

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

35. Subversion. Архитектура системы и репозиторий.
36. Subversion: Основной цикл разработчика. Команды.
37. Subversion: Конфликты. Слияние изменений.
38. GIT: Архитектура и команды.
39. GIT: Организация ветвей репозитория.
40. GIT: Плагин git-flow.
41. Системы автоматической сборки: предпосылки появления
42. Системы сборки: Make и Makefile.
43. Системы сборки: Ant. Команды Ant.
44. Системы сборки: Ant-ivy.
45. Системы сборки: Maven. POM. Репозитории и зависимости.
46. Maven: Структура проекта. GAV.
47. Maven: Зависимости. Жизненный цикл сборки. Плагины.
48. Системы сборки: Gradle. Преимущества и файл сборки.
49. Системы сборки: GNU autotools. Создание конфигурации проекта.
50. Системы сборки: GNU autotools. Конфигурация и сборка проекта.
51. Сервера сборки/непрерывной интеграции.
52. Основные понятия тестирования. Цели тестирования.
53. Понятие полного тестового покрытия и его достижимости. Пример.
54. Статическое и динамическое тестирование.
55. Автоматизация тестов и ручное тестирование.
56. Источники данных для тестирования. Роли и деятельности в тестировании.
57. Понятие тестового случая и сценария.
58. Выбор тестового покрытия и количества тестов. Анализ эквивалентности.
59. Модульное тестирование. Junit 4.
60. Интеграционное тестирование. Стратегии интеграции.
61. Функциональное тестирование. Selenium.
62. Техники статического тестирования. Статический анализ кода.
63. Тестирование системы в целом. Системное тестирование. Тестирование производительности.
64. Тестирование системы в целом. Альфа- и бета-тестирование.
65. Аспекты бысродействия системы. Влияние средств измерения на результаты.
66. Ключевые характеристики производительности.
67. Нисходящий метод поиска узких мест.

68. Пирамида памяти и ее влияние на производительность.
69. Мониторинг производительности: процессы.
70. Мониторинг производительности: виртуальная память.
71. Мониторинг производительности: буферизированный файловый ввод-вывод.
72. Мониторинг производительности: Windows и Linux.
73. Системный анализ Linux "за 60 секунд".
74. Создание тестовой нагрузки и нагрузчики.
75. Профилирование приложений. Основные подходы.
76. Компромиссы (trade-offs) в производительности.
77. Рецепты повышения производительности при высоком %SYS.
78. Рецепты повышения производительности при высоком %IO wait.
79. Рецепты повышения производительности при высоком %Idle.
80. Рецепты повышения производительности при высоком %User.

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

Официальная классификация процессов программной инженерии задается международным стандартом ISO/IEC 12207:2008 “Systems and Software Engineering — Software Life Cycle Processes” и его российским аналогом ГОСТ Р ИСО/МЭК 12207-2010

В этих стандартах процессы привязываются к понятию “Жизненный цикл программного обеспечения”. ЖЦ программного обеспечения называется период времени, который начинается с момента замысла программного продукта и заканчивается в момент, когда ПО больше недоступно для использования.

Основные этапы (они повторяются во всех методологиях в той или иной форме)

- разработка требований (заказчик с разработчиком)
- анализ (определение способов решения)
- проектирование
- разработка
- тестирование (одновременно с разработкой формируются подходы к тестированию)
- внедрение
- эксплуатация (поддержка пользователей, доработки)
- вывод из эксплуатации

Каждый процесс в стандарте ISO описывается по схеме "Входные данные и ресурсы - совокупность действий – выходные данные и ресурсы". Сама совокупность действий не определена конкретно.

Действия, которые могут выполняться во время жизненного цикла ПО распределены по двум категориям. Одна из них называется “Процессы в контексте системы” описывает

процессы для работы с непосредственно программным продуктом. Другая категория называется “Специальные процессы программных средств”, и она содержит в себе процессы в отношении программного продукта, являющегося частью более крупной системы.

Процессы в первой категории делятся на группы:

- **«Процессы соглашения»** нужны для выработки соглашений между стороной заказчика и стороной разработки
- **«Процессы организационного обеспечения проекта»** нужны для организации ресурсов и инфраструктуры для обеспечения удовлетворения организационных целей и установленных соглашений
- **«Процессы проекта»** нужны для планирования, выполнения и оценки и управления продвижением проекта, а также выполнение специализированных целей менеджмента
- **«Технические процессы»** обеспечивают определение требований к системе, их преобразование в полезный программный продукт, его применение и изъятие из обращения, если он не используется

Процессы во второй категории делятся на группы “Процессы реализации программных средств”, “Процессы поддержки программных средств”, “Процессы повторного использования программных средств”. Как уже было озвучено ранее, эта категория описывает процессы в отношении программного продукта, являющегося частью более крупной системы, чем рассматриваемая.

Любой крупной компании-разработчику следует следовать этим процессам для обеспечения надлежащего качества разработки. Однако хочется отметить, что процесс разработки не является предписанием, которому команда должна следовать догматически. Напротив, он должен быть очень гибким и адаптивным (к проблеме, проекту, команде и культуре организации). Таким образом, процесс, адаптированный для одного проекта, может существенно отличаться от процесса, адаптированного для другого проекта. Если модели процессов применяются догматически и без какой-либо адаптации, проект рискует лишь утонуть в бюрократии без какого-либо преимущества для всех заинтересованных лиц.

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

Модель жизненного цикла программного обеспечения — это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего жизненного цикла ПО.

Существует несколько моделей ЖЦ (они же стратегии разработки ПО):

- **Последовательная** (она же “Однократный проход”, Waterfall): в этой модели заранее определены все требования, это позволяет провести все стадии разработки последовательно и один раз. Получившиеся результаты окончательны и, как правило, не подлежат пересмотру (например, проектирование марсохода, после того, как мы его отправим в космос, мы уже не сможем поменять его внутреннюю начинку), но даже если и допустима фаза сопровождения программного продукта,

обратная связь требует возврата к самой начальной стадии (и поэтому ее ожидание может занимать какое-то время). Тем не менее, для такой модели легко спрогнозировать и предъявить заказчику стоимость и сроки разработки.

- **Инкрементная:** здесь нам также заранее известны все требования, однако мы разбиваем продукт на несколько эквивалентных (с архитектурной точки зрения) частей и производим разработку в несколько этапов. Первая версия реализует часть запланированных возможностей, следующая вводит дополнительные возможности и т. д. пока не получим готовый продукт.
- **Эволюционная:** в реальном мире часто бывает такое, что на старте нам известны далеко не все требования к продукту, и эволюционная модель ЖЦ решает эту проблему: система также строится в виде последовательности версий, но требования уточняются по ходу разработки.

В реальной разработке используется инкрементно-эволюционный подход, поскольку чисто инкрементной или чисто эволюционной стратегии развития ПО практически не существует.

Также необходимо упомянуть о модели формальных преобразований, хотя она слабо представлена на рынке разработки. В ней создаются модели ПО, которые последовательно преобразуются друг в друга, а затем в программный код по определенным формальным принципам.

Сравнительная таблица моделей ЖЦ:

Стратегия разработки // В начале процесса определены все требования? // Множество циклов разработки? // Промежуточное ПО распространяется?

Однократный проход // Да // Нет // Нет (отсутствует)

Инкрементная (запланированное улучшение продукта) // Да // Да // Может быть

Эволюционная // Нет // Да // Да

3. Водопадная (каскадная) модель.



{обозначить цифрами}

Разработана приблизительно в 60-х годах и описана Уинстоном Ройсом в 70-х. Суть заключается в том, что у нас есть 7 фаз разработки ПО и есть итерации между ними, в том порядке, в котором они перечислены:

- Стадия определения требований к системе
- Стадия определения требований к ПО
- Стадия анализа требований
- Стадия проектирования программы
- Стадия разработки кода
- Стадия тестирования
- Стадия введения ПО в эксплуатацию

Имеется возможность откатиться к предыдущей фазе, но подчеркивается сложность этого действия. Так, например, тестирование — это первая фаза, на которой может быть обнаружено, что ожидаемые характеристики (функциональность, производительность, объем данных и др.) отличаются от утвержденных в результате анализа, в результате чего придется полностью пересмотреть либо требования, либо дизайн системы, что приводит к глобальной переделке системы и потенциально к росту сроков в 2 раза и стоимости разработки.

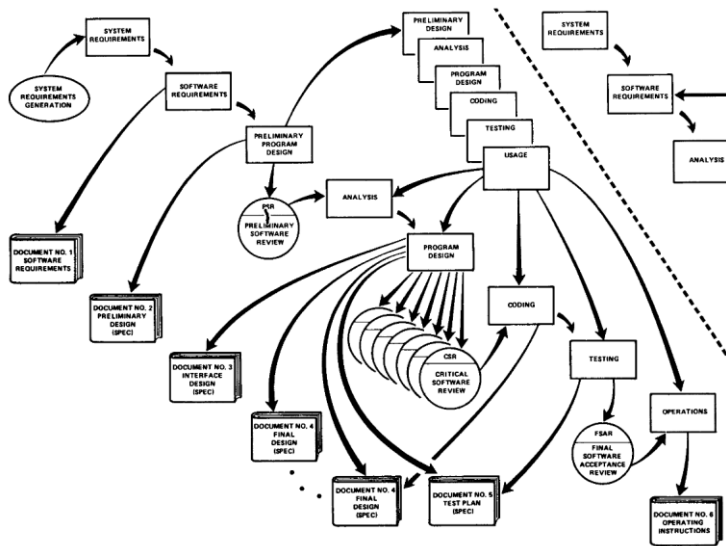
4. Методология Ройса.

Методология Ройса берет за основу каскадную модель, однако существенно дорабатывает ее главный недостаток: высокую рискованность (потенциальное глобальное переделывание системы), вводя 5 шагов:

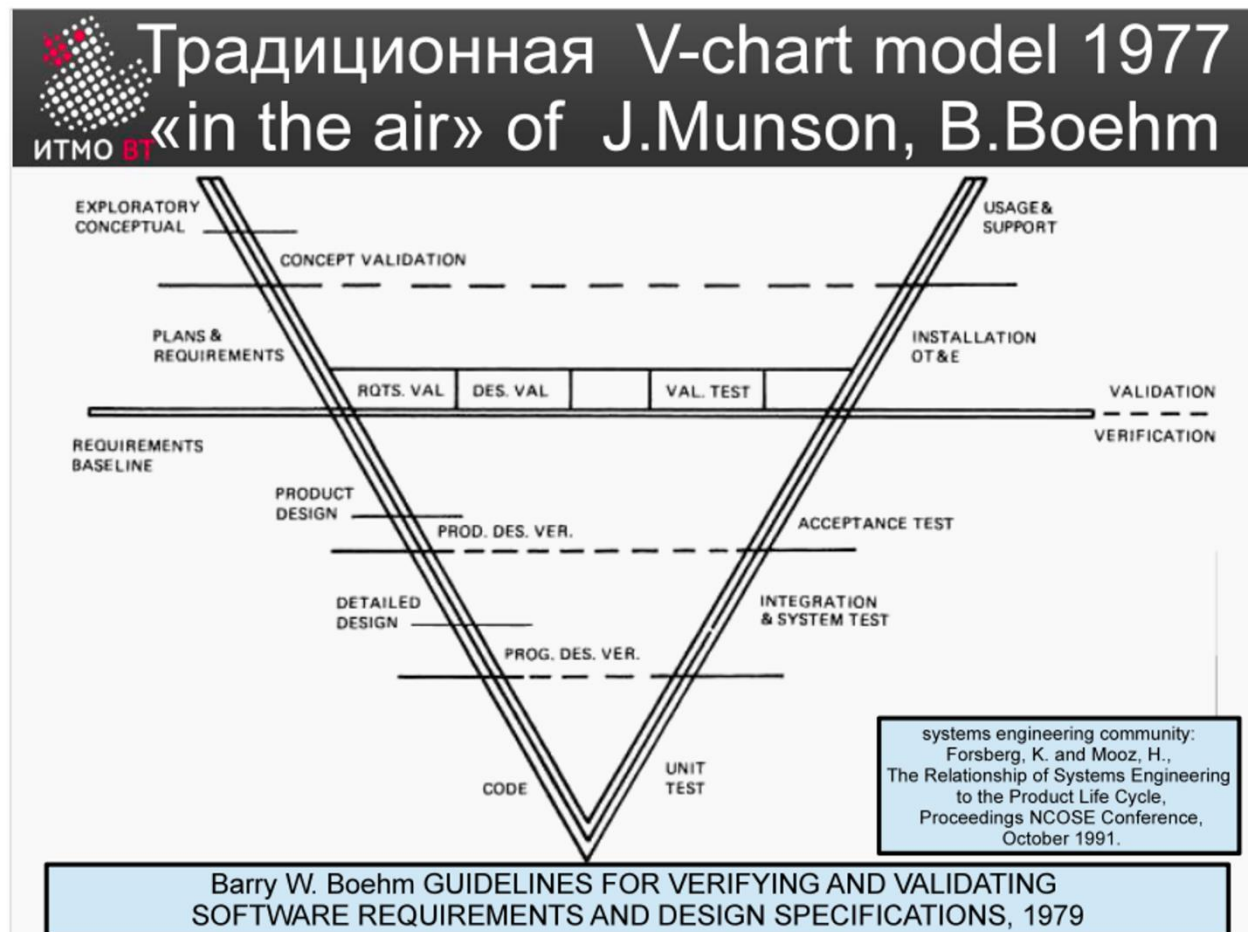
1. **Сначала дизайн программы.** Ройс определил стадию предварительного дизайна ПО между сбором требований и их анализом. Несмотря на то, что из-за отсутствия анализа дизайнер не может учесть все требования, в его силах убедиться, а могут ли они быть вообще реализованы. К этой стадии не допускается участие программистов и аналитиков. В ней предлагается спроектировать, определить и создать модель обработки данных, а также создать документ-обзор системы.
2. **Документирование.** Ройс включает важные документы процесса разработки:
 - Требования к системе
 - Спецификация предварительного дизайна
 - Спецификация дизайна интерфейсов
 - Финальная спецификация дизайна системы
 - План тестирования
 - Инструкция пользовательской эксплуатации
3. **Делай это дважды.** Ройс предлагает параллельно с основной разработкой создавать упрощенную версию системы с ускоренными сроками разработки и использовать ее в качестве пилотной версии. Это позволяет подтвердить или опровергнуть основные характеристики ПО.
4. **Контроль тестирования.** Поскольку тестирование это наиболее рискованная фаза с точки зрения денег и сроков, ему должно быть уделено особое внимание. При планировании тестирования из процесса тестирования должен быть исключен дизайнер системы, должна быть проведена “визуальная инспекция” (повторный

просмотр кода другим лицом, которое без глубокого анализа выявит проблемы в логических путях и/или заметные визуальные дефекты). После исправления простых ошибок провести проверку ПО в тестовом окружении.

