нескольких пользователей, или несколько запросов от одного пользователя.

Затем проверяется устойчивость системы к сбоям: вводятся заведомо неверные данные и проверяется корректность реакции системы на такие ошибки.

После этого проверяются аспекты системы, связанные с

совместимостью.

Затем проводится испытание корректности работы системы в условиях высокой нагрузки и определяются пределы ее производительности.

Тестирование производительности состоит из всех видов тестов CARAT:

● Capacity (функциональные возможности) - последовательное доведение каждого параметра системы до предела и наблюдение за ее поведением

● Accuracy (точность) - точность математических расчетов с заданной константой погрешности. Система должна обеспечить заданную точность в ограниченный промежуток времени.

● Response time (время отклика) - время ответа системы на запрос пользователя.

● Availability (готовность) - обычно выражается в коэффициенте готовности (MTBF-MTTR)/MTBF, где MTBF - mean time before failure - среднее время до отказа, MTTR - mean time to recover - среднее время до восстановления. Определяет время простоя.

● Throughput (пропускная способность) - сколько запросов система может обработать за единицу времени.

**64. Тестирование системы в целом. Альфа- и бетатестирование.**

Тестирование системы в целом начинается после окончания интеграции. На этом этапе нужно провести проверку заявленных характеристик. Состоит из нескольких частей:

● Системное тестирование. Выполняется внутри организации разработчика.

● Альфа- и Бета-тестирование. Выполняется пользователем под контролем разработчика. Альфа - на окружении разработчика, Бета - в реальном пользовательском окружении.

● Приемочное тестирование. Выполняется пользователем в его окружении без контроля разработчика.

Преимущества Альфа- и Бета-тестирования состоят в том, что разработчики могут получить полезные отзывы для завершения разработки, а еще пользователи могут пойти нестандартными путями, которые не были учтены разработчиками.

**65. Аспекты быстродействия системы. Влияние средств измерения на результаты.**

Скорость выполнения программ зависит от многих факторов

* Аппаратный аспект
  + Частота
  + Кол-во процессоров
  + Количество ядер
  + Объем и скорость кэшей
  + Тип и скорость памяти
* Системный и архитектурный аспект
  + Архитектура (монолитная, микро сервисная, потоковая)
  + Виртуализация, кластеризация
* Программный аспект
  + Использование более эффективный алгоритмов
  + Использование в программе многопоточности
* Человеческий фактор (ошибки)

Несложно понять, что использование средств наблюдения в какой то системе, дополнительно её нагружает, однако это плата за то самое наблюдение, итак если говорить про анализ производительности:

1. Выбираются критерии оценки
2. Выбирается средство измерения

* Не интрузивные
* Слабо интрузивные
* Интрузивные

1. Выбираем нагрузку
2. Проводим анализ результатов
3. Делаем рефакторинг кода
4. Повторяем до удовлетворения

**66. Ключевые характеристики производительности.**

* Время отклика системы (latency) – время от выдачи запроса до получения первых результатов
* Пропускная способность – максимальное количество запросов за единицу времени может пройти через какой то конкретный узел системы
* Утилизация ресурса (%util) – какую долю времени ресурс занят полезной работой
* Точка насыщения – момент, когда нагрузка достигает предельного значения, которое может обработать устройство или программа.
* Масштабируемость – насколько можно количественно нагрузить систему
* Эффективность ПО – Отношение полезной работы к общей

**67. Нисходящий метод поиска узких мест.**

Нисходящий метод поиска узких мест последовательно рассматривает систему, от более общих компонент к более частным.

