

Ответы на экзамен по МСПИ (2021)

1. ISO/IEC 12207:2010: Жизненный цикл ПО. Группы процессов ЖЦ.	4
2. Модели ЖЦ (последовательная, инкрементная, эволюционная).	5
3. Водопадная (каскадная) модель.	6
4. Методология Ройса.	7
5. Традиционная V-chart model J.Munson, B.Boehm.	8
6. Многопроходная модель (Incremental model).	9
7. Модель прототипирования (80-е).	10
8. RAD методология.	11
9. Спиральная модель.	12
10. UML Диаграммы: Структурные и поведенческие.	13
11. UML: Use-case модель.	14
12. UML: Диаграмма классов.	15
13. UML: Диаграмма последовательностей	16
14. UML: Диаграмма размещения	17
15. *UP методологии (90-е). RUP: основы процесса.	18
16. RUP: Фаза «Начало».	19
17. RUP: Фаза «Проектирование».	20
18. RUP: Фаза «Построение».	21
19. RUP: Фаза «Внедрение».	22
20. Манифест Agile (2001).	23
21. Scrum.	24
22. Disciplined Agile 2.X (2013).	25
23. Требования. Иерархия требований.	26
24. Свойства и типы требований (FURPS+).	27
25. Формулирование требований. Функциональные требования.	28
26. Требования к удобству использования и надежности.	29
27. Требования к производительности и поддерживаемости.	30
28. Атрибуты требований.	31
29. Описание прецедента.	32
30. Риски. Типы Рисков.	33

31. Управления рисками. Деятельности, связанные с оценкой.	34
32. Управления рисками. Деятельности, связанные контролем и управлением.	35
33. Изменение. Общая модель управления изменениями.	36
34. Системы контроля версий. Одновременная модификация файлов.	37
35. Subversion. Архитектура системы и репозиторий.	38
36. Subversion: Основной цикл разработчика. Команды.	39
37. Subversion: Конфликты. Слияние изменений.	40
38. GIT: Архитектура и команды.	41
39. GIT: Организация ветвей репозитория.	42
40. GIT: Плагин git-flow.	43
41. Системы автоматической сборки: предпосылки появления	44
42. Системы сборки: Make и Makefile.	45
43. Системы сборки: Ant. Команды Ant.	46
44. Системы сборки: Ant-ivy.	47
45. Системы сборки: Maven. POM. Репозитории и зависимости.	48
46. Maven: Структура проекта. GAV.	49
47. Maven: Зависимости. Жизненный цикл сборки. Плагины.	50
48. Системы сборки: Maven. POM. Репозитории и зависимости.	51
49. Системы сборки: GNU autotools. Создание конфигурации проекта.	52
50. Системы сборки: GNU autotools. Конфигурация и сборка проекта.	53
51. Сервера сборки/непрерывной интеграции.	54
52. Основные понятия тестирования. Цели тестирования.	55
53. Понятие полного тестового покрытия и его достижимости. Пример.	56
54. Статическое и динамическое тестирование.	57
55. Автоматизация тестов и ручное тестирование.	58
56. Источники данных для тестирования. Роли и деятельности в тестировании.	59
57. Понятие тестового случая и сценария.	60
58. Выбор тестового покрытия и количества тестов. Анализ эквивалентности.	61
59. Модульное тестирование. Junit 4.	62
60. Интеграционное тестирование. Стратегии интеграции.	63

61. Функциональное тестирование. Selenium.	64
62. Техники статического тестирования. Статический анализ кода.	65
63. Тестирование системы в целом. Системное тестирование. Тестирование производительности.	66
64. Тестирование системы в целом. Альфа- и бета-тестирование.	67
65. Аспекты быстродействия системы. Влияние средств измерения на результаты.	68
66. Ключевые характеристики производительности.	69
67. Нисходящий метод поиска узких мест.	70
68. Пирамида памяти и ее влияние на производительность.	71
69. Мониторинг производительности: процессы.	72
70. Мониторинг производительности: виртуальная память.	74
71. Мониторинг производительности: буферизированный файловый ввод-вывод.	75
72. Мониторинг производительности: Windows и Linux.	76
73. Системный анализ Linux "за 60 секунд".	77
74. Создание тестовой нагрузки и нагрузчики.	78
75. Профилирование приложений. Основные подходы.	79
76. Компромиссы (trade-offs) в производительности.	80
77. Рецепты повышения производительности при высоком %SYS.	81
78. Рецепты повышения производительности при высоком %IO wait.	82
79. Рецепты повышения производительности при высоком %Idle.	83
80. Рецепты повышения производительности при высоком %User.	84

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

Жизненный цикл – время существования программы от момента замысла, до вывода ее из эксплуатации. Определение из ISO: Развитие системы, продукта, услуги, проекта или других изготовленных человеком объектов, начиная со стадии разработки концепции и заканчивая прекращением применения.

Все этапы ЖЦ описаны в ISO. Основные группы:

- Согласование (2 процесса)
- Организация обеспечения (5 процессов)
- Проектирование (7 процессов)
- Технические процессы (11 процессов)
- Реализация (7 процессов)
- Поддержка (8 процессов)
- Повторное использование (3 процесса)

Разработка программного обеспечения начинается с определения и согласования требований. Требования заказчика поступают к аналитикам, которые предлагают способы решения поставленной задачи. После анализа происходит проектирование архитектуры и шаблонов реализации ПО, а затем проектное задание передается разработчикам. формируются подходы и производится тестирование, и, после его успешного завершения, программный продукт внедряется и эксплуатируется. В процессе эксплуатации осуществляется поддержка пользователей, исправление дефектов и проблем ПО и обновление продукта. В конце срока эксплуатации производятся процедуры и работы, связанные с выводом ПО из использования.

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

Модель жизненного цикла - структура, определяющая последовательность выполнения и взаимосвязи этапов, выполняемых на протяжении жизненного цикла.

Каскадная модель: последовательный переход на следующий этап после завершения предыдущего. Для этой модели характерна автоматизация отдельных несвязанных задач.

Инкрементная модель: продукт разбивается на несколько частей на основе номенклатуры функциональных требований, причем части должны быть примерно одинаковы с точки зрения архитектуры.

Эволюционная модель: разрабатывается прототип, архитектурно и функционально дорабатываемый со временем

В реальности в чистом виде инкрементных или эволюционных проектов встречается мало, в основном применяют инкрементно-эволюционную модель.

3. Водопадная (каскадная) модель.

Каскадная модель: последовательный переход на следующий этап после завершения предыдущего. Для этой модели характерна автоматизация отдельных несвязанных задач.

Достоинство: хорошие показатели по срокам разработки и надежности при решении отдельных задач.

Недостаток: неприменимость к большим и сложным проектам из-за невозможности применять изменения требований к системе в течение длительного проектирования.

Стандартная последовательность шагов в каскадной модели такова:

1. Определяются системные требования.
2. Определяются требования к ПО.
3. Требования анализируются.
4. Проектируется программа.
5. Разрабатывается код.
6. Проводится тестирование.
7. ПО вводится в эксплуатацию.

4. Методология Ройса.

Первый шаг: дизайн программы. В нем дизайнеру предлагается спроектировать, определить и создать модели обработки данных и разработать документ: обзор будущей программы.

Второй шаг: документирование дизайна, требования к системе, спецификация дизайна, план тестирования, инструкция по использованию.

Третий шаг: “do it twice”. Тестовая разработка параллельно основному процессу и использование в качестве пилота для подтверждения или опровержения основных спецификаций ПО.

Четвертый шаг: планирование, контроль и мониторинг тестирования. Проведение визуальной инспекции другим лицом для выявления визуально заметных дефектов. Тестирование каждого логического пути внутри программы.

Пятый шаг: подключение пользователя на ранних этапах, чтобы получить предварительный, критический и финальный просмотр.

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

В основе V-модели лежит та же последовательность шагов, что и в каскадной модели, но каждому уровню разработки соответствует свой уровень тестирования, направлена на тщательную проверку и тестирование продукта.

Модульное, интеграционное и системное тестирование проводятся последовательно на основании критериев на основании критериев верификации. Последним этапом является приемочное тестирование. Статическое тестирование может выполняться на ранней стадии разработки.



6. Многопроходная модель (Incremental model).

В инкрементной модели полные требования к системе делятся на различные сборки.

Цикл разделен на более мелкие легко создаваемые модули. Каждый модуль проходит через фазы определения требований, проектирования, кодирования, внедрения и тестирования.

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». Процесс продолжается до тех пор, пока не будет создана полная система.