5. **Подключение пользователя.** Во время разработки Ройс предложил давать возможность наблюдения непосредственно пользователю конечной системы. Для этого выделено 3 точки: предварительный, критически и финальный просмотр. На каждой из этих точек пользователю дается возможность получить пользовательский опыт, его оценку и подтверждение дальнейшего продвижения разработки.



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



Поскольку в то время (конец 1970х) много внимания уделялось качеству, а как следствие тестированию, то многие модели того времени сфокусировались именно на нем.

Так, в 1977 году Барри Боем и Джэк Мансон предложили V-образную модель.

За основу V-chart модели была взята каскадная модель, однако ее слегка изогнули в V-образную форму, сопоставив почти каждой фазе из каскадной модели соответствующее тестирование (отсюда и название — V-chart)

- Концепт системы ↔ Использование и поддержка

--- Линия валидации концепта

- Сбор требований и планирование ↔ Установка системы в конечном окружении

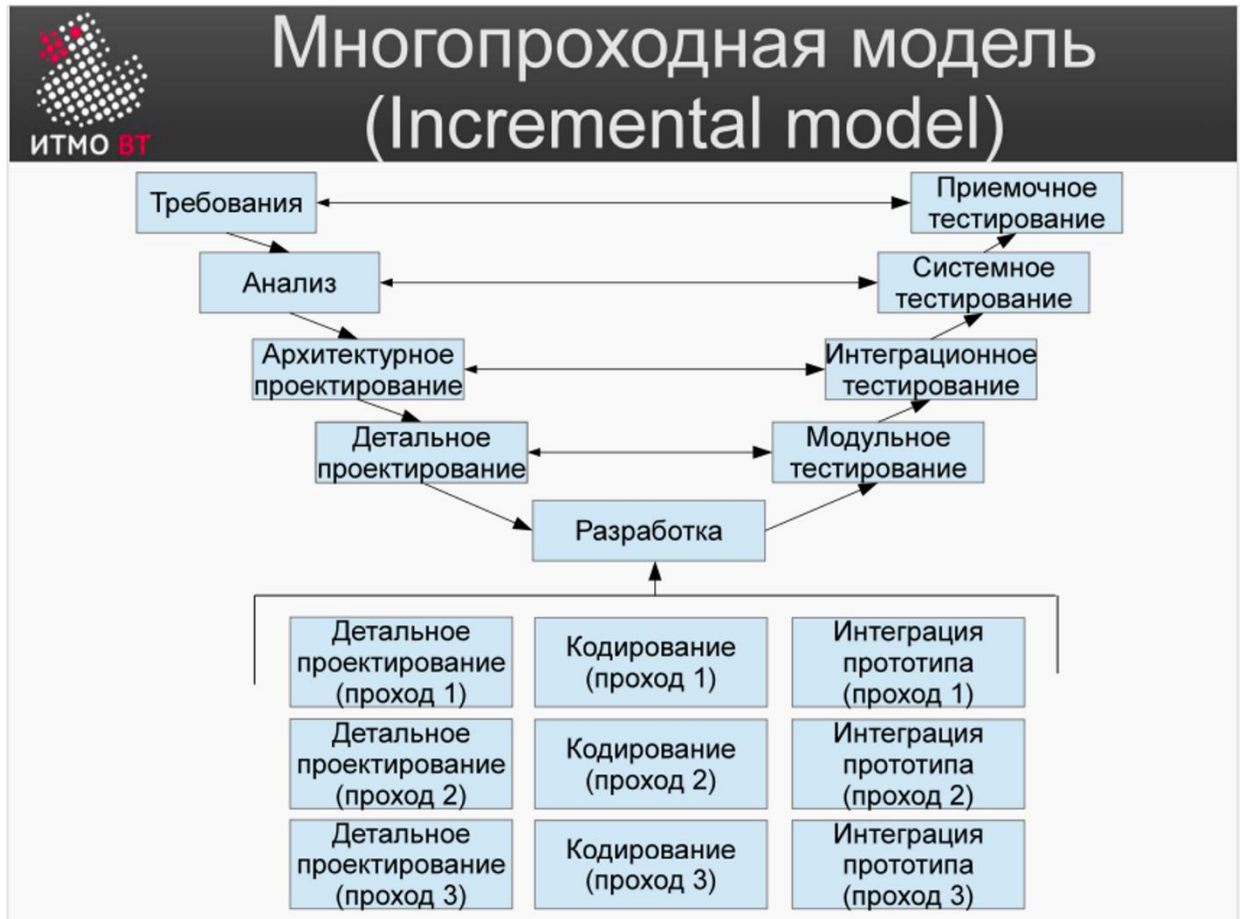
--- Граница сбора требований ↔ валидации и верификации. Ниже — верификация.

- Дизайн продукта ↔ Приемочное тестирование
- Детализация дизайна ↔ Системное и интеграционное тестирование
- Кодирование ↔ Unit-тестирование

Необходимо также определить эталонное поведение системы, и четко его задать вне кода, поскольку с ним будет сравниваться результат во время тестирования.

Динамическое тестирование включает компьютерное исполнение тестов, а статическое тестирование предполагает проверку артефактов разработки без их компьютерного исполнения, например, спецификации, технических решений, дизайна и пр.

6. Многопроходная модель (Incremental model).



Частный случай V-chart модели. На этапе разработки происходит вместо одного этапа несколько, каждый включающий в себя:

- Детальное проектирование
- Кодирование
- Интеграция прототипа

Разработка становится более управляемой и видимой для заказчика, но есть и недостатки. Архитектура системы и многие ее части стремятся к устареванию и деградированию, то есть частям системы будет требоваться полный и/или частичный рефакторинг за счет средств заказчика. А из-за высокой скорости изменений поддерживать документацию в больших проектах становится также достаточно сложно. Помимо этого, присуще сложности с заключением контракта и оцениванием стоимости разработки программного продукта.

7. Модель прототипирования (80-е).



Суть данной модели заключается в том, что мы сначала быстро спланируем всю итерацию, быстро проанализируем требования и подходы к реализации, разработаем прототип, покажем его заказчику. И если заказчику все понравится, то мы разрабатываем окончательную версию ПО, тестируем и внедряем ПО, далее сопровождаем его. В ином случае делаем новый прототип, новый интерфейс и базу данных, пока заказчик не скажет, что его все устраивает.

В современных методах разработки применяются развитые средства макетирования для создания такого рода прототипов. Это так называемые "мокапы" (mock-up), простые визуальные модели пользовательского интерфейса, а также более сложные UX-модели (от User Experience Design)- модели интерфейса, которые можно дать попробовать «понажимать» пользователю для оценки удобства программы.

8. RAD методология.



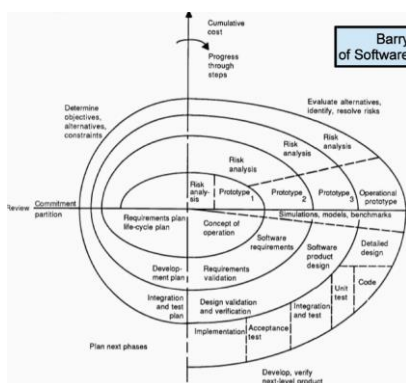
Rapid Application Development порождена корпорацией IBM в 1980-е годы, обнародована в 1991. Идея заключается в том, что в те года была очень слабо развита автоматизация бизнес-процессов, программы писались в текстовом редакторе, база данных была доступна практически в ручном режиме и невозможно было разрабатывать ПО в терминах бизнес-логики.

Решением проблемы заключалось создание интегрированных сред разработки, а методология заключается в том, что после небольшого планирования из готовых компонентов, как из кубиков, собирается ПО.

Пользователь принимает непосредственное участие в процессе разработки. Это обычно не отличается продуманным интерфейсом или удобством разработки, но это позволяет очень быстро разработать функциональность, которая была нужна “уже вчера”.

Сегодня таковыми примерами можно считать Oracle E-Business Suite. В них можно создавать формы данных, выполнять запросы к системе с помощью пользовательского интерфейса.

9. Спиральная модель.



В начале 80-х, проанализировав все существующие модели, Б. Боем сформулировал метод разработки крупного программного обеспечения, который получил название спиральной модели.

Каждый виток спирали представляет собой одну фазу разработки продукта для построения очередной версии ПО или прототипа. Сначала производится постановка целей

для каждой итерации, выявляются альтернативные решения и ограничения, которые необходимо учитывать в данной фазе.

Затем производится анализ рисков, за счет снижения вероятности возникновения которых минимизируется число действий, необходимых для разработки ПО. На каждой итерации спиральной модели перед созданием прототипа производится сверка с картой рисков.

После анализа рисков производится разработка и валидация полученной версии ПО.

На последней части витка производится планирование следующей итерации. В зависимости от фазы планированию подлежат действия по разработке требований и жизненного цикла разработки, собственно, сама разработка, а также действия по интеграции разработанных частей в единое целое и проведение тестирования.

Ключевой особенностью метода стало включение в него постоянной оценки изменений и рисков. На каждой итерации спирали при переходе между четвертями мы анализируем риски — и проект может не перейти к следующей стадии, если он не прошел риск-анализ.

10. UML Диаграммы: Структурные и поведенческие.

UML — графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования артефактов, создаваемых при разработке программного продукта.

Диаграммы - это представления моделей UML. Они показывают наборы сущностей, которые раскрывают тот или иной аспект программной системы и являются способом визуализации того, что или как будет делать моделируемая система.

UML диаграммы делятся на структурные и поведенческие:

- К **структурным** диаграммам относятся доменные модели (они же диаграммы классов), диаграмма развертывания. Эти диаграммы нужны для статического описания системы, демонстрации ее архитектуры, конфигурации или специфических элементов предметной области.
- К **поведенческим** диаграммам относятся диаграммы прецедентов использования (они же Use-Case диаграммы), диаграммы последовательностей/состояний. Они представляют собой динамическую картину действий, происходящих в системе.

11. UML: Use-case модель.

Use-case модель отражает требования к программному продукту и представляет из себя диаграмму вариантов использования программного продукта.

Главным элементом модели является действующее лицо (actor) — пользователь, который непосредственно взаимодействует с системой. Под пользователем указывается его роль.

Эктор никогда не является частью системы. Под эктором тоже можно считать время (в ряде случаев)

Прецедент содержит в себе действие, выполняемое в системе (например, “Пользователь вводит логин и пароль”. Прецеденты одних модулей могут выступать экторами для других модулей.

Ассоциация используется в диаграмме прецедентов для указания того, что эктор использует систему и выполняет указанный прецедент.

Также в Use-Case модели используются следующие стереотипы:

- **Стереотип Include (включение)** организует иерархию прецедентов использования системы и позволяет включать одну общую деятельность в несколько Use-Case’ов.
- **Стереотип Extend (точка расширения функциональности)** указывает, собственно, на точку расширения функциональности. Например, при просрочке возврата книги в библиотеке с пользователя взимут штраф. То есть при наличии определенных условий будет применена дополнительная функциональность. Не стоит путать со стереотипом Generalization.
- **Стереотип Generalization (обобщение)** указывает на отношение is a (A является B), показывающая, что данный объект является потомком более высокого класса объектов.

Напоследок, в Use-Case диаграмме используется элемент System Boundary (граница системы). Он представляет интерес, когда есть несколько подсистем, и требуется знать разделение между ними.

- Actor — действующее лицо.

- Use Case — прецедент использования.