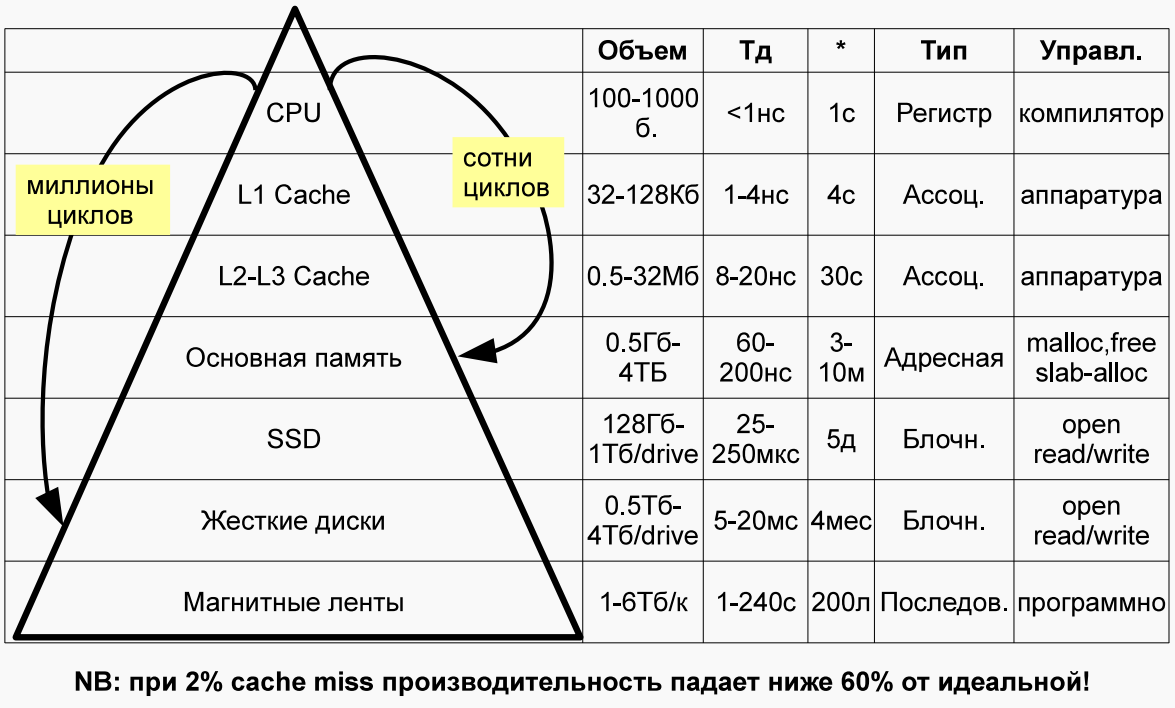
В начале мы должны проверить все системные журналы, файлы конфигурации системы и ПО. Проверить качество аппаратуры, проводов, шин итд. Проверить корректность введенных данных администратора, например как много места он выделил для java машины. Веха – проверка аппаратуры и начальных условий

Проверка ОС. ОС обладает большим количеством средств наблюдения за своими подсистемами. Веха – проверка состояния ОС

Мониторинг приложения – поиск проблем в алгоритмах, api, проблемы связанные с многопоточностью и синхронизацией. Для этого используются средства мониторинга и профилирования приложений.

Мониторинг микроархитектуры – мы можем начать выравнивать размеры данных, оптимизировать кэши. Однако до этого уровня обычно не доходят.

**68. Пирамида памяти и ее влияние на производительность.**



Чем ниже память в пирамиде, тем дороже обращение к ней, например обращение к основной памяти (в середине пирамиды), может занять в сотни раз больше времени, чем обращение к регистрам процессора или кэшу первого уровня. Поэтому минимизация обращений к нижним уровням пирамиды, во много раз повышает производительность.

**69. Мониторинг производительности: процессы.**

При мониторинге процесса он может находится в трех состояниях:

* Готов к использованию (runnable)
* Находится на исполняющем устройстве
* Находится в режиме ожидания из за ввода/вывода, освобождения блокировки итд.

Пусть наш процесс в данный момент находится на процессоре, и работает программа пользователя. Раз в 10 мс происходит часовое прерывание (clock interrupt), во время которого ОС наращивает счетчики производительности, проверяет в очереди процессы с большим приоритетом, а также проверяет не исчерпал ли он свой квант времени. Если наш процесс не исчерпал свой квант времени и имеет высший приоритет, то он продолжает работу. Если нет, то процесс переходит в в режим ожидания и происходит context switch с более приоритетным процессом (мы сохраняем в памяти все регистры старого процесса, а новый выбирается из очереди процессов, готовых к выполнению). Все процессы, которые находятся в режиме ожидания, наращивают свои приоритеты со временем.

**70. Мониторинг производительности: виртуальная память.**

Виртуальная память – метод управления памятью, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, через автоматическое перемещение частей программы между основной памятью и хранилищем.

В работе виртуальной памяти задействованы физическая память, устройство подкачки и виртуальные страницы, которые могут принадлежать либо физической, либо виртуальной памяти. Каждая страница может быть в памяти, на swap-устройстве или быть зарезервированной.

Весь процесс поделен на страницы от 4 кб. Страницы, связанные с памятью, называются именованной памятью, а созданные в динамической куче

- анонимной памятью.

Scan rate – число отсканированных страниц за единицу времени. Если эта величина высокая, значит не хватает оперативной памяти. При недостатке памяти ОС производит последовательное сканирование редко используемых, неизменяемых страниц, и принимает решение о их записи на диск, или удаления.