Заказчик может наблюдать за разработкой, вносить изменения, которые несильно увеличат стоимость. Недостаток заключается в устаревании архитектуры системы.



7. Модель прототипирования (80-е).

Модель прототипирования позволяет создать прототип программного продукта до или в течение этапа составления требований к программному продукту. Потенциальные пользователи работают с этим прототипом, определяя его сильные и слабые стороны, о результатах сообщают разработчикам программного продукта. Если прототип не подходит, то разработка начинается заново. Таким образом, обеспечивается обратная связь между пользователями и разработчиками, которая используется для изменения или корректировки спецификации требований к программному продукту. В результате такой работы продукт будет отражать реальные потребности пользователей.



8. RAD методология.

В настоящее время одной из наиболее распространенных методологий является RAD - Rapid Application Development.

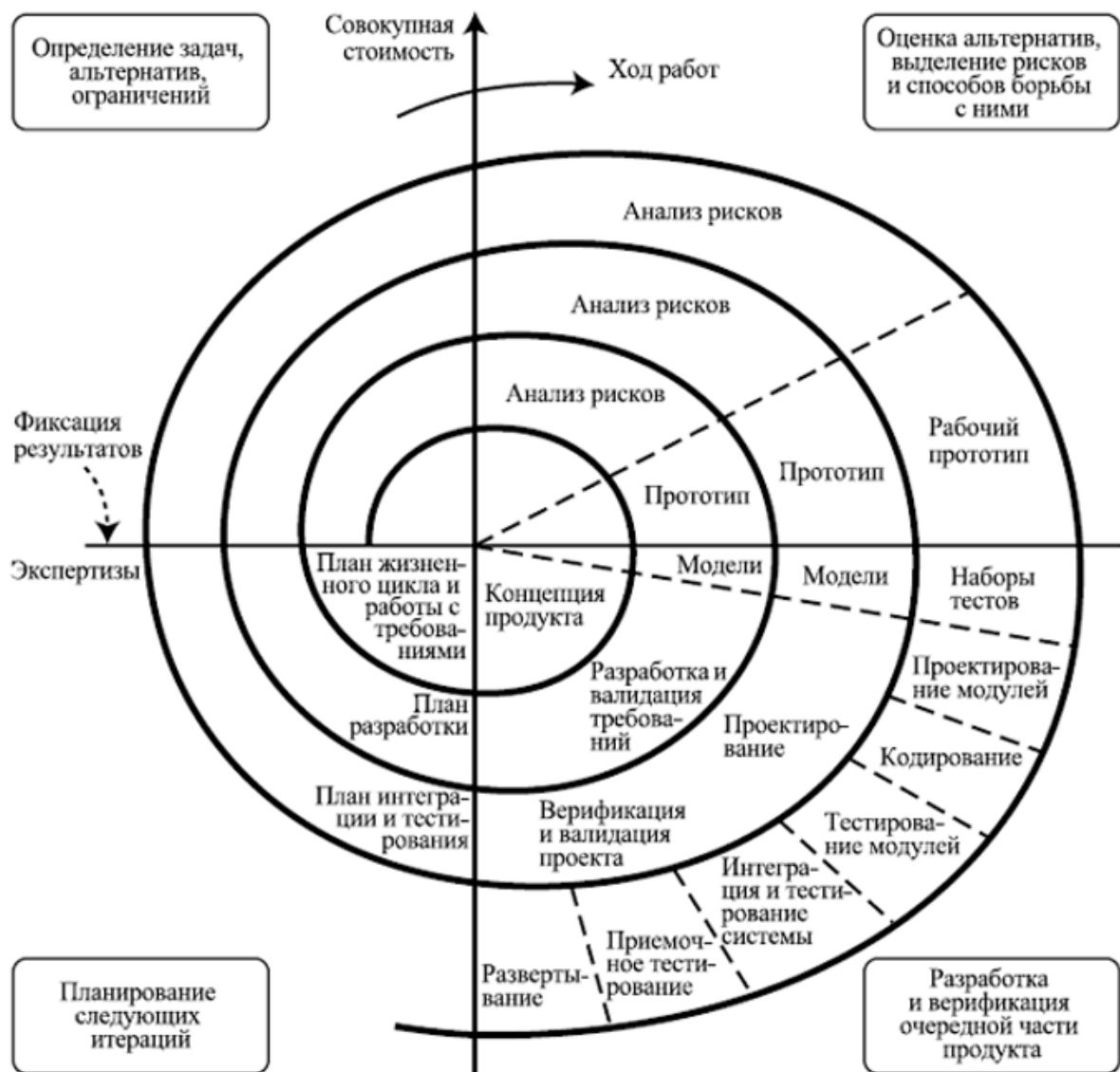
RAD - разновидность инкрементной модели. Пользователь принимает непосредственное участие в процессе разработки. При помощи интерфейса пользователь способен создавать простейшие функции. В RAD-модели компоненты или функции разрабатываются несколькими командами параллельно, как несколько мини-проектов.

Временные рамки одного цикла жестко ограничены. Созданные модули затем интегрируются в один рабочий прототип. Синергия позволяет очень быстро предоставить клиенту для обозрения что-то рабочее с целью получения обратной связи и внесения изменений.



9. Спиральная модель.

Спиральная модель представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и поэтапное прототипирование. Данный подход может оказаться довольно затратным в применении. Именно поэтому он не очень хорошо подходит для небольших проектов. В спиральной модели особое внимание уделяется управлению рисками. Контроль рисков, в свою очередь, требует проведения специфического анализа на каждой итерации. Изменения – неотъемлемая часть разработки.



10. UML Диаграммы: Структурные и поведенческие.

Основное назначение UML - графическое представление различных аспектов разработки программного обеспечения. В UML существуют *структурные* и *поведенческие* диаграммы.

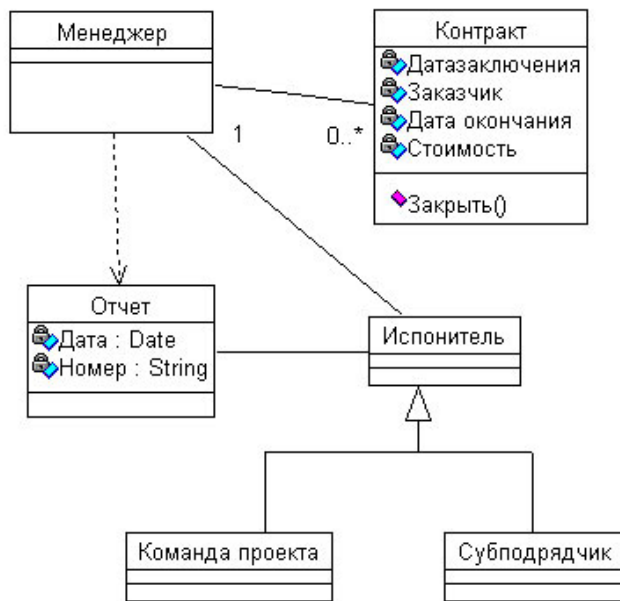
Структурные диаграммы: Эти диаграммы используются для демонстрации статической структуры элементов в системе. Они могут изображать архитектурную организацию системы, ее физические элементы, текущую конфигурацию, а также специфические элементы предметной области.

Поведенческие диаграммы: Носят статический характер. Однако события, происходящие в системах программного обеспечения, являются динамическими: объекты создаются и уничтожаются, объекты передают сообщения другим объектам и системам, внешние события активизируют операции над определенными объектами.

11. UML: Use-case модель.

Use-case-диаграмма - это диаграмма динамического поведения в UML, которая моделирует функциональность системы с использованием участников, прецедентов и других важнейших объектов. «Актеры» - это люди или организации, которые работают под определенными ролями внутри системы. Отношение включения указывает на то, что поведение одного прецедента включается в некоторой точке в другой прецедент в качестве составного компонента. Отношение расширения отражает возможное присоединение одного варианта использования к другому в некоторой точке.

12. UML: Диаграмма классов.



Основным преимуществом диаграмм классов является наглядное изображение предметной области. Эти диаграммы по мере дальнейшей работы на стадиях проектирования могут быть уточнены и расширены. Например, могут быть добавлены типы данных и область их видимости, кратность и роли ассоциаций и т. д.

Существует четыре типа связей в UML:

Зависимость – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемент.

Ассоциация – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Например, класс Человек и класс Школа имеют ассоциацию, так как человек может учиться в школе.

Обобщение - выражает специализацию или наследование, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру родителя.

Реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться.

13. UML: Диаграмма последовательностей

Диаграммы последовательностей используются для уточнения диаграмм прецедентов, более детального описания логики сценариев использования.