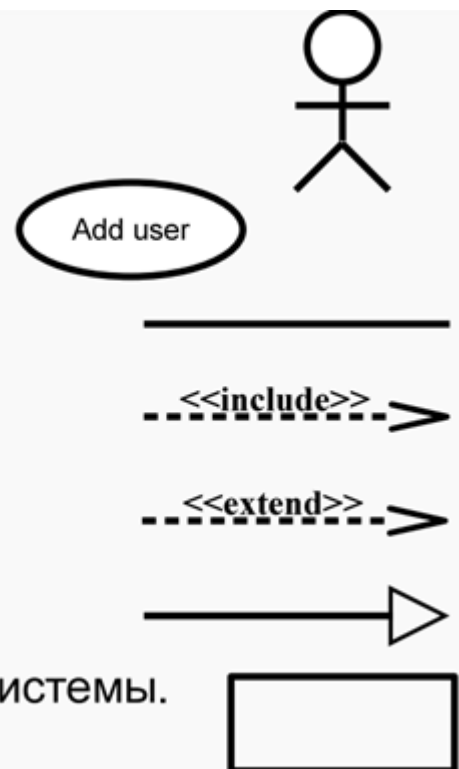
- Association — ассоциация, использование.

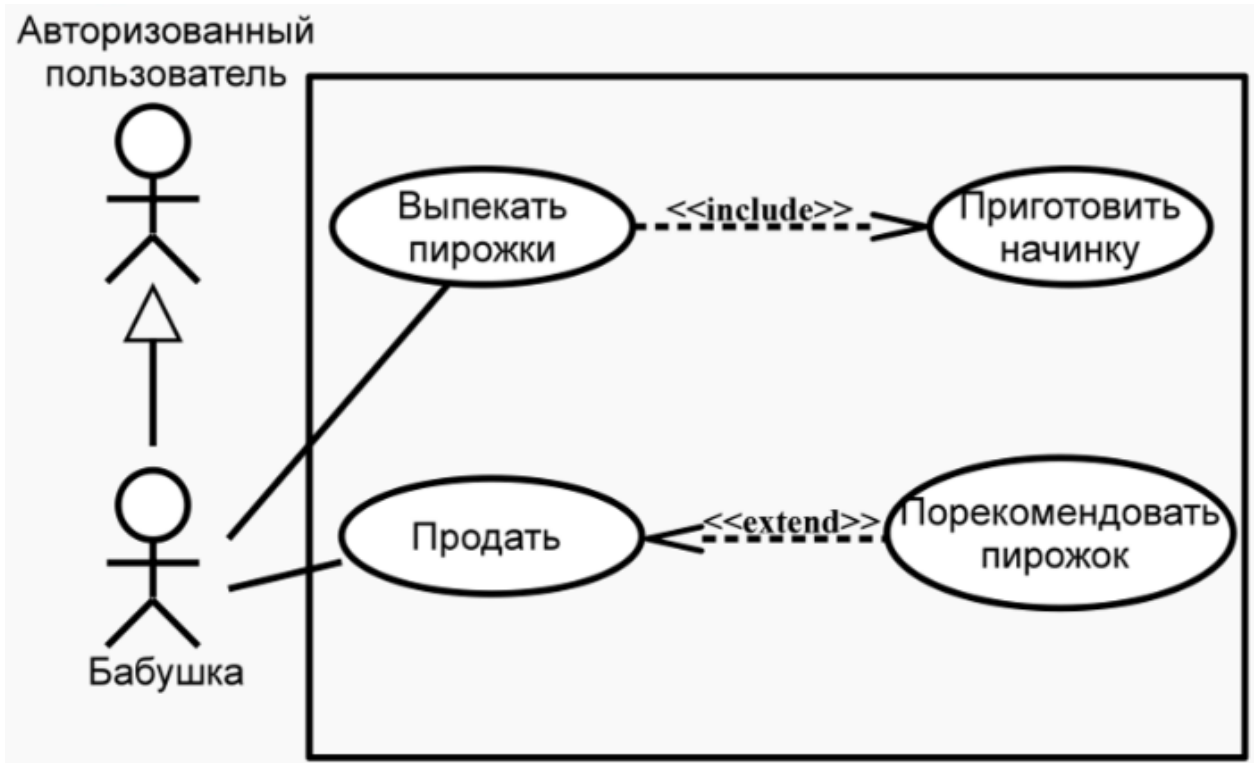
- Include — включение.

- Extend — точка расширения функционала.

- Generalization — обобщение.

- System boundary — границы системы.





В системе «Домашние пирожки» выпуска и продажи пирожков актер «Бабушка», которая является потомком актора «Авторизованный пользователь», может «Выпекать пирожки» и «Продать» их. Use Case «Выпекать пирожки» включает в себя деятельность «Приготовить», которая также может быть включена как часть другого прецедента, например «Выпекать торты» для актора «Кондитер». Расширяющим функционалом прецедента «Продать» является прецедент «Порекомендовать пирожок», если покупатель не может определиться с выбором.

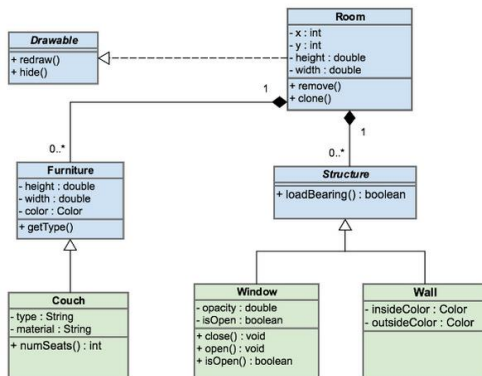
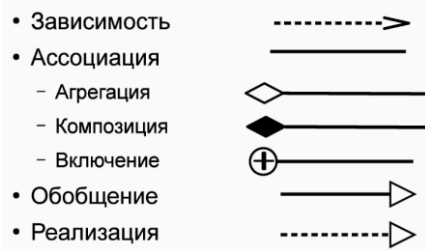
12. UML: Диаграмма классов.

Диаграмма классов (она же доменная модель) является примером структурной диаграммы и отображает в первом приближении реализованную предметную область. Например, мы рисовали такую диаграмму на 1 курсе к каждой лабораторной работе по программированию на Java. Преимущество такой диаграммы заключается в ее наглядном изображении предметной области. Данная диаграмма по мере дальнейшей работы на стадиях проектирования может быть уточнена и расширена. Например, могут быть добавлены типы данных и область их видимости, кратность и роли ассоциаций и т.д..

Также здесь можно сказать про отношения UML:

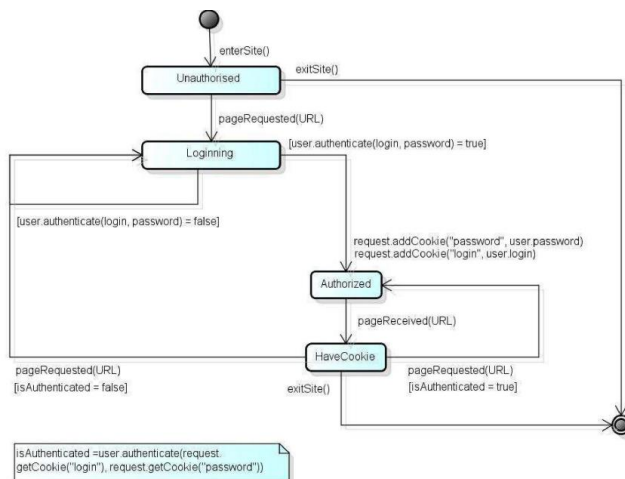
- **ассоциация**, целевой элемент связан с исходным и есть три подвида
 - **агрегация**, целевой элемент является частью исходного и может существовать отдельно
 - **композиция**, четкая форма включения, часть не может существовать без целого и доступна только через точки взаимодействия.
 - **включение**, исходный элемент содержит целевой и имеет доступ к его пространству имен
- **обобщение**, исходный элемент является специализацией более обобщенного (целевого) элемента. т.е это наследование

- **реализация**, исходный элемент гарантированно выполняет контракт, который определяется целевым элементом, реализует его интерфейс



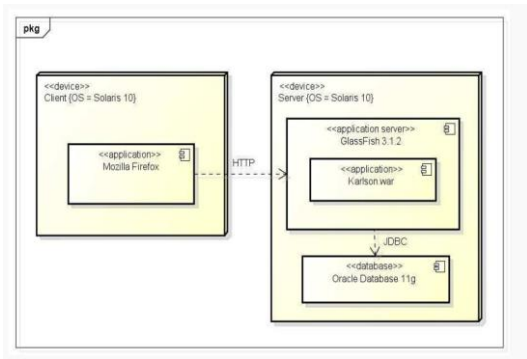
13. UML: Диаграмма состояния

Диаграмма состояний (диаграмма конечных автоматов) является примером поведенческой диаграммы и отображает модель “Finite state machine” (программа конечных автоматов), представляющая собой набор состояний и условиями переходов между ними.



14. UML: Диаграмма размещения

Диаграмма размещения является примером структурной диаграммы и отображает физическую архитектуру размещения программного обеспечения, а также возможности его использования. Например, на этой диаграмме может быть нарисована серверная машина, в которой имеется сервер приложений Java, в нем крутится наша лаба по веб-программированию, а также может иметься пользовательская машина с запущенным в ней браузером, и соединение происходит по протоколу связи HTTPS.



Клиентом разработанной системы служит браузер Файрфокс, который по протоколу HTTP осуществляет запрос страниц на сервере, сервер представляет собой программно-аппаратный комплекс под управлением операционной системы SolarisIO, на нем размещено два программных компонента - база данных и сервер приложений. Приложение представлено в виде артефакта Karlson.war.

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

* Unified Process методологии берут за собой идею о создании четкой унифицированной структуры создания программного продукта, подходящей для любой команды людей, позволяющей создать с ее помощью любой программный продукт.

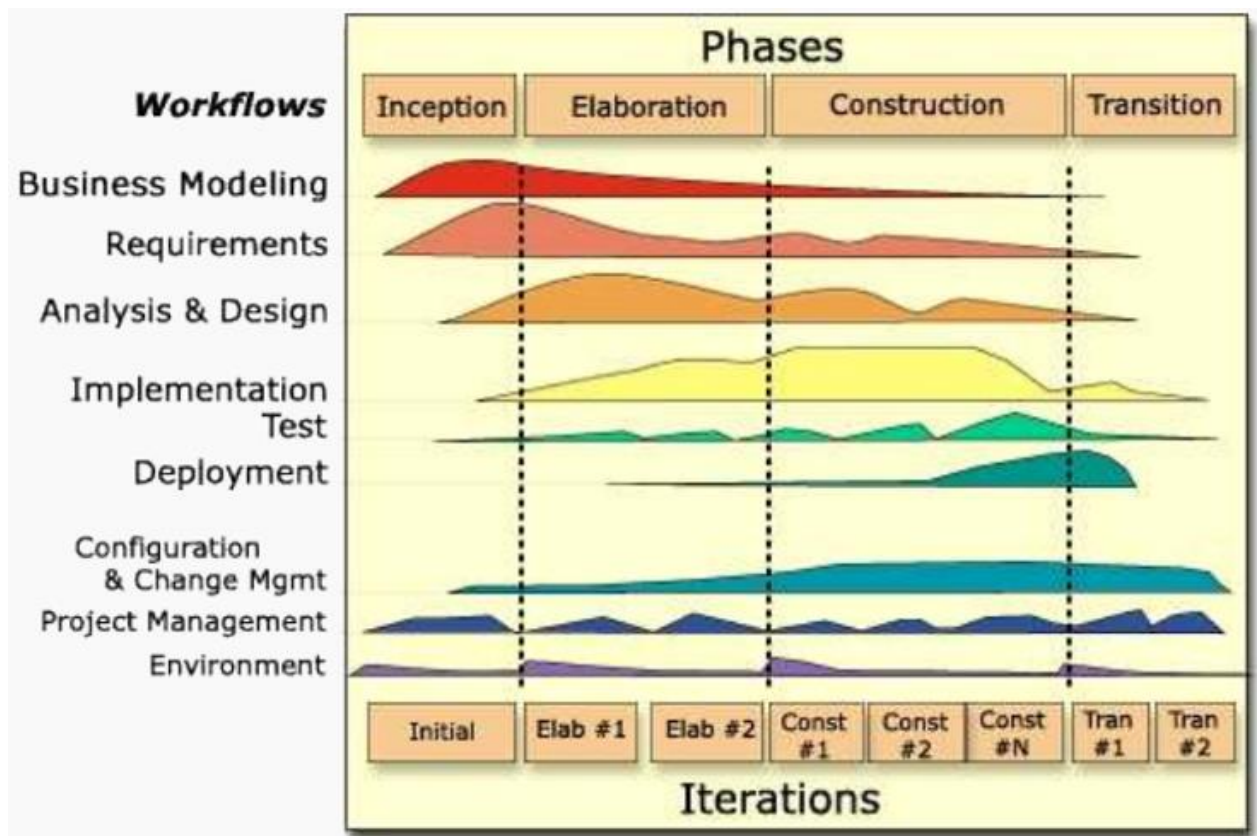
Одна из таких методологий носит название Rational Unified Process. Одноименную компанию зарегистрировали в 90-е годы. Вся разработка представляла из себя инкрементно-эволюционный процесс, в котором все было разбито на фазы. Помимо них, в RUP также имеют место быть т. н. дисциплины. Они представляют собой набор подходов, правил и указаний, которые необходимы для решения той или иной задачи, например, анализ требований. Все дисциплины созданы с целью конкретизировать действия разработчиков для того, чтобы максимально избежать сюрпризов и конкретизировать процесс разработки.