Если процессор обратился к отсутствующей в памяти странице, то происходит страничный сбой, он может быть минорным – когда для страницы нужно просто создать запись в таблице адресации, или мажорным – когда страница должна быть загружена из диска подкачки.

**71. Мониторинг производительности: буферизированный файловый ввод-вывод.**

При чтении данных вначале указывается количество байт, которые нужно прочитать (IO record size). Интенсивность чтения данных определяется требованиями программы.

Предположим, внутри программы производится обращение к функции fread, которая находится в системной библиотеке. Функция fread формирует запрос к ядру и вызывает соответствующую функцию ядра read. Ввод-вывод в ядре попадает в подсистему VFS(virtual file system, виртуальная файловая система), и запрос попадает на уровень, не связанный с конкретной файловой системой. Имя файла преобразуется в DNLC (directory name lookup cache — кэш, ускоряющий обработку имен файлов) в номер файла inode в необходимой файловой системе, ОС экономит ресурсы при обращении к устройствам

ввода-вывода, и операция read будет работать в первую очередь с буферным кэшем (Buffer cache). В нём данные содержатся в виде блоков данных файла в ОЗУ, формируя промежуточное хранилище. Каждые 30 секунд (время зависит от ОС) данные, которые были помечены как изменённые, сохраняются на диск. Ниже, под виртуальной файловой системой существуют реализации модулей ядра физических файловых систем. К точке монтирования подключается устройство с использованием его специального файла, после этого работу ведёт драйвер, и на уровне аппаратного интерфейса данные перемещаются в дисковое устройство.

Для каждой файловой системы существуют наблюдаемые параметры: количество чтений (r/s), количество записей (w/s), объем читаемых и записываемых данных (rkB/s и wkB/s), средние времена запросов (avgrq-sz и avgqu-sz), время ожидания (await,r\_await,w\_await), время обслуживания (svctm), и процент занятости устройства (%util).

Применительно к вырожденным типам обмена с дисковой подсистемой различают случайный доступ (random IO, каждый запрос обращается к новому месту диска) и последовательный доступ (sequential IO, данные записываются и читаются большими группами последовательно), при этом скорость чтения/записи во втором случае во много раз выше, чем в первом. Для случайного доступа существенно повысить скорость обмена данными может использование твердотельных накопителей (solid-state drive, SSD) из-за отсутствия в них (в отличие от дисков) движущихся частей

**72. Мониторинг производительности: Windows и Linux.**

В Windows самым распространённым является стандартное средство системного мониторинга Диспетчер Задач, где, помимо всего прочего, выводится статистика использования процессора, памяти, дисковой подсистемы, видеокарты, сети и др.

Для более детального исследования поставляются также специальные системные оснастки. Наиболее детальную информацию можно получить из средства Microsoft SysInternals, которое базируется на внутрисистемных счетчиках и информации, вследствие чего показывает наиболее точные результаты по сравнению с другими средствами.

В Linux существует большое число средств мониторинга, наблюдающих за определёнными подсистемами ОС и учитывающих особенности архитектуры этих подсистем. В общем случае они являются не интрузивными.

Утилита top позволяет динамически наблюдать характеристики запущенных процессов, такие, как текущий приоритет (PR), занимаемая память (VIRT — общий размер адресного пространства процесса, RES — размер в физической памяти, SHR— в совместно используемой с другими процессами) и др.

Утилита sar имеет множество опций для вывода той или иной системной информации ядра.

Отдельно следует выделить средство perf, которое может собирать и показывать большое число характеристик не только ядра, но и запущенной под управлением perf программы. Perf умеет работать со счетчиками производительности процессора для сбора таких событий, как промахи мимо кэша.

Для детального наблюдения за процессом можно использовать strace. Эта утилита позволяет проводить трассировку системных вызовов, которые процесс выполняет к ядру ОС и системным библиотекам. Strace в общем случае интрузив.

System tap (stap) позволяет установить точки сбора информации в ядре, собрать и агрегировать информацию о подсистемах ядра.

**73.** **Системный анализ Linux "за 60 секунд".**

Системный анализ Linux "за 60 секунд" – последовательность команд, позволяющая быстро оценить ситуацию и составить первое впечатление о источнике ошибок.

Изображение выглядит как текст

Автоматически созданное описание

**74. Создание тестовой нагрузки и нагрузчики.**

В корпоративных системах наблюдение реальных, критически важных для бизнеса систем часто запрещено из-за страха перед внесением искажений в нормальную работу системы средствами мониторинга, или перед наличием дефектов в этих средствах. В таких случаях обычно создают отдельную тестовую систему, являющуюся полной копией реальной, и проводят измерения на ней.