Диаграммы последовательностей 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 показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами.

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, связанных, например, с обеспечением безопасности.

Это единственная диаграмма, на которой применяются трехмерные обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры.

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

RUP (Rational Unified Process) - это универсальная методология распределения задач и сфер ответственности. Процесс основан на инкрементно-эволюционной методологии. Цель – создание высококачественного программного обеспечения.

RUP оптимизирует командную работу - обеспечивает команде разработчиков свободный доступ к базе знаний с инструкциями для использования программных средств. RUP ориентирован на создание и поддержание моделей. UML позволяет команде легко донести свои требования к проекту, его архитектуру и план реализации.

В основе RUP лежит шесть главных принципов:

1. Итеративная модель разработки - устранение рисков на каждой стадии проекта позволяет лучше понять проблему и вносить необходимые изменения, пока не будет найдено приемлемое решение
2. Управление требованиями - RUP описывает процесс организации и отслеживания функциональных требований, документации и выбора оптимальных решений (как в процессе разработки, так и при ведении бизнеса)
3. Компонентная архитектура - архитектура системы разбивается на компоненты, которые можно использовать как в текущем, так и в будущих проектах
4. Визуальное моделирование ПО - RUP методология разработки показывает, как создать визуальную модель программного обеспечения, чтобы понять структуру и поведение архитектуры и его компонентов
5. Проверка качества ПО - в процессе разработки программного обеспечения контролируется качество всех действий команды
6. Контроль внесённых изменений - отслеживание изменений позволяет выстроить непрерывный процесс разработки. Создается благоприятная обстановка, в рамках которой команда будет защищена от изменений в рабочем процессе.

Фазы: Начало, Проектирование, Построение, Внедрение

В конце каждой фазы существует отметка завершения этапа (Project Milestone) — момент, когда ваша команда оценивает, достигнуты ли поставленные цели. При этом команда принимает важные решения, влияющие на ход следующей фазы.

16. RUP: Фаза «Начало».

Команда определяет структуру и основную идею проекта. Предлагают технические решения. Команда решает, стоит ли вообще заниматься этим проектом, исходя из его предполагаемой стоимости, необходимых ресурсов и цели, которую нужно достичь.

Веха “Lifecycle Objects”: наступает, когда все заинтересованные стороны достигают согласия в оценке сроков, стоимости, требованиях, используемых технологиях, приоритетах, а также в оценке рисков и выборе стратегии их преодоления и смягчения последствий.

17. RUP: Фаза «Проектирование».

Цель этой фазы — анализ требований к системе и ее архитектуры, разработка плана проекта и устранение элементов наивысшего риска. Создаются прототипы. Это самая важная фаза из всех, она знаменует переход от низкого уровня риска к высокому. Уточняются сроки и стоимость системы.

Веха “Lifecycle Architecture”: проверяется соответствие проекта концепции и требованиям, стабильность архитектуры, формирование критериев тестирования, отсутствие основных технологических рисков по итогам тестирования прототипов, приемлем ли проект заказчику по цене, исполним ли он (соответствуют ли запланированные ресурсы затраченным).

18. RUP: Фаза «Построение».

В этой фазе RUP методологии команда начинает разработку всех компонентов и функций программного обеспечения, интегрирует их в конечный продукт. Это производственный процесс, в рамках которого команда сосредоточена на управлении ресурсами, чтобы оптимизировать расходы, время и качество продукта. Проводятся плановые демонстрации. В конце происходит подготовка к передаче заказчику.

Веха “Initial Operational Capability”: проверяется стабильность продукта перед передачей пользователю, готовность сторон к передаче, приемлемость соотношения запланированных и затраченных ресурсов.

19. RUP: Фаза «Внедрение».

Фаза, когда продукт готов и доставлен покупателям. Но после того как пользователи получают продукт, могут возникнуть новые трудности. Команде нужно будет исправить ошибки, отловить баги и доделать функции, которые не были реализованы в первично установленный срок. Происходит оценка удовлетворенности пользователя.

Веха "Product Release": проверяется удовлетворение пользователей и соотношение запланированных и затраченных ресурсов.

20. Манифест Agile (2001).

Во главу угла в Agile-подходе ставятся требования заказчика, и, следовательно, реакция на них. Данный подход невозможен, или практически невозможен, если на проект выделяется фиксированный бюджет, без возможности его расширения.

Agile-методологии хорошо работают для внутренних проектов разработки в компаниях, которые занимаются различными бизнесом, или когда заказчик непосредственно покупает рабочие часы разработчиков. Разработчики, со своей стороны, должны постоянно демонстрировать результат работ, этот результат оценивается, разработчики получают деньги, и немедленно начинают разрабатывать новый функционал. При этом важно открыто сотрудничать с заказчиком, принимать во внимание его замечания.

Еще одним важным принципом является максимальное сокращение расходов на труд разработчиков, не относящийся к созданию кода. В небольших проектах с типовой архитектурой это реализуется просто, а в более сложных порождает проблемы, так как без моделей и документирования архитектуры невозможно создать проект, выходящий за рамки типового.

Основные принципы:

- Наивысшим приоритет - удовлетворение потребностей заказчика.
- Изменение требований приветствуется, даже на поздних стадиях разработки.
- Работающий продукт следует выпускать как можно чаще.
- На протяжении всего проекта разработчики и заказчик должны ежедневно работать вместе.
- Над проектом должны работать мотивированные профессионалы.
- 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.
- Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта. Простота 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.

В Scrum процесс разработки сильно упрощен. 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. Другим артефактом является *инкремент продукта*.

Спринт - время от двух до четырех недель, за которое разработчики реализуют выбранный набор требований из бэклога спринта. Каждый спринт заканчивается демонстрацией результатов заказчику. Также через несколько спринтов проводятся ретроспективы, в рамках которых может проводиться работа над ошибками и перераспределение обязанностей для более эффективного использования каждого участника команды.

Команда в Scrum небольшая, от 3 до 10 человек. Также выделяется особая роль - Product Owner - это человек, определяющий порядок разработки требований из бэклога (и, в случае, если он является бизнес-заказчиком, зачастую сам их формулирующий). Для каждого спринта задачи конкретизируются и передаются на разработку команде. 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. Предпринимаются попытки масштабировать Scrum на большие команды, (например, Scrum-of-Scrum), которые пока не показали существенных успехов.

22. Disciplined Agile 2.X (2013).

В Disciplined Agile 2.X (DAD) подход к делению на фазы и дисциплины в DAD 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, DAD 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.

Требование - условия или возможности, которым должна соответствовать система. Требования должны быть сформулированы однозначно, а также грамотно и четко описывать то, что система должна выполнять. Могут быть общими или подробно описывать, что должно быть реализовано, но при этом не описывают, каким образом с точки зрения архитектуры. Для этого есть документы об архитектуре системы. Требования описываются с помощью шаблона SRS (Software Requirement Specification) или Use-Case Model.



24. Свойства и типы требований (FURPS+).

Требование должно быть однозначным и полным, требования не должны противоречить друг другу. Требования должны иметь приоритет. Приоритет зависит от степени критичности функций для бизнеса. Требование должно ссылаться на источник. Модификация требований должна быть контролируемой, в любой момент времени необходимо знать историю изменений.

В RUP для описания и разработки требований используется модель FURPS+ (Functional, Usability, Reliability, Performance, Supportability).

Требования делятся на два типа: функциональные - определяющие, что делает система и нефункциональные - определяющие ограничения и характеристики. Включают в себя:

- требования к пользовательским характеристикам
- требования к надежности
- производительности
- условиям поддержки, например
- обновление без остановки функционала

Кроме того, могут быть ограничения на те или иные программные интерфейсы, требования к реализации, физические требования (параметры импульсов)

25. Формулирование требований. Функциональные требования.

<ID><Система>должна<требование>, где ID - уникальный номер требования - пример формулирования требования. Ключевое слово - должна, оно описывает безусловную необходимость.

Функциональные требования определяют, что система должна делать, и включают в себя:

- Feature set - набор свойств продукта, необходимый для выполнения конкретной деятельности (обычно, здесь подразумеваются крупные функции)
- Capacity - возможности ПО.
- Security - требования к безопасности.

26. Требования к удобству использования и надежности.

В Usability обычно входят особенности использования, такие, как:

- Человеческий фактор, то есть учет физических особенностей человека.
- Согласованность интерфейса пользователя
- Требования к документации и требования к учебным материалам.
- Требования к надежности предназначены для фиксирования способности ПО безотказно работать.
- В требованиях указывают допустимое число отказов и сбоев за определенный промежуток времени.
- Recoverability – способность к восстановлению, точность вычислений и предсказуемость поведения и среднее время между отказами.