Всего есть 4 фазы: Inception (Начало), Elaboration (проектирование), Construction (построение), Transition (внедрение). В рамках инкрементного подхода фазы могут делиться на итерации. Любая фаза в RUP заканчивается вехами.

В RUP четко определено, как должен строиться весь процесс. Явно описаны все роли, которые принимают участие в создании программного продукта (около 30 ролей).

Роль — группа обязанностей, которую берет на себя определенный участник команды. У каждой роли есть набор деятельности. На входе и выходе деятельности используются, создаются и модифицируются артефакты. Артефактом в разработке принято называть любой результат труда роли, например, диаграмма, документ, программный код и пр. Деятельность, которая производится ролью, четко регламентирована.

В отличие от других моделей ЖЦ, в RUP каждый процесс детально описан и связан с другими процессами. Сам по себе RUP представляет собой программный продукт, оформленный в виде веб-приложения на апплетах Java с гиперссылками.



16. RUP: Фаза «Начало».

Основное направление работ в фазе Начало - оценить проект, понять, сколько на него нужно потратить ресурсов и времени, какие проблемы пользователей и при помощи какой функциональности проект он сможет решить. Формируется артефакт "Концепция" (Vision).

Цели в фазе Начало:

- Определение границ проекта, решаемых и не решаемых им задач, то есть, ограничение области применения разрабатываемого ПО.
- Разработка и описание основных сценариев использования системы. В рамках этой цели необходимо выяснить, как будут использовать систему пользователи, и определить для каждого сценария последовательность действий. Например, какова последовательность действий в сценарии отправки сообщения в социальной сети?
- Предложение возможных технических решений - на основе каких технологий, API, сторонних решений будет разработано ПО.
- Подсчёт стоимости и разработка графика работ.
- Оценка рисков и подготовка окружения, с помощью которого будет проводиться разработка.

Любая фаза в RUP заканчивается вехами. Решение принимается заинтересованными сторонами (stakeholders). На вехе "Lifecycle Objects" заинтересованные лица пришли к

согласию в оценке сроков, первоначальной стоимости, требованиях, приоритетах и технологиях. После этого производится переход на следующую фазу разработки._

17. RUP: Фаза «Проектирование».

На фазе Проектирование основная задача - разработка и тестирование стабильной и неизменной архитектуры системы, создание одного или нескольких прототипов системы, которые определяют исполняемую архитектуру. Исполняемая архитектура - это полностью законченные на базе выбранных технологий несколько характерных функций разрабатываемой системы. Объём ее рекомендуемой реализации определяется на основании внесения новых архитектурных элементов каждым анализируемым требованием, и, как только реализация новых требований перестает создавать дополнительные элементы архитектуры, можно считать, что один из возможных способов построения готов.

Тестирование на этом этапе предназначено для проверки основных нефункциональных требований к системе, например, пропускной способности или времени отклика. На основании архитектуры, прототипов и результатов тестирования уточняются планы разработки и производится переоценка и контроль рисков, уточняются сроки и стоимость системы.

Веха "Lifecycle Architecture" - веха стабильности архитектуры. Контрольные точки на данной вехе исходят из того, что стоимость и сроки разработки согласно требованиям, определенным ранее, соблюдены или могут быть приемлемы с точки зрения заказчика. Контролю подлежат потраченные ресурсы и средства, что важно с точки зрения планирования следующей фазы.

Заказчик может принять и аргументированный перерасход средств, который может возникнуть после разработки архитектуры и проверки её характеристик.

18. RUP: Фаза «Построение».

До фазы Построения должна быть разработана архитектура приложения; внесение в неё кардинальных изменений к этому моменту должно быть исключено. Основная цель на данной фазе - экономически эффективно, с надлежащим качеством и быстро разработать программный продукт.

Во время фазы итеративно и инкрементально проводится анализ, проектирование, разработка и тестирование продукта, происходит создание необходимых выпусков продукта (альфа, бета, ...), предусмотренных планом проекта. Проводятся плановые демонстрации версий заказчику. В конце фазы производится подготовка продукта к передаче заказчику, обследование места установки, разработка учебных материалов и обучение пользователей. Особое внимание уделяется организации и проведению тестирования согласно ранее разработанному плану тестирования, в котором описаны не только проводимые тесты на модульном, интеграционном и системном уровне, но и метрики качества ПО и способы их получения из внутренних систем разработчиков.

На вехе "Initial Operational Capability" принимается решение о возможности внедрения продукта на стороне заказчика. Необходимо учитывать стабильность выпуска (на основании метрик) и готовность пользователей, причём как техническую, так и с точки

зрения обучения. Ещё раз проверяется приемлемость соотношения реальных и запланированных затрат.

19. RUP: Фаза «Внедрение».

Фаза Внедрение предназначена для запуска продукта в продуктивное использование и финального подтверждения пользователями пригодности продукта для их практических нужд.

Если система является "заказной", то есть разработанной для конкретного заказчика, то специальная группа разработчиков занимается установкой и настройкой оборудования и ПО системы. При этом может потребоваться конвертация и перенос данных из старой системы в новую.

Если продукт предназначен для конечного пользователя, который сам будет устанавливать ПО у себя на компьютере, на данной фазе необходимо предусмотреть всё, что связано с будущими продажами продукта. Формирование каналов сбыта, подготовка специалистов по продажам и другие задачи, с которым придётся столкнуться разработчикам.

Цели:

- Провести бета-тестирование, сравнить работоспособность старой и новых версий
- Перенести продуктивную БД, обучить пользователей и поддерживающий персонал
- Запустить маркетинг и продажи
- Отладить процессы устранения сбоев, дефектов, проблем с производительностью
- Убедиться в самодостаточности пользователей
- Провести (совместно со всеми заинтересованными сторонами) открытый анализ соответствия разработанного продукта исходной концепции

Основной вопрос на вехе "Product Release" - удовлетворены ли пользователи? Кроме того, важно провести работу над ошибками, и оценить, что в процессе разработки было хорошо, а что требует исправлений. Особому контролю подлежат затраты на разработку - необходимо оценить их отношение к спланированным в фазе Начало для формирования корректировок в будущих проектах.

20. Манифест Agile (2001).

Способность удовлетворять постоянное изменение бизнес-требований явилось важной предпосылкой появления в 2001 году Agile-манифеста. Декларируется, что "Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества".

«Мы постоянно открываем для себя более совершенные методы разработки ПО, непосредственно этим занимаясь и помогая в этом другим. За это время мы выяснили что:

- **Люди и взаимодействие** важнее процессов и инструментов.

- **Работающий продукт** важнее исчерпывающей документации.
- **Сотрудничество с заказчиком** важнее заключенного согласованного контракта.
- **Готовность к изменениям** важнее следования первоначальному плану.

То есть, не отрицая важности всего того, что справа, мы отдаем предпочтение тому, что слева.»

Манифест во многом повторяет концепцию, названную “Духом RUP”. Дух RUP зародился, как ни странно в Rational Unified Process и описывает отношения к организации процесса и взаимодействия с заказчиком.

Данный манифест в дальнейшем породил другие методологии разработки, как на основе RUP, так и не на его основе (Scrum).

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 различным бизнесом, приносящим доход, или когда заказчик непосредственно покупает время (т. е. рабочие часы) разработчиков.

21. Scrum.

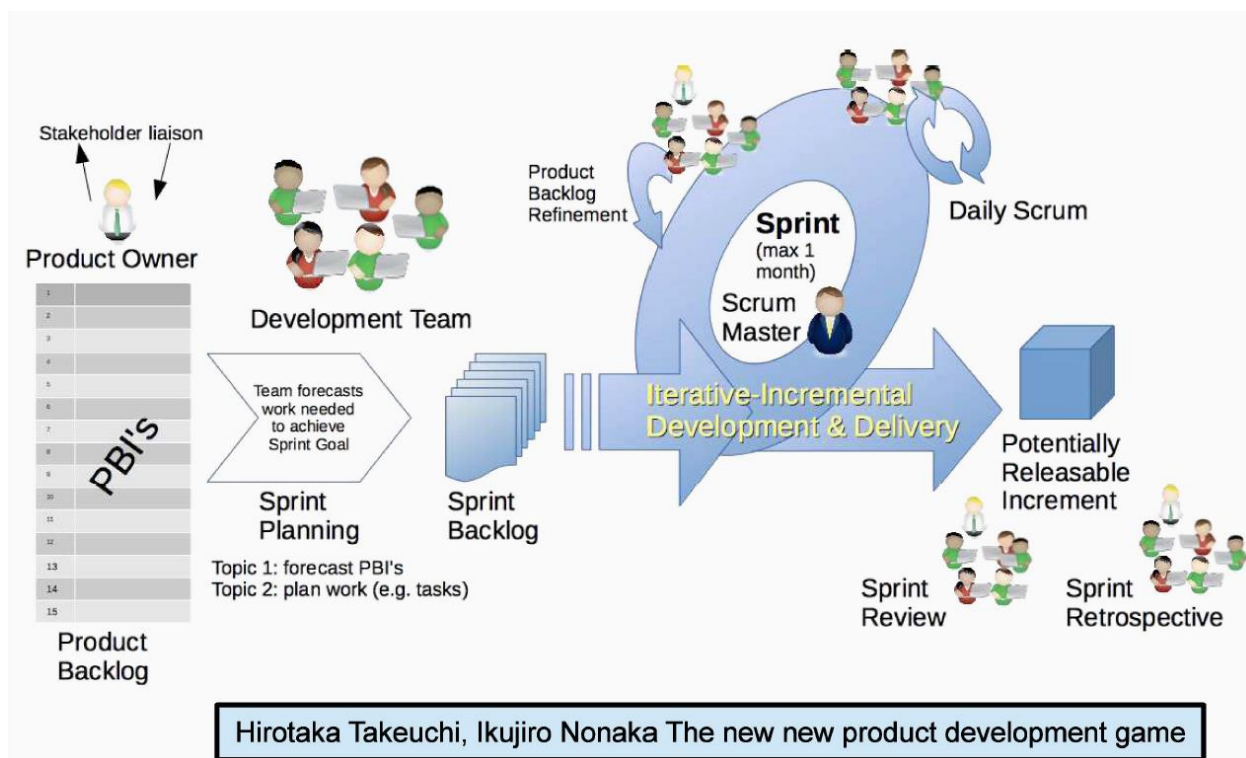
Scrum относится к представителям гибких методологий разработки ПО, разработанным в 1986 году, популярным в нулевые годы. Основой этой методологии является так называемый артефакт Backlog (бэклог — упорядоченный по приоритетам список требований с оценкой трудоёмкости разработки), в который заказчик записывает функциональность, которую он хочет видеть в продукте, отсортированной в порядке приоритетов, задаваемых самим заказчиком, с оценкой трудоемкости их реализации.

Существуют бэклог продукта и бэклог спринта, отличающиеся уровнем детализации. Также имеет место быть артефакт “инкремент продукта”.

Разработка делится на спринты. Спринтом называют период времени от 1-2 до 4 недель. За это время разработчики реализуют выбранный набор требований из бэклога. Каждый спринт заканчивается демо-версией, которую оценивает заказчик. В течение нескольких спринтов производится ретроспектива с целью оптимизации рабочего процесса (перераспределение обязанностей и пр.).

Команда в Scrum не очень большая (3-10 человек), также среди них выделяют особую роль — владелец продукта (Product Owner), он определяет порядок реализаций требований из бэклога (часто, он и является их автором). Для каждого спринта задачи конкретизируются и передаются на разработку команде. Также выделяется роль Scrum Master’а, он ответственен за проведение Scrum-митинга, который помогает команде планировать спринт и следит за внутренними отношениями в команде.

Scrum-митинг проходит ежедневно утром, где каждый участник команды отчитывается о проделанной работе и имеющихся сложностях, а также том, что он планирует сделать к следующей встрече.



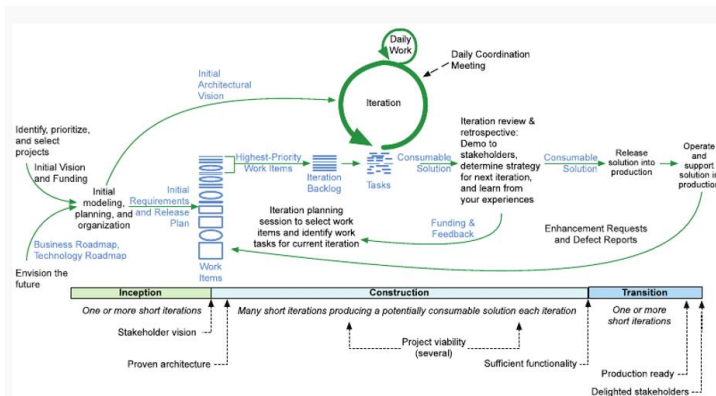
22. Disciplined Agile 2.X (2013).

С введением гибких методологий разработки, очень быстро выяснилось, что они плохо масштабируются на команды с большим числом участников, которые привыкли вставать на работу в 7 утра и не отличаются высокой мотивацией, какой, например, обладают владельцы и участники стартапа.