Для тестовой системы нужно иметь возможность создавать нагрузку, близкую по характеристикам к реальной пользовательской нагрузке. Нагрузку, аналогичную реальной, можно создать с помощью средства создания синтетической нагрузки или средства записи реальной нагрузки, позволяющего запомнить нагрузку на реальном устройстве и использовать эти данные для тестовой системы. При этом синтетическая нагрузка всегда будет отличаться от реальной, и в средствах создания такой нагрузки используется большое число параметров, позволяющих гибко её настроить.

**75. Профилирование приложений. Основные подходы.**

С помощью профилирования можно узнать время исполнения функции, объем созданных объектов, проследить за потоками приложений и борьбой потоков за захват блокировки.

Существует два основных подхода. Согласно первому, диагностические точки можно внедрять в сами функции из указанного набора. Для данного способа для начала следует определить проблемные места.

Второй подход предполагает использование прерываний программы с заданной периодичностью. Профилировщик прерывает программу и собирает интересующую программу. Собирается инфа о состоянии стека и кучи.

Интервал нужно выбирать оптимально.

**76. Компромиссы (trade-offs) в производительности.**

Борьба за производительность сопряжена с компромиссами (trade-offs). Изменения в одном месте (компоненте, методе, алгоритме и т.п.) неизбежно ведут к изменениям в других местах.

Например, чем быстрее мы хотим получить доступ к данным, тем больше нам потребуется на это памяти: последовательный поиск в массиве (linear search) всегда вступает в противоречие в плане скорости/занимаемой памяти с индексированием (indexing), так как для индекса требуется дополнительная память, но его присутствие намного ускоряет поиск.

Таким образом, время, потраченное CPU, связано с тем количеством памяти, которое требует программа: если сделать требования программы скромнее, то она будет требовать больших ресурсов CPU, а программа с большими требованиями к памяти будет работать быстрее за счёт используемых алгоритмов. Однако память тоже является ограниченными ресурсом.

Другим примером компромисса является блочный доступ к диску с кэшированием блоков данных в ядре. Больше кэш — в общем случае, быстрее чтение-запись, но меньше памяти останется для других задач.

Грамотный выбор алгоритма и учет требований архитектуры может ускорить приложение в миллионы раз!

**77. Рецепты повышения производительности при высоком %SYS.**

Высокая загруженность CPU задачами уровня ядра

1) Высокая нагрузка на подсистему ввода-вывода, нужно реже читать/писать, сжимать данные, применять буферизацию или заменить устройства на более быстрые

2) Недостатки в работе планировщика, частая диспетчеризация, нужно проверить не слишком ли много потоков и что делает ОС

3) Избыточная подкачка страниц. Нужно выдать больше памяти системе, меньше процессам. Либо запретить выгрузку важных процессов из памяти.

4) Трата времени процессора в других системных функциях. Нужно найти и исключить лишние функции и настроить параметры ядра.

**78. Рецепты повышения производительности при высоком %IO wait.**

Высокое время ожидания CPU

1) Проблемы,связанные самими приложениями. Следует оптимизировать запросы к диску. Можно ввести объем большими порциями за 1 операцию.

2) Буфера/Кэши, нужно расширить память для промежуточного использованияинастроить кэши.

3) Проблемы с аппаратурой, нужно купить более совершенную дисковую систему. Эффективны ССД-накопители

**79. Рецепты повышения производительности при высоком %Idle.**

При высоком времени простоя может быть мало процессов на стадии выполнения. В этом случае нужно распараллелить алгоритмы в приложении. Проанализировать блокировки, например, когда много потоков пытаются завладеть одним и тем же участком кода. Нужно держать блокировку как можно меньше и делать ее легкой. Добавить потоки в пулы приложений. Проблемы могут корениться в самой ОС, это связано с дефектами в ОС на системных блокировках. Нужно промониторить их. Также может помочь настройка подсистем ядра.

**80. Рецепты повышения производительности при высоком %User.**

При высокой загруженности процессора нужно воспользоваться средствами профилирования, они помогут найти самые затратные функции.

Рецепты:

1. Использовать алгоритмы меньшей сложности

2. использовать объекты повторно, так в линуксе процесс сначала попадает в состояние зомби и может восстать.

3. На уровне микроархитектуры важно избавиться от кэш-промахов и промахов мимо TLB, Кэш промахи можно исправить работой над структурой данных, а мио тлб можно исправить использованием больших страниц памяти.