27. Требования к производительности и поддерживаемости.

Требования к производительности включают в себя:

- Скорость решения задач.
- Эффективность.
- Время готовности системы к решению задач.
- Пропускную способность системы.

Поддерживаемость включает в себя:

- Расширяемость системы
- Адаптируемость под конкретные задачи
- Совместимость
- Возможность проведения профилактики
- Локализуемость
- Требования к установке на разные системы.

28. Атрибуты требований.

Есть приоритезация требований по MoSCow, требования делятся на:

- Must have - фундаментальные, обязательные.
- Should have - важные, которые следует реализовать при наличии времени. Could have – потенциально возможные.
- Won't have - могут быть добавлены в будущем, но текущие временные рамки не позволяют их сделать.

Кроме того, допускается оценка приоритетов их по 10-ти балльной шкале.

У требований могут быть разные атрибуты: статус (Предложенные / Одобренные / Отклоненные / Включенные), трудоемкость (Человеко-часы, Функциональные точки, use-case points), риск, стабильность (Высокая / Средняя / Низкая) и целевая версия (версия в которую планируют включить реализацию данного требования).

29. Описание прецедента.

Главным элементом модели является действующее лицо или эктор - пользователь, который непосредственно взаимодействует с самой системой. Под пользователем указывается его роль.

Внутри прецедента использования пишется глагол или глагольная фраза, которая указывает на то, что именно должна сделать система (пример: "Пользователь вводит логин и пароль"). Прецеденты одних модулей программы могут служить экторами для других модулей.

Кроме того, описываются предусловия прецедента, то есть условия, при которых этот прецедент может произойти.

Прецедент: PieSelling
ID: 2
Краткое описание: Бабушка продаёт пирожки.
Главные актёры: Бабушка-продавец, Клиент (или любой другой пользователь).
Второстепенные актёры: нет.
Предусловия: Клиент знает, какой пирожок он предпочитает. Бабушка проверила весь ассортимент. У клиента достаточно средств.
Основной поток: 1. Клиент обращается к Бабушке за пирожками. 1.1. Клиент называет количество и номенклатуру пирожков. 1.2. ALT1 PieRecomendation. 1.3. Бабушка подтверждает номенклатуру и называет общую стоимость.

30. Риски. Типы Рисков.

Риск - потенциально опасный фактор, сочетание вероятности события и его последствий на разработку системы. Типы:

- Прямые (можно управлять) и непрямые (возникают из-за внешних причин, поэтому ими невозможно управлять).
- Ресурсные связаны с недостатком ресурсов (денег, людей времени).
- Бизнес-риски. Например, конкуренция двух компаний
- Технические риски. Находятся в пределах компетенции разработчиков и поэтому являются управляемыми.
- Политические риски. Связаны с изменением влияния в компании-заказчике. Например, смена директора.
- Форс-мажоры.

31. Управления рисками. Деятельности, связанные с оценкой.

В процессе оценки (assessment) риска специалист по работе с риском проводит первичное знакомство 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. Таким образом, внутри оценки риска производится его идентификация, анализ и назначение ему приоритета.

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).

Каждый из классов состоит из элементов, а элемент из атрибутов, которые указывают на возможные источники возникновения рисков.

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.

После расчета рисков создается список, отсортированный по величине экспозиции, и на основе этого списка создается документ «ТОП-10 рисков». Данный документ необходимо постоянно обновлять по мере продвижения разработки.

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

После того, как риск всесторонне оценен, его можно поместить под управление различных систем контроля риска, которые широко представлены на рынке. В случае, если система контроля определит его наступление или повышенную вероятность наступления, будет возможно предпринять корректирующие действия.

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.

Во время разработки риски необходимо непрерывно переоценивать. Менеджер проекта, к примеру, раз в неделю может анализировать существующий список рисков, формально проверять вероятность их наступления и менять список наиболее опасных.

Полный список рисков может быть достаточно большим, и поэтому наиболее часто должны контролироваться 10 наиболее вероятных и деструктивных рисков. Данный список должен постоянно меняться по результатам недельного анализа всех рисков.

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.

Невозможно менять систему одновременно в нескольких компонентах. Поэтому изменения выстраиваются в последовательность, которую необходимо контролировать.

Модель: заказчик вносит запрос на изменение системы, создается документ или материальные артефакты (требования, отчет о проблеме). На каждый запрос формируются записи в change log entry. После этого менеджер проекта менеджер определяет техническую необходимость и преимущества изменений. Потом процесс поступает в комитет по изменениям, который анализирует и меняет статус. После этого, если запрос подтвержден производится анализ и реализация изменений.

34. Системы контроля версий. Одновременная модификация файлов.

Системы для управления изменениями в программном коде и поддерживают групповую работу нескольких человек над кодом одновременно, а также контроль за изменениями файлов. 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.

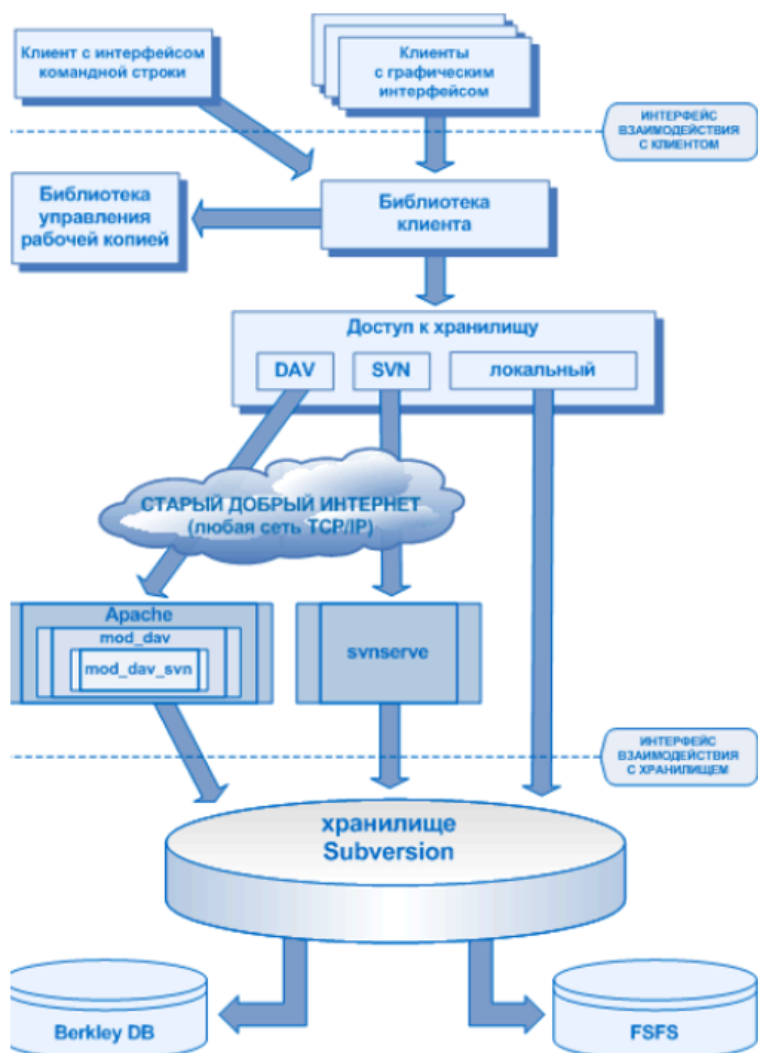
Удаленный доступ может осуществляться по нескольким протоколам svn+https и ssh+svn.

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) и tag (функционально целостные изменения).

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.

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 репозитория.