Disciplined Agile 2.X является относительно свежим решением на рынке и берет по своей сути лучшее из двух миров *UP и Agile процессов. Деление на фазы и дисциплины было частично взято из RUP ("Начало", "Построение" и "Внедрение"), но основной цикл разработки построен на базе гибких методологий (в частности, Scrum).

Помимо озвученного, Disciplined Agile 2.X предлагает процессы, выходящие за рамки процесса разработки, например, управление архитектурой и повторным использованием кода, управление персоналом, непрерывное улучшение процессов разработки и т. д.

DAD активно эксплуатируется компанией IBM.



23. Требования. Иерархия требований.

Требование — условия или возможности, которым должна удовлетворять система. А также это подробное описание того, что должно быть реализовано. Важно, что требование не описывает, как именно оно должно быть реализовано.

Пример требования в ПО: “SEC0 Система должна обеспечивать доступ к любой загруженной в нее информации по запросу федеральных органов власти США”

В RUP требования описываются с помощью шаблона Software Requirement Specification, она же SRS который обычно и является формальным техническим заданием на разработку, а также модели прецедентов (Use Case Model), описывающие требования в виде UML-диаграммы для более наглядного представления.



Обычно у заказчика нет четкого набора требований. У него есть потребности. По своей сути это представляет собой пожелания, которые решают конкретную задачу. К сожалению, заказчик обладает огромным магическим мышлением. Так, например,

заказчик может сформулировать потребность следующим образом: “Мне нужна сковородка, и чтобы на ней не пригорала еда!”

Далее в дело вступает аналитик и в рамках предметной области преобразовывает потребности в набор функций, которые должна реализовывать система. Они все еще не являются полноценными требованиями, но при этом являются куда более содержательными, без потери смысла для заказчика (т. е. они все еще написаны понятным для него языком). Характерный пример функции: “Сковородка должна иметь антипригарное покрытие и термоизолирующую ручку”.

После утверждения набора функций с заказчиком можно переходить непосредственно к формулированию требований, включающих подробное конкретное описание будущего программного продукта. Например: “Диаметр сковородки должен быть 25см, величина бортиков 5см, она должна быть выполнена из сплава алюминия с кремнием марки АК7П (7% кремния).

24. Свойства и типы требований (FURPS+).

Требования должны быть (свойства):

- Корректными
- Однозначными
- Полными
- Непротиворечивыми
- Поддающиеся приоритизации
- Проверяемыми
- Модифицируемыми
- Отслеживаемыми

Для определения последовательности реализации требования должны иметь приоритет. В современной итеративной разработке бизнес-систем приоритет отражает последовательность бизнес-функций и зависит от степени критичности для бизнеса этих функций. Заказчик обычно определяет его самостоятельно.

Любые требования должны поддаваться трекингу, с возможностью в любой момент модифицировать его, в том числе модифицировать приоритет.

В RUP также используется классификация требований FURPS+:

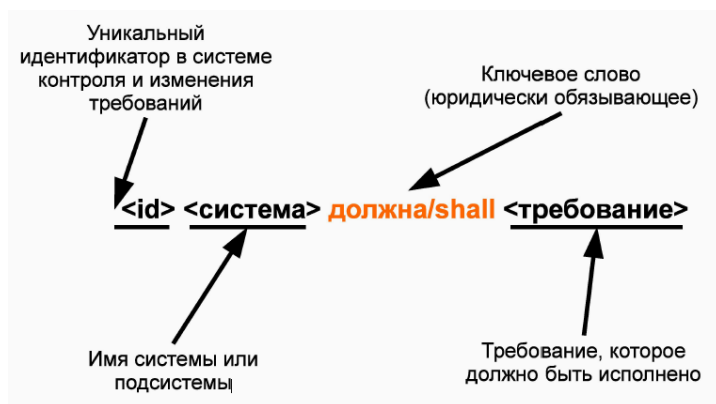
- Functional — определяют, что именно должна делать система (+ часто требования по безопасности)
- Non-Functional — определяют характеристики и ограничения системы
 - Usability — требования по UX
 - Reliability — требования по надежности
 - Performance — требования по производительности
 - Supportability — требования по расширяемости и масштабируемости
- (+) Дополнительные

- Ограничение архитектуры (ограничения на использование тех или иных прикладных программных интерфейсов и продуктов для формирования архитектуры)
- Требования к реализации (математический метод решения задачи) и интерфейсу
- Физические требования (параметры импульсов)

25. Формулирование требований. Функциональные требования.

Требование имеет четкую структуру:

<id> <система> должна/shall <требование>



Требование всегда нумеруется, должно иметь имя системы и, непосредственно, само требование, удовлетворяющее свойствам требований. Возможны разные виды нумерации, обычно в номер стараются включить буквенный идентификатор типа требования, например **FR10** – десятое функциональное требование.

Функциональные требования нужны, чтобы описать, что именно должна делать система. Они описывают возможности и функциональность программного продукта, а также требования по безопасности.

Определяют:

- **Feature sets** - наборы функциональных требований для выполнения конкретной деятельности.
- **Capabilities** - возможности ПО.
- **Security** - Требования к безопасности (обычно описываются отдельно и включают в себя метод аутентификации, список ролей, существующих в системе, шифрование, хранение данных в защищенных источниках. Кроме того, данные требования могут описывать и организационные меры, исключающие доступ злоумышленников к системе.)

Пример: “FR0 Система должна иметь возможность просмотра образовательных статей”

26. Требования к удобству использования и надежности.

В требования по юзабилити обычно входят те особенности использования, которые надо учесть при разработке ПО.

Требования по юзабилити нужны для:

- Учета особенностей пользователя, например для пользователей с ограниченными возможностями обычно реализуют специальные элементы интерфейса.
- Эстетических требований
- Согласованности UI
- Описания справочной подсистемы и пользовательской документации
- Учебных материалов и мастеров ПО для упрощения UX

Пример: U0 “Система должна иметь внешнее оформление в соответствии с brandbook’ом корпорации заказчика”

Требования к надёжности (reliability) - предназначены для фиксирования способности ПО безотказно функционировать в течение определённого периода времени. Отказ системы - это неспособность системы выполнять основную функцию. Кроме отказов, существуют еще и сбои - случаи неспособности выполнять отдельный функционал при сохранении общей работоспособности системы.

В требованиях обычно указывается допустимое число отказов и сбоев за определённый промежуток времени. Часто надёжность системы характеризуются единым интегральным параметром, который называется MTBF - среднее время между отказами. Коэффициент готовности системы - отношение времени исправной работы к сумме времён исправной работы и вынужденных простоев объекта, взятых за один и тот же календарный срок. На рынке сейчас присутствуют программные кластерные системы, которые обеспечивают готовность прикладного ПО с коэффициентом 99,999 (5 минут простоя в год.)

Требования по надежности нужны для:

- Обеспечения взаимодействия со случаями отказа системы
- Способности восстановления продуктивной части системы после отказа
- Точности и предсказуемости ее поведения

Пример: “R0 Система должна обеспечивать максимальное время простоя на уровне 15 минут в год с учетом планового технического обслуживания”

27. Требования к производительности и поддерживаемости.

Требования к производительности (Performance) включают в себя большой набор различных требований. В первую очередь, таким требованием является скорость решения вычислительных задач. Особое значение скорость принимает в системах реального времени, когда необходимо произвести расчеты в точно определенный интервал времени, иначе может быть нарушено функционирование всего объекта управления.

Требования по производительности нужны для:

- Определения скорости и эффективности (процент времени, которое тратится на выполнение полезных задач, по отношению к времени на выполнение общесистемных) решения задач
- Определения готовности системы решить поставленную задачу (реактивность системы)
- Регулирования пропускной способности (какой объём данных или запросов система может обработать за единицу времени), времени отклика, восстановления и т. п.
- А также ограничение на потребляемые ресурсы

Пример: "P0 Система должна рассчитывать время чтения статьи не дольше, чем за 2 секунды"

В сложных корпоративных системах отдельно выделяются требования поддержки (Supportability) программного обеспечения.

Требования по расширяемости нужны для:

- Обеспечение поддерживаемости и совместимости системы (разные ОС, браузеры),
- Возможности адаптировать систему под разные конкретные задачи и масштабировать ее
- Способности конфигурировать систему и локализовать
- Возможность проведения профилактик и обслуживания.
- А также указывает системные требования, требования к установке, ограничения на реальную продакшн платформу

Пример: "S2 Система должна быть совместима с UNIX-серверной вычислительной мощностью"

28. Атрибуты требований.

Существуют разные атрибуты требований:

- Приоритет (правило MoSCoW):
 - MUST - Minimal Usable SubseT — фундаментальное требование, без него продукт не имеет смысла (телефон без возможности звонить)
 - Should have — важное требование, было бы здорово, если бы его реализовали (вызов экстренных служб по кнопке)
 - Could have — не важное требование, потенциально важное требование, к примеру, пользовательское отношение (логичная структура меню телефона)
 - Won't have — эти требования могут быть реализованы в следующих версиях
- И/или приоритет цифрами
- Статус (принято ли требование к исполнению или нет):
 - Предложенные
 - Одобренные
 - Включенные
 - Отклоненные

- Трудоемкость (человеко-часы, попугаи, функциональные точки, use-case points)
- Стабильность — насколько требование не подвержено изменениям
 - Низкая
 - Средняя
 - Высокая
- Целевая версия внедрения (версия, в которую планируется включить реализацию данного требования)
- Риски

С течением времени атрибуты изменяются и требуется периодическая повторная их оценка. На это очень сильно может влиять конкуренция в бизнес-среде, которая и приводит к изменению приоритетов в реализации.

29. Описание прецедента.

При разработке диаграмму прецедентов используют лишь для первого приближения модели. Все прецеденты трансформируются в требования в виде их текстового описания, GUI или иных способов, а получившийся артефакт называют “Описание прецедента”. Например:

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 PieRecommendation. 1.3. Бабушка подтверждает номенклатуру и называет общую стоимость.

Содержит ID, краткое описание, главных актёров (внешних пользователей, взаимодействующих с системой, но не являющихся её частью), второстепенных актеров, предусловий и основного потока, описывающего последовательность действий.

Ключевое слово extend (UML) выражается в виде альтернативного сценария. Существует основной поток событий, где продаются именно те пирожки, и в том количестве, как это было указано клиентом. Если срабатывает условие активизации альтернативного прецедента, то в определенный момент времени вызывается сценарий, который обычно описывает дополнительные действия системы по обработке действий пользователя.

30. Риски. Типы Рисков.

Риск — это фактор и событие, потенциально негативно влияющее на проект. Анализ рисков пришел к нам в сферу программной инженерии из бизнес-сферы и стал основополагающим элементом цикла разработки. Ввел их Б. Бозм, он же автор спиральной модели разработки ПО.

Типы рисков:

- Прямые — ими можно управлять

- Ресурсные: люди, время, финансы, организация. Пример: “Васю бросила девушка и на работу он больше не выходит в течение недели”.
- Бизнес-Риски: конкуренция, подрядчики, убыточность решения. Пример: “Корпорация “Рога и копыта” внезапно выпускает продукт-убийцу, делая бесполезной ваш программный продукт, и ваша команда может смело отправляться искать новую работу”.
- Технические риски: границы проекта, технологические, внешних проектных зависимостей. Пример: “У нас прод упал”, “В библиотеке, которую мы используем, обнаружили уязвимость нулевого дня, караул”, “Петя устроился работать к нам Senior разработчиком, а вместо этого он бежит и ищет себе замену за 70% его зарплаты”
- Политические риски: сферы влияния менеджмента. Например, с появлением нового менеджера могут измениться условия сделки.
- Непрямые — с ними уже ничего не поделаешь
 - Форс-мажор. Пример: “На здание, где по удивительному стечению обстоятельств находилась в этот момент вся команда разработки и поддержки, упал метеорит. Никто не выжил...”

31. Управления рисками. Деятельности, связанные с оценкой.

Оценка риска включает в себя:

- Идентификацию риска
- Анализ риска
- Приоритизацию риска

Как идентифицировать риск? Марвин Дж. Карр предложил строить таксономию источников рисков. По сути своей это дерево, где на первом уровне расположены классы рисков, то, где они непосредственно могут возникнуть (связанные с самой разработкой (Product Engineering); риски, связанные с окружением, где осуществляется разработка (Development Environment); и риски, связанные с программным обеспечением (Program Constraints)), далее в классе определены конкретные риски в виде элементов и у каждого риска есть атрибуты, так и ищем потенциальные источники рисков.

Риск идентифицирован. Как его анализировать? Существуют разные математические модели и их визуализация (например, графическая), однако два основных параметра риска это его вероятность и масштаб (магнитуда) потерь. Так как работа осуществляется с людьми, то данные параметры обычно задаются нечётко, с учетом субъективного отношения человека. Как правило, пятибалльной шкалы оценки параметров бывает достаточно: низкие, незначительные, средние, значительные и высокие вероятность или потери.

Риски проанализированы. Что теперь? Вычисляем т. н. экспозицию риска: произведение вероятности риска на масштаб потерь. Полученные риски сортируем по экспозиции.

Контроль и управление риском:

- Планирование управления/реакции на риски.

- Мониторинг рисков.
- Разрешения неопределённостей, связанных с рисками.

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

Контроль и управление риском включает в себя:

- Планирование реакции
- Мониторинг рисков
- Разрешение неопределенностей

Что можно сделать с риском, который потенциально может наступить? Самое первое и логичное, что может прийти на ум — попытаться избежать риск. Для этого необходимо разработать комплекс мер по отсрочке, исключению или хотя бы уменьшению вероятности наступления риска. Иногда мы можем перевалить этот риск на кого-нибудь еще. Например, запросим дополнительное финансирование у заказчика и тогда он рискует получить проект не в срок. Возможно также пытаться сократить вероятность наступления риска. К примеру, если мы точно понимаем, что данная технология недостаточно опробована, можно попытаться обосновать использование другой, более известной, и, соответственно, менее рискованной в реализации. Ну и четвертый вариант — это принять риск и потери с ним связанные. Такова судьба, а менеджмент лучше распускать.

Во время разработки риски необходимо постоянно переоценивать и вести т. н. мониторинг рисков. Список может быть довольно большим, поэтому контролируем первые топ-10 самых деструктивных и вероятных рисков. Часто применяют в помощь автоматизацию, например, Jira.

Для того, чтобы риск не наступил, нужно строить большое число прототипов системы, использование моделирования, сбор постоянной обратной связи от заказчика и пользователей, так мы будем разрешать неопределенности еще на этапе их зарождения, и минимизируем их вероятности. Также в помощь уменьшения рисков будет непрерывная работа над ошибками, постоянная рефлексия специального отдела контроля качества, и грамотная работа отдела по набору персонала.

33. Изменение. Общая модель управления изменениями.

Изменение это контролируемое, журналируемое обновление системы.

У сложных систем изменения нарастают как снежный ком и их контролирование представляет собой очень серьезную проблему в менеджменте.

Системы контроля версий являются примерами систем учета изменений и позволяют вести журналы. При помощи журнала можно просмотреть историю правок и оценить их влияние на поведение системы. Каждое изменение имеет атрибуты: идентификатор, дату, ответственного, описание, связанный с ним журнал изменения и т.д.

В разработке ПО не только изменения программного кода могут попадать под контроль - отдельному управлению изменениями обычно подлежат сами требования к ПО, выявленные дефекты, артефакты архитектуры, анализа и дизайна.

Все начинается с того, что наш пользователь хочет внести какое-то улучшение в систему или сообщить о какой-то проблеме в ней. Формируется артефакт требований или отчета о проблеме. Все это формирует т. н. запрос на изменение, который порождает запись в логе/журнале изменений системы.

Далее запросы на изменение попадают к менеджеру проекта. Умный менеджер смотрит на запрос и дает ему техническую оценку: насколько изменение осуществимо и сколько это будет стоить, далее специальный комитет на этом основании оценивает: оправдано ли вообще это изменение (например, проблемы одного пользователя часто в расчет не берутся, если же пользователей станет много, тогда изменение будет реализовано с куда большей вероятностью). Все это отражается в журнале изменений.

Если изменение оказалось действительно важным и нужным, оно попадает в отдел по работе с изменениями, которые проведут оценку того, как это изменение влияет на ее другие части. К сожалению, в большой системе невозможно, чтобы изменения в одной части никоим образом не повлияли на другую часть, так что данный вопрос нуждается в серьезном анализе.

После этого мы можем оценить ресурсы, требуемые для осуществления изменения, спланировать процесс изменений и создать график работ. Затем мы реализуем изменения, создаем соответствующие артефакты, в совокупности реализующие требуемые изменения. После этого мы тестируем систему, обновляем документацию, и выпускаем новую итерацию. При этом создаются артефакты отчета о тестировании, новая документация и выпуск новой версии системы.

В завершение изменения передаются проектному менеджеру, который оценивает и верифицирует изменения, и утверждает окончательную версию изменения, закрывая его в журнале.

34. Системы контроля версий. Одновременная модификация файлов.

Сами по себе системы контроля версий требуются для обеспечения одновременного доступа к файлам и программному коду нескольких человек, а также контроль над самими изменениями, которые вносят программисты в исходный код.

Существуют 3 вида систем контроля версий:

- На основе файловой системы. Такие CVS создавали файлы слежения за директорией и такие системы работали только в рамках одной файловой системы. Устаревший подход.
- Централизованные. В данных CVS есть единый центральный узел, с которым разработчики обмениваются изменениями по специальному протоколу, сервер ведет полный их учет. Главный и очевидный недостаток заключается в том, что если центральный узел упал, это автоматически вызывает панику у всего отдела

руководства и программистов, которым не посчастливилось с этим столкнуться.

Пример: Subversion

- Распределенные. В них разработчики делятся между собой полной версией репозитория, каждый может претендовать на роль центральный узла. Обратные изменения попадают после нескольких стадий локальных проверок. Пример: Git.

Главной проблемой CVS является разрешение ситуаций, когда несколько человек работают над одним файлом. И есть 2 подхода по разрешению таких ситуаций:

- Lock-modify-unlock

Тут все просто. Кто первый занял файл и захватил его блокировку, остальные могут, как вариант, сделать копию файла, и работать в ней. Недостатки, это снижает командный дух, если программист сидит за файлом слишком долго. Возможно, потенциальным решением здесь будет работать над файлами ночью. Но это будет работать до момента, пока до столь умного решения не догадается еще один разработчик, и проблема снова встанет столбом.

- Copy-modify-merge

Тут же мы себе копируем весь репозиторий, вольны вносить какие угодно изменения, далее вся команда собирается вместе и в конце рабочего дня задает один главный вопрос: “50 конфликтных мест в файле, как же нам так удалось это сделать и как с этим жить?”. Собственно, при всех преимуществах, решаемых над первым подходом, этот подход даровал нам конфликты при слиянии рабочих копий.

35. Subversion. Архитектура системы и репозиторий.

Централизованная система контроля версий Subversion имеет несколько уровней. Начинается все с централизованного репозитория, который физически может иметь реализации в файловой системе или Berkley DB.

Далее к репозиторию обеспечивается определенный уровень доступа. Он может быть доступен с помощью демона svnserve или с помощью сервера Apache, их возможности эквивалентны.

Удаленный доступ к репозиторию осуществляется по протоколу svn, еще более удаленный доступ (не в рамках одной локальной машины) осуществляется связкой https+svn/ssh+svn.

На том конце провода нас ждет клиент SVN, который не только обменивается файлами с центральным узлом, но и управляет локальной копией файлов.

Репозиторий в Subversion это набор файлов проекта, организованный определенным образом для удобной работы с ним. Каждая фиксация увеличивает версию репозитория на 1. Версия репозитория это всегда целое число. Гарантируется, что при успешной фиксации репозиторий целостный.

- trunk — основная ветвь разработки
- branches — хранение модификаций и версий продукта (релизы, выполнение фич без влияния на trunk)
- tag — функционально целостные изменения (хотфиксы и пр.)

Отдельно хочется упомянуть, что в Subversion нет понятия “ветки”, таковыми здесь выступают подкаталоги, а все файлы копируются целиком при создании новой “ветки”.

36. Subversion: Основной цикл разработчика. Команды.

- svn checkout — получение первоначальной рабочей копии
- Утром: обновить рабочую копию (svn update)
- Днем: вносить изменения (svn add delete mkdir move copy)
- Вечером: просмотреть сделанное (svn diff и svn status). Откатить изменения (svn revert)
- Ночь: загрузить изменения других коллег (svn update), разрешить конфликты, сделать фиксацию (svn commit)

37. Subversion: Конфликты. Слияние изменений.

В Subversion при обновлении копии (svn update) могут возникать конфликты.

Самый частый конфликт — конфликт содержимого файла. Это когда один и тот же файл менялся разными разработчиками (например, по-другому поменяли одну и ту же строку).

Как правило, в этом случае изменение откладывают на потом (postpone), так как редко в этом случае вы можете что-то делать самостоятельно. Следует найти разработчика, с которым возникло конфликтное изменение и согласовать с ним разрешение конфликта. Однако вы можете принять изменения извне, переопределить их своими изменениями, открыть редактор и поменять все руками, запустить внешнее средство и пр.

При выборе postpone создаются три файла. filename.mine - до update — локальный файл), а также filename.rOLDREV и filename.rNEWREV — файл из удаленного узла до и после правок. С помощью triple diff утилит сравнивать эти файлы и разрешать конфликты. Часто есть современные IDE, которые позволяют в интерактивном режиме разрешать конфликты.

Еще из типа конфликтов выделяют конфликты структуры. Например, в основном репозитории перемешали файлы, а в локальной копии были сделаны в них изменения (очень мерзко, согласен). Конфликт решается обновив руками структуру согласно тому, что прилетело из удаленного узла, так и пообщавшись с тем, кто сотворил себе изменения в структуре репозитория (желательно, уволив его после этого).

При работе над параллельными ветками, часто требуется применить все изменения из одной ветки в другую. Для этого существует команда svn merge. Она подгружает изменения из “веток” (каталогов, в svn нет веток) в рабочую копию с учетом изменения

структуры. При слиянии возможны конфликты, которые ничем не отличаются от указанных выше.

38. GIT: Архитектура и команды.

Git — децентрализованная* система контроля версий, придуманная самим Линусом Торвальдсом, поскольку в момент разработки линукса он поссорился с создателем CVS, на которой в тот момент базировалось ядро линукса.

Архитектура гита подразумевает существование всей копии репозитория у каждого разработчика. Каждый разработчик независимо вносит какие-то правки в репозиторий и фиксирует в него же изменения. Если же надо отправить изменения другому разработчику, Git предоставляет удобный интерфейс для их передачи.

Таким образом, каждый разработчик сам выбирает, какие изменения брать, а какие нет, чтобы работать с репозиторием и исходным кодом. Git — это очень гибкая и удобная система контроля версий, сейчас является доминирующей на рынке.

В git book выделяют три основных сценария рабочего процесса с Git'ом:

- Централизованный рабочий процесс: есть централизованный репозиторий (например, на GitHub'е), все разработчики синхронизируют работу именно с ним.
- Рабочий процесс с менеджером по интеграции: каждый разработчик имеет канонический локальный репозиторий. Каждый открывает к нему доступ для остальных только для чтения, правки вносятся в локальном репозитории, потом запрашивается у менеджера по интеграции слияние правок в его каноническую репозиторию.
- Рабочий процесс с лейтенантами и диктатором. Самый известный пример: разработка ядра Linux. Лейтенанты — это интеграционные менеджеры: каждый отвечает за свою конкретную часть. У лейтенантов есть свой интеграционный менеджер — диктатор, его репозиторий считается эталонным, он принимает окончательное решение о внесении правок.

Чтобы начать работать с git, вам пригодится команда `git init`. Или можно клонировать репозиторий по его URL командой `git clone`.

В git имеется три зоны, где может находиться файл: `unstaged (modified) area`, `stage area`, `committed area`. Переход от `unstaged area` в `staged area` достигается командой `git add`. Обратный переход тоже возможен, для этого в помощь команды `git restore --staged` или `git reset`.

Переход из `staged area` в `committed area` достигается командой `git commit`. После фиксации мы можем просто поменять файл для его перехода в `unstaged area`, чтобы вернуть его в `staged area`, мы можем отменить коммит, переместив указатель HEAD на один коммит назад (`git reset --hard`)

После того, как файл вышел из `modified area`, изменения навсегда останутся в git, и они никуда не исчезнут без стороннего вмешательства в .git каталог. Это является большим преимуществом Git'a, его архитектура серьезно уберегает нас от нечаянного удаления

файла и других нелепых ситуаций. Посмотреть состояние репозиторий можно командой `git status`.