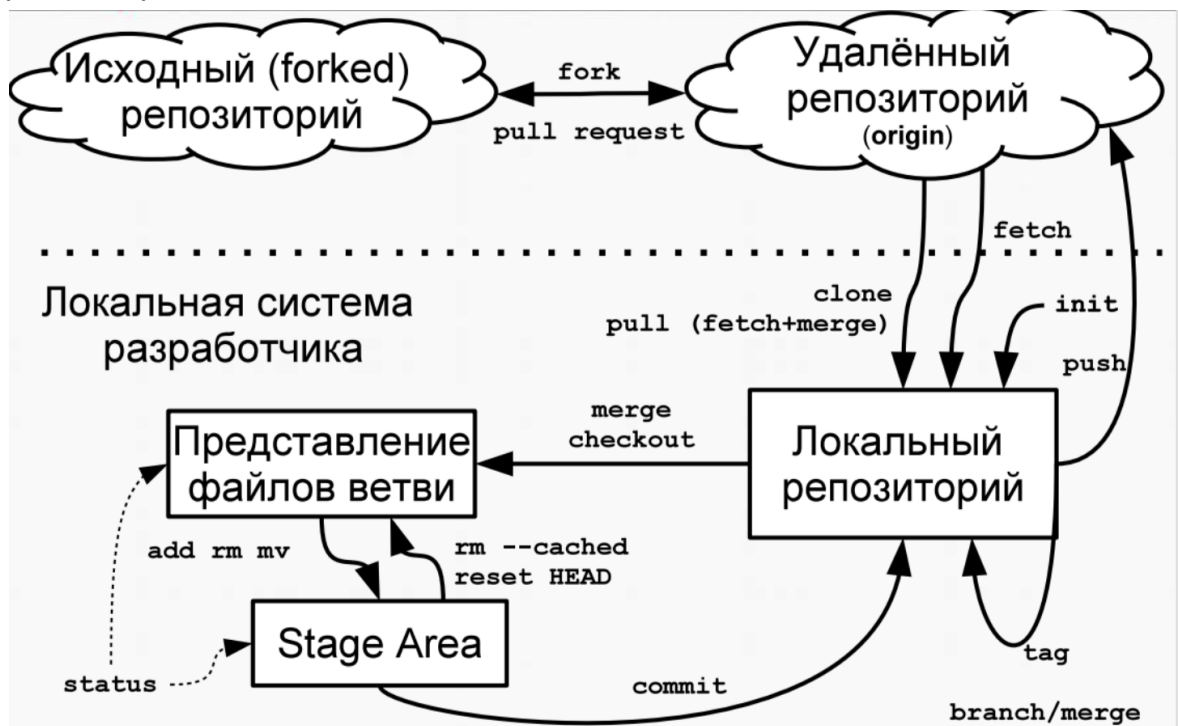
При работе над параллельными ветками часто необходимо изменения из одной ветки сливать с изменениями в другой, образуя одну общую ветвь. Для этого предназначена команда `svn merge`. То есть мы берем из нашей ветки и все изменения, что мы проводили с ней, и присваиваем их другой, второй. Процесс очень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

Автоматизация сборки — этап процесса разработки программного обеспечения, заключающийся в автоматизации широкого спектра задач, решаемых программистами в их повседневной деятельности.

Плюсы:

- Улучшение качества продукта
- Ускорение процесса компиляции и линковки
- Избавление от излишних действий
- Минимизация «плохих (некорректных) сборок
- Избавление от привязки к конкретному человеку
- Ведение истории сборок и релизов для разбора выпусков
- Экономия времени и денег

Существуют императивные(Ant) и декларативные(Maven) системы сборки

Ant'y, сообщают что делать - "скомпилировать эти файлы, а затем скопировать их в эту папку. Затем возьмите содержимое этой папки и создать архив."

Maven рот объявляет то, что мы хотели бы иметь в результате - "вот имена библиотек, от которых зависит проект, и мы хотели бы создать веб-архив", Maven знает, как получить библиотеки и найти исходные классы.

42. Системы сборки: Make и Makefile.

Make является мощной системой для автоматизации процесса сборки программного обеспечения. GNU Make является стандартной реализацией Make. Ее основной задачей является автоматическое выявление модифицированных файлов исходного кода сложных приложений и исполнение команд, направленных на их повторную компиляцию.

Для конфигурации make используются Make-файлы, которые позволяют сохранить группы команд для их последующего исполнения. Make-файлы могут управлять процессом компиляции программного обеспечения благодаря наличию зависимостей, целей и правил. Правила сообщают GNU Make о том, когда, почему и как нужно исполнять заданные последовательности команд для генерации результирующих файлов на основе файлов исходного кода. Целями являются файлы, которые должны генерироваться с участием GNU Make, причем их имена располагаются слева от символов двоеточий в описаниях правил. Чаще всего каждое из правил имеет по одной цели; однако, в рамках одного правила допускается использование сразу нескольких целей. Зависимости располагаются справа от символов двоеточий в описаниях правил и указывают на то, какие файлы или другие цели могут инициировать исполнение команд, описанных в рамках правила, по причине модификаций.

43. Системы сборки: Ant. Команды Ant.

Императивная система сборки, управляется файлом сборки build.xml, в котором в формате XML описан проект. XML удобен для машинной разработки. Описание состоит из целей (target) с явным указанием зависимостей, для каждой цели определены действия, которые необходимо выполнить в процессе сборки. Цели можно вызвать в явном виде. В Ant есть свойства с информацией о проекте, которые необходимо менять для разных условий. Они могут задавать напрямую или через файл. Команды:

tasks (задачи) - соответствуют инструкциям командной строки, например javac, war и т.д. Группа задач может выполняться последовательно с помощью указания целей.

- clean - удалить результаты предыдущих развертываний приложения, если они были
- init - создать необходимую структуру для развертывания
- compile - скомпилировать ваши сервлеты или POJO
- copy - скопировать скомпилированные файлы и web-содержимое в структуру развертывания, созданную модулем init
- war - создать war-файл и открыть браузер

44. Системы сборки: Ant-ivy.

Ivy — это менеджер зависимостей для Ant, который серьезно упрощает работу с продуктом. До этого в Ant зависимости приходилось собирать вручную, этот процесс был длительным и включал в себя много процедур.

Ivy решает проблему скачивания и использования сторонних библиотек за пользователя и способен, в частности, работать с репозиториями Maven2. Так же его легко добавить в build.xml (Смотри пример 1). После этого он может обращаться к удаленному репозиторию и скачивать отсутствующие зависимости.

Зависимости задаются в отдельном файле, который по умолчанию называется ivy.xml (Смотри пример 2).

Пример 1:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant ..." >
  <target name="resolve">
    <ivy:retrieve>
  </target>
...

```

Пример 2:

```
<dependencies>
  <dependency org="javax.servlet" name="servlet-api" rev="2.5" />
</dependencies>

```

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

Maven - одна из самых распространенных систем декларативной сборки. Большинство действий выполняется прозрачно для разработчика и вызывается через интерфейс верхнего уровня. У пользователя остается возможность изменения выполняемых команд сборки.

Maven управляется на базе описания проекта, которое основано на POM - Project Object Model. POM состоит из нескольких частей на основе XML. В нем указаны имя, версия, местонахождение исходного кода, плагины, альтернативные профили.

Существует три типа репозиториев Maven:

Локальный репозиторий - это директория, которая хранится на нашем компьютере. Она создается в момент первого выполнения любой команды Maven и хранит все зависимости проекта (библиотеки, плагины и т.д.). Когда мы выполняем сборку проекта, то JAR-файлы всех зависимостей автоматически загружаются в локальный репозиторий. Это помогает избежать использования ссылок на удаленный репозиторий при каждой сборке проекта.

Центральный репозиторий - это репозиторий, который обеспечивается сообществом Maven. Он содержит огромное количество часто используемых библиотек. Если Maven не может найти зависимости в локальном репозитории, то автоматически начинается поиск необходимых файлов в центральном репозитории.

Удаленный репозиторий - репозиторий, определяемый самим разработчиком. Там могут храниться все необходимые зависимости. Это нужно на случай, если Maven не сможет найти необходимые зависимости и в центральном репозитории. Указывается в pom.xml.

Зависимости в Maven описываются при помощи GAV-синтаксиса - GroupId:ArtifactID:VersionID.

У зависимостей существует т.н. scope, или область действия (*compile, provided, test*). Данный параметр указывает, в какой момент жизненного цикла приложения применяется данная зависимость.

46. Maven: Структура проекта. GAV.

В Maven, в отличие от Ant, существует система каталогов проекта по умолчанию:

- *target* - целевая директория компиляции.
- в *src/main* лежат исходные файлы.
- в файле *properties* отдельно выделяются зависимые от языка пользователя ресурсы.
- в папке *test* лежат тесты.

Модули, внешние ресурсы, зависимости в Maven описываются при помощи GAV-синтаксиса - GroupId:ArtifactID:VersionID.

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

Зависимости в Maven описываются при помощи GAV-синтаксиса - GroupId:ArtifactID:VersionID.

У зависимостей существует т.н. scope, или область действия (*compile*, *provided*, *test*). Данный параметр указывает, в какой момент жизненного цикла приложения применяется данная зависимость.

Жизненный цикл сборки:

- generate-sources - фаза применяется, когда часть исходного кода автоматически генерируется
- compile - компиляция кода
- test - тестирование приложения
- package - сборка пакетов
- integration-test - интеграционное тестирование
- install - установка приложения в локальном репозитории
- deploy - установка приложения на сервер приложений, если он указан

Плагины в Maven используются для управления сборкой. Их надо в явном виде включать в модель POM. Вызов плагинов управляется общей логикой сборки проекта. Плагины в качестве точек входа содержат цели, связанные с ЖЦ сборки.

48. Системы сборки: Maven. POM. Репозитории и зависимости.

Maven - одна из самых распространенных систем декларативной сборки. Большинство действий выполняется прозрачно для разработчика и вызывается через интерфейс верхнего уровня. У пользователя остается возможность изменения выполняемых команд сборки.

Maven управляется на базе описания проекта, которое основано на POM - Project Object Model. POM состоит из нескольких частей на основе XML. В нем указаны имя, версия, местонахождение исходного кода, плагины, альтернативные профили.

Существует три типа репозиториев Maven:

Локальный репозиторий - это директория, которая хранится на нашем компьютере. Она создается в момент первого выполнения любой команды Maven и хранит все зависимости проекта (библиотеки, плагины и т.д.). Когда мы выполняем сборку проекта, то JAR-файлы всех зависимостей автоматически загружаются в локальный репозиторий. Это помогает избежать использования ссылок на удаленный репозиторий при каждой сборке проекта.

Центральный репозиторий - это репозиторий, который обеспечивается сообществом Maven. Он содержит огромное количество часто используемых библиотек. Если Maven не может найти зависимости в локальном репозитории, то автоматически начинается поиск необходимых файлов в центральном репозитории.

Удаленный репозиторий - репозиторий, определяемый самим разработчиком. Там могут храниться все необходимые зависимости. Это нужно на случай, если Maven не сможет найти необходимые зависимости и в центральном репозитории. Указывается в pom.xml.

Зависимости в Maven описываются при помощи GAV-синтаксиса - GroupId:ArtifactID:VersionID.

У зависимостей существует т.н. scope, или область действия (*compile*, *provided*, *test*). Данный параметр указывает, в какой момент жизненного цикла приложения применяется данная зависимость.

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

GNU autotools основывается на макропроцессоре 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 платформонезависимы.

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.

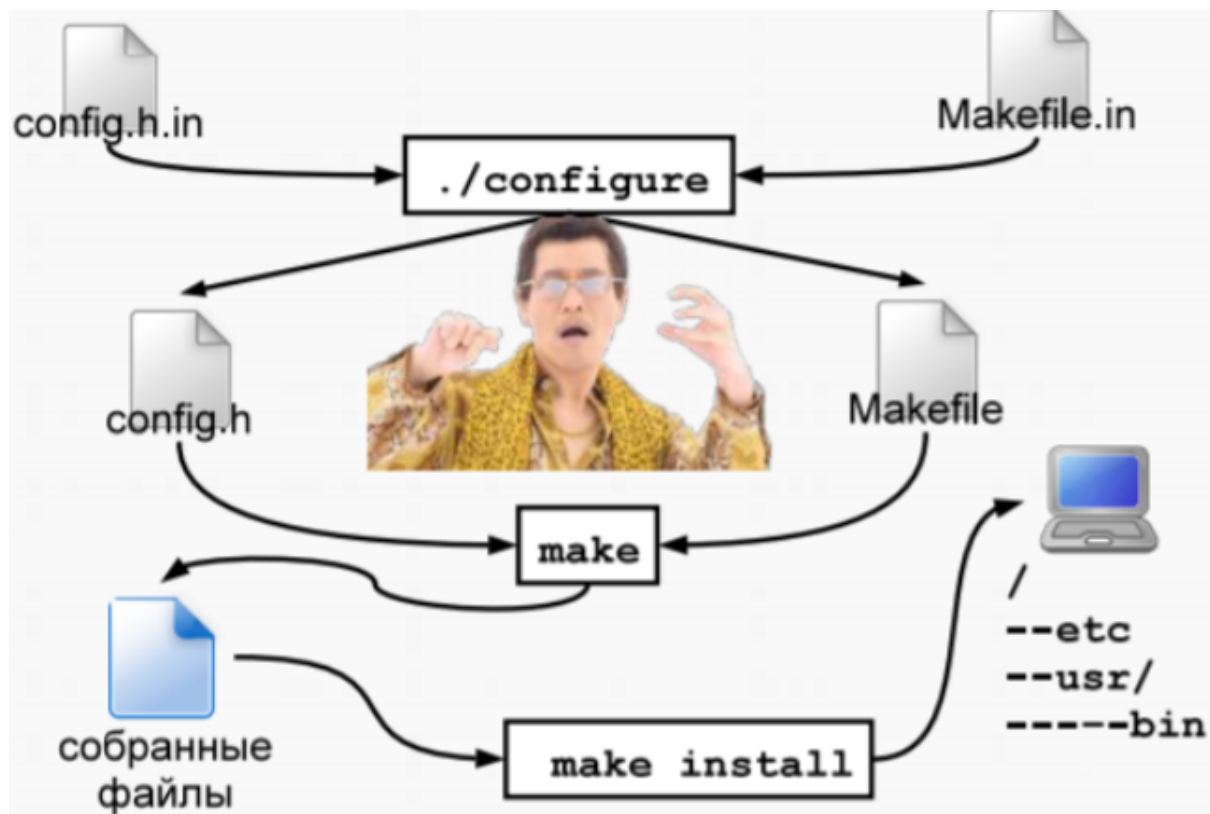
50. Системы сборки: GNU autotools. Конфигурация и сборка проекта.

GNU autotools основывается на макропроцессоре 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 создаётся файл `config.h` и необходимые платформозависимые 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 разработчика, которая привела к получению неверного результата. В широком смысле – непреднамеренное отклонение от истины или правил.
- Fault - дефект, изъян. Неверный шаг в алгоритме в программе. Следствие ошибки, потенциальная причина неисправности.
- Failure - неисправность, отказ, сбой (наблюдаемое проявление дефекта, в том числе крах или падение проги)
- Error - невозможность выполнить с использованием программы задачу, получить верный результат.
- Bug - означает дефект, отказ, невозможность выполнить задачу.

Цели тестирования:

1. Обнаружение дефектов
2. Повышение уверенности в уровне качества
3. Предоставление информации для принятия решений
4. Предотвращение дефектов

Основная цель: Увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах.

53. Понятие полного тестового покрытия и его достижимости.

Пример.

Тестовое покрытие - критерий, отображающий добротность тестирования. Характеризует полноту охвата тестами программного кода либо требований к нему.

Основной подход к оцениванию – формирование тестового пула. Значение тестового покрытия кода находится в прямой зависимости от количества отобранных вариантов проверки к нему.

Многозадачность и универсальность современного софта обуславливает невозможность организации тестового покрытия с показателем в 100%. Так что для максимального охвата тестируемого кода, разработаны особые приёмы и инструменты.