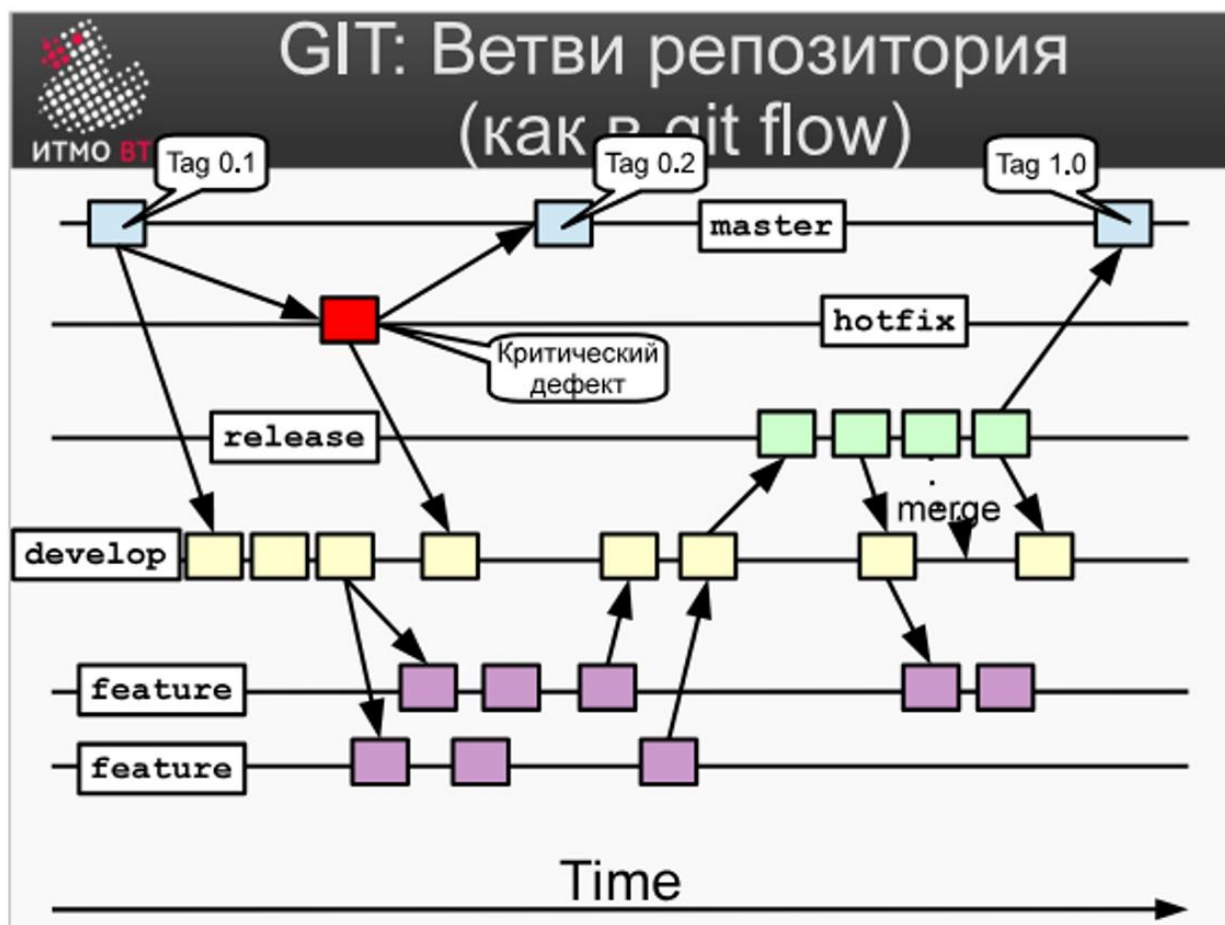
У каждого коммита есть хеш, имя автора коммита и его e-mail. Это важно, поскольку ветки в `git` это не больше, чем указатели на коммиты, а сами по себе коммиты складываются в граф древовидной* структуры. В отличие от `svn`, создание новых веток в гите это не больше, чем сделать файл с указателем на 40 символов хеша.

При работе с ветками, `git` предоставляет удобную команду `git branch` (`git checkout`), позволяющую создавать новые ветки или переключаться между существующими. Ветки можно сливать одну в другую, командой `git merge`.

Также для централизованной работы, например на сервисе GitHub вам будут полезны команды `git fetch`, `git pull` (она по сути делает `git fetch origin/<branch>` + `git merge <branch>`).

Всю древовидную структуру репозитория можно получить командой `git log (--graph)`, сравнить два коммита можно командой `git diff`.

39. GIT: Организация ветвей репозитория.



В Git создавать ветку принято на почти любое самостоятельное изменение кода. Ветки в гите легкие — это просто указатель из 40 символов, являющиеся ничем иным как хеш коммита в древовидной структуре репозитория.

При создании репозитория, автоматически создается ветка `master` (`main` с 2020 года), которая является как правило эталонной в нашем репозитории. Далее как правило происходит ответвление в ветку `dev`, где происходит основной процесс разработки, в `master` как правило лежит только релизный образец исходного кода (на помощь здесь приходит CI/CD). Каждая фича принято, чтобы разрабатывалась в отдельной ветке.

Когда накапливается достаточно изменений, все ветки собираются воедино, происходит слияние, решение конфликтов, тестирование, и из новых фич собирается релиз, который и является новой вехой в разработке программного продукта. Ветвь `hotfix` иногда бывает нужна, чтобы оперативно поправить критические баги, которые по каким-то причинам не были покрыты тестированием (человек не идеален). Изменения оттуда напрямую коммитятся в мастер и создают новый релиз, пусть и меньшей итерации.

Процесс повторяется для всех новых функциональных требований.

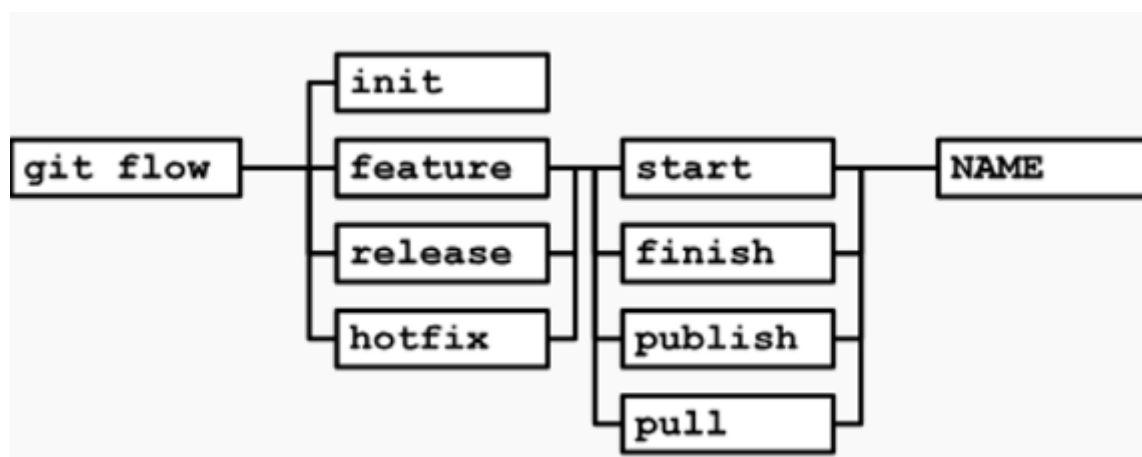
40. GIT: Плагин `git-flow`.

Каждое проделываемое действие в гите может быть достаточно громоздким, поэтому над гитом была создана абстракция, которая получила название `git flow`. Это по своей сути плагин, который предоставляет интерфейс гита в терминах версий, а не операций.

Для использования `Git flow` необходима определённая начальная настройка репозитория. Для этого у `Git flow` есть функция `init`, которая при старте задаёт пользователю несколько вопросов и производит необходимую настройку.

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.

Структура команд `git flow`:



Все это позволяет пользоваться репозиторием Git на более высоком уровне, независимым от синтаксиса git образом.

41. Системы автоматической сборки: предпосылки появления

№1: рутинный процесс сборки.

При сборке необходимо повторять одинаковую последовательность команд:

```
javac -d build -cp lib src/*.java
```

```
jar -cfm ...
```

Как вариант, можно сделать скрипт. Однако он будет

- Платформенно-зависимым
- Очень большим и иметь перспективы стать еще больше

№2: зависимость от конкретной платформы

- Собранная программа под одну машину не будет работать на другой
- Собранная программа под одну операционную систему не будет работать на другой

Например, серьёзные отличия существуют в системах с разной разрядностью, что влечет за собой различия в размере минимальной единицы информации, с которой можно производить арифметические и иные действия, что влияет не только на быстродействие программы но и на необходимость генерирования другого кода для целевой системы. Кроме того, нужно учитывать, где размещается конфигурация системы, которая используется системными службами. Например, локальный файл соответствия IP-адресов именам машин в Ubuntu Linux находится в /etc/hosts, в Solaris в /etc/inet/hosts, И прочие проблемы.

№3: медленная сборка

- Сборка может занимать сутки, файлов с исходным кодом может быть тысячи, они могут быть большие.
- Не обязательно компилировать **все** файлы: давайте только те, которые были изменены с учетом их зависимостей.
- Нет параллелизма — и современное оборудование простаивает

42. Системы сборки: Make и Makefile.

Make считается средством сборки низкого уровня. Представляет из себя стандартную утилиту, включенную в Unix-подобные системы. Суть в следующем. Программисту надо написать файл сборки (makefile), в котором надо описать опции компиляции и зависимости, которые будут резолвиться при сборке, а также пути к стандартным

компиляторам, средствам упаковки и прочие команды для терминала, которые будут исполняться в рамках одного таргета.

Синтаксис `makefile`'а выглядит примерно следующим образом:

```
<variable_name> = <variable_value>

target: dependencies
    actions
    ...
```

Когда мы запускаем утилиту `make`, она смотрит наличие `makefile` в текущем рабочем каталоге (`pwd`), смотрит в этом мейкфайле наличие таргета по-умолчанию (`all`), далее она строит и пробегает по графу зависимостей и с учетом времени модификации производит последовательность действий, необходимых для сборки конечного продукта.

Таким образом, `Make` — это язык, который в неявном виде позволяет определить последовательность действий при сборке, а также императивно указать последовательность действий, необходимую и достаточную для того, чтобы произвести сборку продукта. К сожалению, ее простота не совмещается с удобством, программисту нужно держать в голове достаточное количество макропеременных и действий по умолчанию, определенных в системе.

43. Системы сборки: Ant. Команды Ant.

`Apache Ant` является императивным средством сборки программ, написанных на языке `Java`. Весь процесс управляется при помощи файла `build.xml`. Его структура выглядит примерно следующим образом:

```
<?xml version="1.0"?>
<project name="Hello" basedir=".">
    <property file="build.properties"/>
    <target name="init">
        <mkdir .../>
    </target>
    <target name="compile" depends="init">
        <javac .../>
    </target>
    <target name="build" depends="compile">
        <jar .../>
    </target>
</project>
```

`XML` это не лучшее решение для императивного описания сборки. Однако он очень хорош для машинной верификации, поэтому разработчики `Ant` решили использовать именно его. Идейно, `Ant` очень похож на `Make`, оба представляют системы сборки одного класса.

Имеется возможность вызывать цели (`antcall`) явно, не через атрибут `depends`. Эти цели могут быть объявлены в другом файле, поэтому присутствует возможность разбивать сборку на модули.

Имеется поддержка неизменяемых значений — свойств. Задаваться могут как в build.xml, так и в переменных окружения среды, так и в .property файлах. Подстановка переменной \${...}.

Некоторые команды из Ant:

- checksum — создает контрольную сумму для файлов
- chmod — изменить права доступа к файлу
- copy, delete, mkdir, move — команды файловой системы
- javac, java, jar, javadoc — команды JDK
- exec, sleep, waitfor — команды исполнения среды
- echo — вывести в лог сообщение
- scp, ftp — команды передачи по сети
- junit, junitplatform — команды платформы для запуска тестов (JUnit 4 & 5)
- bzip2, tar, war, zip — команды архиваторов

Многие из этих команд зачем-то повторяют стандартные команды в операционной системе. Однако не все так просто, эти команды поднимают нас на уровень выше, делая процесс сборки независимым от конкретной платформы. Везде, где запускается Apache Ant, запустится процесс сборки с исходным build.xml. За это дружно благодарим “истинную” кроссплатформенность джавы.

44. Системы сборки: Ant-ivy.

В связи с развитием интернета (и огромным числом легаси проектов, которые по объективным причинам имели Ant скрипты сборки на 10+ МБ), Ant смог заполучить себе менеджер зависимостей. Сторонние библиотеки встраиваются в сборку автоматически, и загружаются из центрального репозитория (Maven), а менеджерит всем этим хозяйством утилита Ivy - менеджер зависимостей для Ant

Таким образом, это существенно упрощает работу с third-party библиотеками, а процесс их встраивания в build.xml очень простой и понятный:

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

Зависимости можно задать в файле ivy.xml:

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

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

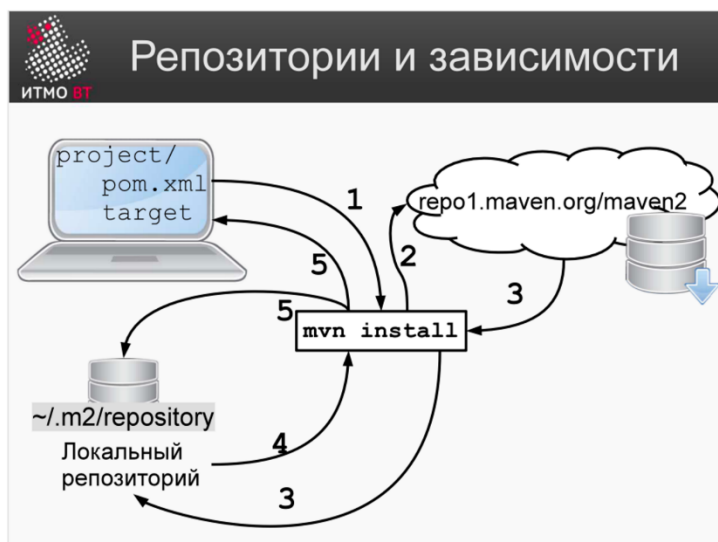
Maven является логическим продолжением автоматических средств сборки проекта. Императивные средства хороши до того момента, пока мы в состоянии проследить полный цикл сборки программы, но для больших проектов это не всегда выполнимая задача. Часто мы хотим сказать: что именно собирать, а не как это надо делать. Т.е. нужны системы, которые знали, как надо собирать исходный код.