Существуют три подхода для оценки качества и выражения тестового покрытия в численном представлении в зависимости от области проверки: покрытие требований, покрытие кода и покрытие на базе анализа потока управления

54. Статическое и динамическое тестирование.

Статическое тестирование не связано 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.

Динамическое требует созданной программной архитектуры, осуществляется сборка и запуск модулей или всей системы.

55. Автоматизация тестов и ручное тестирование.

Ручное тестирование - взаимодействие профессионального тестировщика и софта с целью поиска багов. Таким образом, во время ручного тестирования можно получать фидбек, что невозможно при автоматизированной проверке. Ручное тестирование может занимать много времени, зато в краткосрочной перспективе экономит в разы больше денег. Его стоимость зависит только от тестировщика, а не инструментов для автоматизации.

Плюсы ручного тестирования:

- *Пользовательский фидбек.* Весь отчёт тестировщика может быть рассмотрен как обратная связь от потенциального пользователя.
- *UI-фидбек.* Полностью протестировать UI можно только вручную.
- *Дешевизна.*
- *Тестирование в реальном времени.* Незначительные изменения могут быть исследованы сразу, без написания кода и его исполнения.
- *Возможность исследовательского тестирования.* Его целью является проверка разнообразных возможностей приложения. Важно, что используются не заранее составленные тест-кейсы, а придуманные на лету сценарии.

Минусы:

- Человеческий фактор.
- Трудоемкость повторного использования.
- Невозможность нагрузочного тестирования.

Автоматизированное тестирование - это написание кода. С его помощью ожидаемые сценарии сравниваются с тем, что получает пользователь, указываются расхождения. Автоматизированное тестирование играет важную роль в тяжёлых приложениях с большим количеством функций.

Плюсы:

- Возможность нагрузочного тестирования.
- Можно достаточно быстро смоделировать большое количество пользователей.
- Экономия времени.
- Возможность повторного использования.

Минусы:

- Дороговизна.
- UI-тестирование не может в полной мере покрыть требования к пользовательскому интерфейсу.
- Отсутствие «человеческого взгляда». Возможны ошибки, которые заметит только человек.

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

Описание ПО является источником тестовых данных для *черного ящика* (Спецификации, Требования, Дизайн). Тесты подают на вход программы исходные данные и сравнивают результаты с эталоном.

Исходный код является источником данных для *белого ящика* (Переходы, Утверждения, Условия). В методе белого ящика возможно исследовать исходный код. Метод подразумевает построение графа с целью определения тестового покрытия. Определяется число путей обхода существующего кода программы, и соответственно максимальное необходимое число тестов.

Источниками также могут быть модели (UML) и опыт разработчика. Кроме того, разработанные во время определения требований сценарии использования системы также могут служить источником данных для формирования тестового покрытия.

Роли: проектирование тестов, автоматизация тестов, выполнение тестов, анализ результата.

57. Понятие тестового случая и сценария.

Тестовый случай состоит из набора входных значений, предусловия, ожидаемого результата и постусловий. Набор значений должен быть достаточным, чтобы покрыть особенности основных функций. Содержание спецификаций тестовых сценариев описывается в стандарте "Документация при тестирование программ" Ожидаемые результаты должны создаваться, как часть спецификация тестовых сценариев.

Если ожидаемые результаты не указаны, то правдоподобные, но ошибочные результаты могут быть приняты за верные. В идеале эталонные результаты должны быть определены до тестов. Тестовый случай с одинаковым набором данных должен приводить к одинаковому результату. Тестовый сценарий – последовательность тестовых случаев. Необходимо тестировать как позитивные, так и негативные случаи. И быть готовыми как к положительному, так и к отрицательному результату. Test Driven Development предполагает написание тестов, а по ним уже написание кода.

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

Полное тестовое покрытие недостижимо, для выбора кол-ва тестовых сценариев существует целый набор методов.

Метод эквивалентного разбиения (Анализ эквивалентности) представляет из себя анализ разных значений, с которыми функция ведет себя одинаково. Функция разбивается на части, для каждой из которых существует свой набор тестовых случаев. Отдельные тесты составляются для граничных значений. Если таких участков относительно немного, это позволяет резко сократить количество тестовых случаев. Пример на картинке.



Метод таблицы альтернативных решений: создается таблица входных и выходных данных.

Метод таблицы переходов: выделяются явные состояния внутри системы, определяются переходы между этими состояниями, которые далее покрываются тестами.

Разработанные во время определения требований сценарии использования системы могут служить источником данных для формирования тестового покрытия. При этом в каждый описанный сценарий добавляются конкретные значения, вводимые пользователем. Необходимо учитывать как основные, так и альтернативные пути сценария. Обычно каждому такому пользовательскому сценарию соответствует целая группа тестовых сценариев.

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 тестировании 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.

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 вручну 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.

	Сквозной контроль	Технический Анализ	Инспекция
Основное Предназначение	Поиск дефектов	Поиск дефектов	Поиск дефектов
Дополнительная цель	Обмен опытом	Принятие решений	Улучшение процесса
Подготовка	Обычно нет	Популяризация	Формальная подготовка
Ведущий	Автор	В зависимости от обстоятельств	Подготовленный модератор
Рекомендованный размер группы	2-7	>3	3-6
Формальная процедура	Обычно нет	Иногда	Всегда
Объем материалов	небольшой	От среднего до большого	небольшой
Сбор метрик	Обычно нет	Иногда	Всегда
Выходные данные	Неформальный отчет	Формальный отчет	Список дефектов, результаты метрик, формальный отчет

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 (нефункциональные 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 бета-тестирование.

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.

Системный и архитектурный аспект. Архитектура может быть распределенной, где требуется баланс производительности на всех уровнях обработки информации, кластерной, виртуализованной.

Низкоуровневый аппаратный аспект. Связан с техническими характеристиками аппаратуры. Пример: от тактовой частоты зависит линейная скорость выполнения каждого потока программы.

Программный аспект. Связан с выбором подходящих алгоритмов для разрабатываемых программ. Зачастую сроки важнее оптимальности решения задач. Пример: Qsort и пузырьки.

Человеческий фактор. Во время анализа сначала выбираются критерии оценки, после этого выбираются средства измерения, бывают неинтрузивные (не оказывают влияние на результаты), интрузивные (оказывают влияние на результаты) и слабо интрузивные. В вычислительных системах любое измерение влияет на результат. Далее необходимо выбрать нагрузку и нагрузить систему. Нагрузку необходимо приблизить к пользовательской. В конце провести анализ и внести изменения, основанные на его результатах.

Применительно к измерению производительности вычислительных систем, практически любое измерение тем или иным образом влияет на результат самого измерения, так как тоже требует некоторых ресурсов.

Например, счетчики производительности операционной системы работают незаметно для пользователя, но, если бы они не собирались, процессорное время на уровне системы не тратилось бы, и отдавалась пользовательской программе.

Например, если журнал измерений растёт со скоростью 1 Мб/с, то это значение нужно исключить из производительности дисков.

66. Ключевые характеристики производительности.

Параметры связаны между собой.

1. Время отклика системы (latency) - измеряется от выдачи запроса до получения первых результатов.
2. Пропускная способность - показатель максимального числа запросов через канал ввода-вывода, систему, узел.
3. Утилизация ресурса (%util) - показывает на сколько процентов занят ресурс.
4. Занятость (%busy) - время работы одного процесса.
5. Ожидание (%wait) - сколько времени не пуста очередь.
6. Точка насыщения - точка достижения максимальной производительности.
7. Масштабируемость - на сколько еще можно загрузить систему.
8. Эффективность - это отношение полезной работы ко всей.

67. Нисходящий метод поиска узких мест.

Классический нисходящий метод поиска узких мест последовательно рассматривает систему от более общих компонент к более частным.

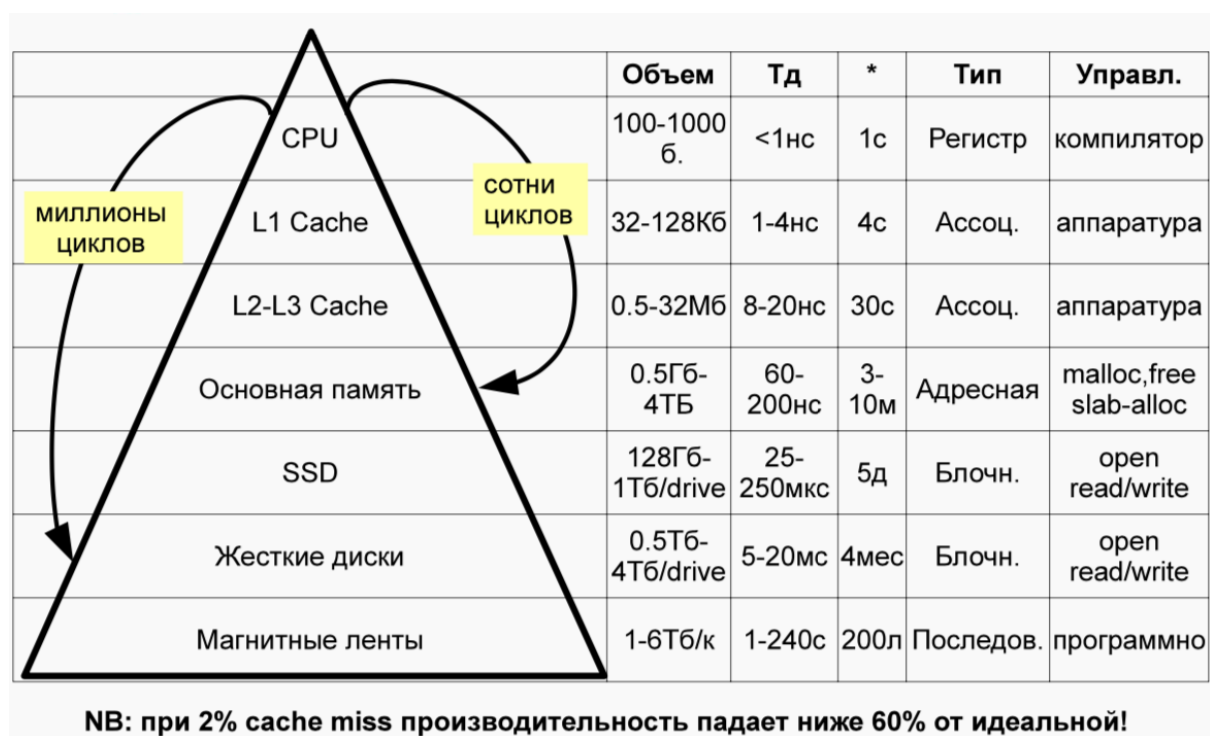
В первую очередь ищутся ошибки аппаратуры и конфигурации системы администраторами. Проверяются системные журналы, файлы конфигурации системы и прикладного ПО.

После исключения ошибок начинается наблюдение за системой. Операционная система обладает достаточно развитыми средствами наблюдения за своими подсистемами. Средства системного межузлового мониторинга помогают наблюдать общую картину работы приложения.

Следующим этапом является наблюдение за самим приложением. Фиксируются алгоритмические проблемы, проблемы API, проблемы, связанные с многопоточностью, блокировками и синхронизацией. Используются средства мониторинга и профилирования приложений.

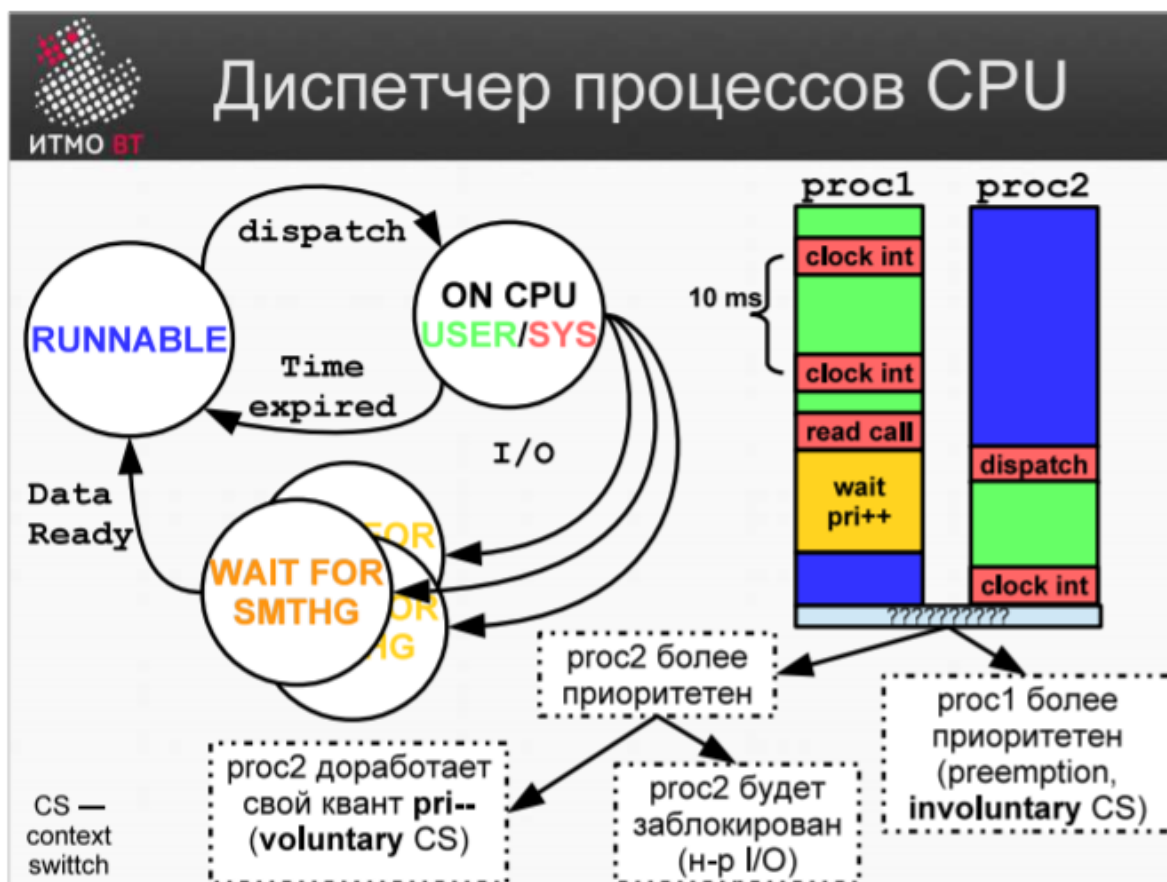
Если полученного в ходе предыдущего испытания прироста недостаточно, происходит переход к мониторингу микроархитектуры.

68. Пирамида памяти и ее влияние на производительность.



Сверху пирамиды находится самая дорогая и маленькая память. Чем ниже память располагается в пирамиде, тем дольше обращение к ней. И нужно минимизировать кол-во обращений к медленной памяти. Стоит помнить про локальность памяти.

69. Мониторинг производительности: процессы.



Показанная схема приблизительно одинакова во всех современных операционных системах. Результаты диспетчеризации можно контролировать при помощи системных утилит.

Процесс или поток внутри процесса может глобально находиться в трёх состояниях: он может быть готов к исполнению (Runnable), находиться на исполняющем устройстве (либо на уровне пользователя), либо на уровне ОС (On CPU User/Sys), или ожидать ввода/вывода, освобождения блокировки и тп. (Wait for smthg). При этом, во время ожидания нет необходимости занимать ресурсы процессора.

Пусть процесс **proc1** в настоящий момент находится на процессоре и работает программа пользователя. Раз в 10 мс (время может быть другим) происходит т.н. прерывание от часов (clock interrupt), во время которого ОС наращивает счетчики производительности, проверяет наличие в очереди на исполнение процесса с более высоким приоритетом, чем у текущего, а также проверяет, не исчерпал ли текущий процесс свой квант времени. В нашем случае, исполнение **proc1** продолжается, так как квант времени не исчерпан и **proc2** не является более приоритетным, чем **proc1**, поэтому он остается в режиме готовности в очереди на выполнение. Так продолжается до тех пор,

пока не будет исчерпан квант времени, либо не будет сделан вызов из proc1 к устройству ввода-вывода.

При появлении такого вызова (на слайде — read call), proc1 переходит в режим ожидания, и происходит т.н. context switch (процессор переходит к исполнению другого процесса, сохранив окружение предыдущего). Внутри ядра происходит диспетчеризация, в ходе которой выбирается процесс с высшим приоритетом, готовый к выполнению. Процесс proc2 начинает выполняться. Процесс proc1 отправляется в состояние ожидания, но у него повышается приоритет (чем больше время ожидания, тем больше увеличивается приоритет, и наоборот, трата процессорного времени приводит к понижению приоритета, что типично для систем разделения времени).

В момент прерывания возможны различные варианты действий. Если приоритет proc2 выше, чем у proc1, то он либо дорабатывает свой квант времени, либо выполняет операцию, связанную с вводом-выводом. Если proc1 более приоритетен, то он снова перейдет на процессор (произойдет вытеснение - preemption, или несвободное переключение контекста - involuntary CS).

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 swar-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 просканированных страниц за единицу времени. Если эта величина высокая, значит не хватает оперативной памяти.

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 (`avgqr-sz` и `avgqw-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 или 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. Команда `iostat` показывает рассмотренные выше характеристики обмена.

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 неинтрузивными.

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 интрузивен.


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

73. Системный анализ Linux "за 60 секунд".

ИТМО BT

Системный анализ "за 60 секунд"

- uptime — load average за 1, 5, 15 минут.
- dmesg | tail — последние ошибки.
- vmstat 1 — есть ли свободная память, paging, распределение CPU.
- mpstat -P ALL 1 — распределение по CPU.
- pidstat 1 — статистика по процессам, горячие процессы .
- iostat -xz 1 — параметры ввода-вывода.
- free -m — проверка исчерпания кэшей/буферов.
- sar -n DEV 1 — сетевая статистика по интерфейсам.
- sar -n TCP,ETCP 1 — сетевая статистика по соединениям.
- top — онлайн-мониторинг параметров.



http://www.brendangregg.com/Articles/Netflix_Linux_Perf_Analysis_60s.pdf

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 инфо 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 ограниченным 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 загрузка 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 использования и настроить кэши.
- 3) Проблемы с аппаратурой, нужно купить более совершенную дисковую систему. Эффективны SSD-накопители

79. Рецепты повышения производительности при высоком %Idle.

При высоком времени простоя СПУ 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 загрузке 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 восстановиться.
3. На уровне микроархитектуры важно избавиться от кэш-промахов и промахов мимо TLB, Кэш промахи можно исправить работой над структурой данных, а мимо тлб можно исправить использованием больших страниц памяти.