Так появились декларативные средства сборки, и Maven является одним из его представителей. Maven представляет декларативную систему сборки программ, написанных на языке Java. С другой стороны, у нас появилось развитие интернета и система репозитория, где хранятся все сторонние библиотеки на все случаи жизни, и мы указав, что хотим видеть их в нашем проекте, просто получаем их из репозитория и пользуемся ими.

Мы описываем так же, как и в Ant'е, в XML-файле. Только в отличие от Ant'овского XML, в этом мы описываем объектную модель проекта (Project Object Model — POM). Она содержит в себе

- Имя, версию и тип проекта
- Местоположение исходного кода
- Зависимости (сторонние, внутренние)
- Плагины — средства, используемые во время разработки (они дополнительно знают, как работать с исходным кодом, чтобы его декларативно собрать)
- Профили — альтернативные конфигурации проекта

Репозитории и зависимости принципиально имеют следующую схему:



Указывая в `pom.xml` зависимости (1), мы даем знать утилите Maven информацию о том, за какими внешними библиотеками надо обратиться к центральному узлу (2). Получив стороннюю библиотеку определенной версии (3), она загружается в локальное хранилище (3). Далее происходит сборка (4) с учетом предоставленного кода, и готовый программный продукт помещается в каталог `target` (5).

46. Maven: Структура проекта. GAV.

Структура проекта Maven в виду декларативных ограничений фиксированная:

target — рабочая и целевая директория

src/main — основные исходные файлы

src/main/java — каталог исходных файлов Java

src/main/webapp — раздаточное содержимое web-страниц

src/main/resources — встраиваемые ресурсы, не нуждающиеся в компиляции

src/test — исходные файлы тестов

src/test/java — каталог исходных файлов тестов Java

src/test/resources — встраиваемые тестовые ресурсы, не нуждающиеся в компиляции

Система наименования модулей в Maven строится по принципу GAV — GroupID:ArtifactID:Version. Пример:

```
<project>
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>6.0.0</version>
</project>
```

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

В Maven транзитивные зависимости. Это означает, что если сторонняя библиотека зависит от еще одной библиотеки, то исходный проект тоже будет зависеть от этой библиотеки.

Зависимости состоят из:

- GAV-описания
- Scope
 - compile — зависимая библиотека будет участвовать в компиляции и дальнейшей загрузке
 - provided — зависимая библиотека будет участвовать при линковке, но не будет включена в итоговую сборку. Используется в энтерпрайзе, где сервер приложений часто уже обладает нужными зависимостями самостоятельно.
 - test — используется только в тестовом окружении
 - Иные: runtime, system
- Type:
 - jar
 - pom
 - war
 - ear

- zip

Зависимости со скоупом `compile` и типом `jar` являются зависимостями по умолчанию.

Жизненный цикл Maven-приложения:

- `generate-sources/generate-resources` — эта фаза сгенерирует недостающие исходные файлы, если есть соответствующее правило DSL
- `compileJava` — скомпилировать основные исходные файлы
- `testCompileJava` — скомпилировать тестовые файлы
- `test` — протестировать исходный код
- `package` — собрать скомпилированные файлы в пакет (в `Jar` или `War` архив)
- `integration-test (if any, pre and post)` — запустить интеграционный тест
- `install` — установка в локальном репозитории
- `deploy` — если есть сервер приложений, загрузить собранный артефакт туда

Плагины в Maven управляют процессом сборки программы. Они знают, как именно собирать исходный код. И они встраиваются в определенный этап ЖЦ, например, `packaging` плагины могут выступать способы упаковки кода в `jar` или `war` архив.

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

Gradle — современная система сборки.

Основными достоинствами его являются:

- Нейтральность к языкам программирования. В отличие от Maven, который предназначен, в основном, для сборки Java-приложений, gradle не ограничивает в языках программирования.
- Преемственность описания зависимостей и возможность использования настраиваемых репозиторий зависимостей, в том числе от Maven, используя различные нотации описания.
- Использование понятного человеку DSL (Domain Specific Language) для описания сборки, который сам по себе разработан и использует возможности Groovy.
- Инкрементальная и параллельная сборка - позволяет запускать плагины только при наличии действительной необходимости выполнения задач по сборке, и экономит время сборки проекта. Инкрементальная сборка существует в различных системах сборки, но, например, в Maven все равно тратится достаточное количество времени на выполнение плагина, даже если собирать нечего.

Пример `build.gradle.kts`:

```
plugins {  
    kotlin("jvm") version "1.9.23"  
}  
  
group = "smth"
```

```

version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain(17)
}

```

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

Имеется довольно огромное число платформенно-зависимого кода, с которым нам приходится иметь дело. И как способ выручить нас на помощь приходит утилита GNU autotools. **GNU autotools** основаны на макропроцессоре общего назначения **m4**, существовавшего с момента написания первых версий **Unix**. **GNU autotools** полностью платформонезависимы.

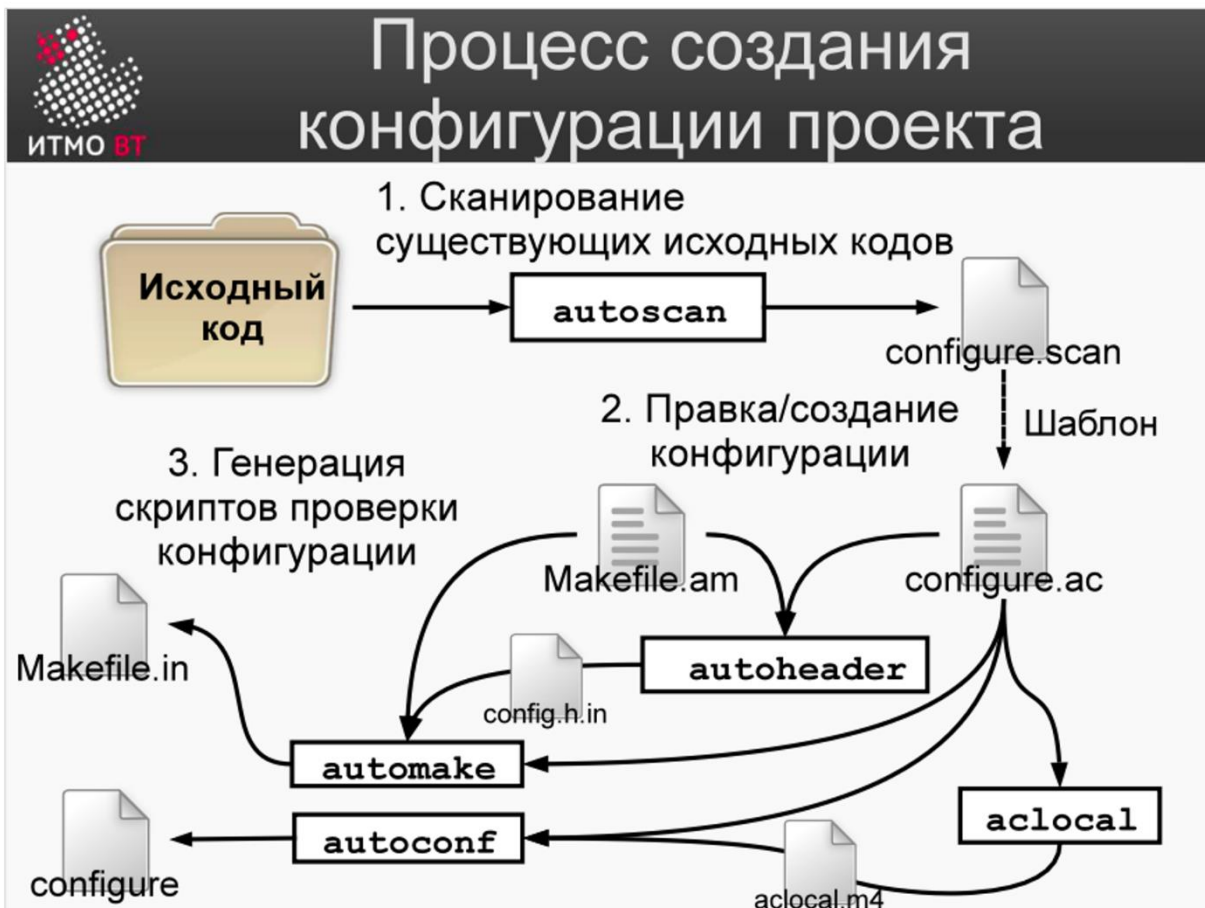
Мы имеем огромную базу кода, многих из которых является платформеннозависимым. С помощью утилиты autoscan мы проводим диагностику и сканирование исходных кодов на предмет таких платформеннозависимых вызовов, и на выходе получаем файл configure.scan. Далее нам нужно доработать эти файлы вручную (configure.ac).

Также мы создаем makefile.am, где в явном виде указываем названия исполняемых программ, из каких исходников они должны быть собраны, а также зависимости. Такой файл создается в каждом подкаталоге базы исходного кода.

Далее мы пробегаемся утилитами autoheader, который создаст шаблон заголовочного файла, а команда aclocal проверит, что установлено на компьютере разработчика, и посмотрит, что из этого используется в проекте.

Затем утилитами automake и autoconf создаются конечные makefile.in и configure файлы, которые уже будут использоваться в процессе конфигурации на конечной машине.

Принципиальная схема:



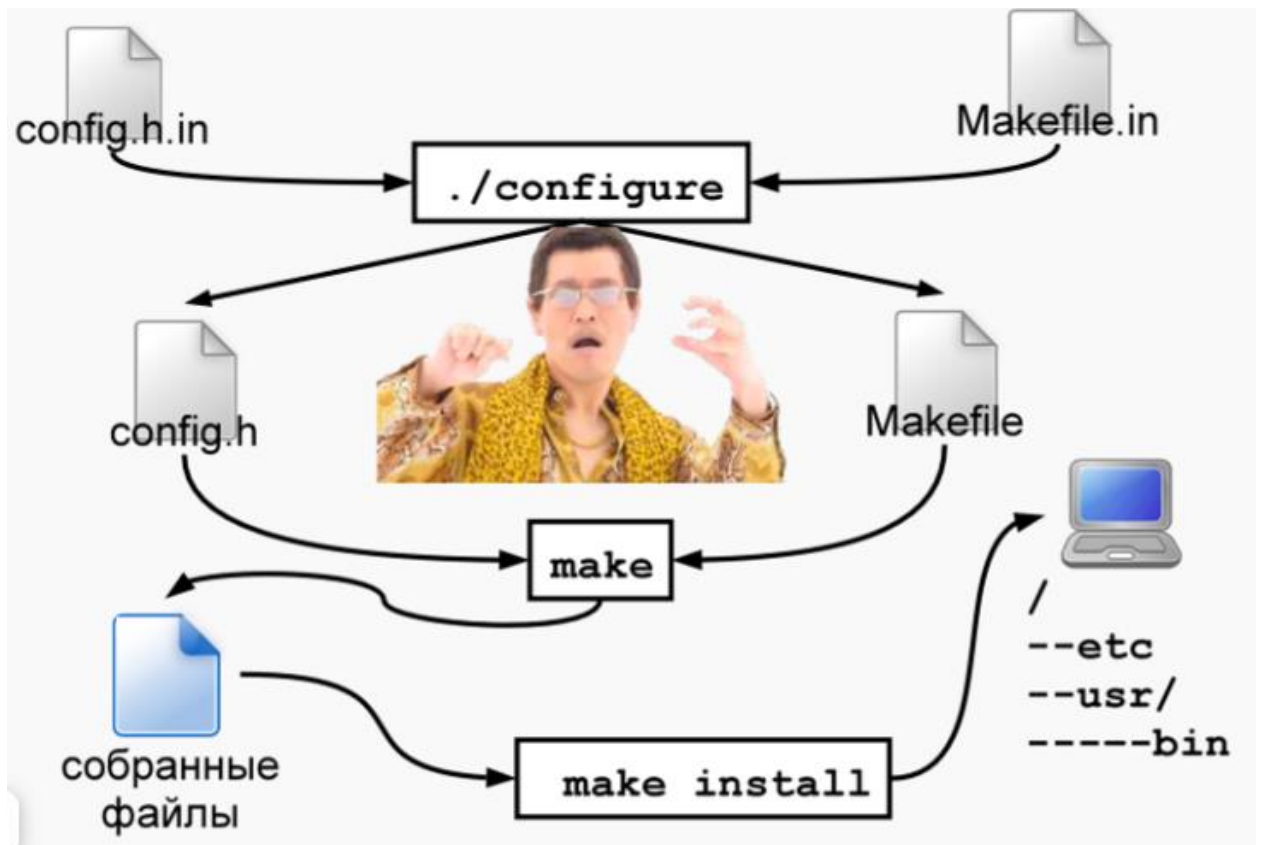
50. Системы сборки: GNU autotools. Конфигурация и сборка проекта.

Имеется довольно огромное число платформенно-зависимого кода, с которым нам приходится иметь дело. И как способ выручить нас на помощь приходит утилита GNU autotools. **GNU autotools** основаны на макропроцессоре общего назначения **m4**, существовавшего с момента написания первых версий **Unix**. **GNU autotools** полностью платформонезависимы.

На этапе создания конфигурации мы получили исполняемый файл `configure`. При его запуске на конечной системе будет произведена проверка наличия нужных исполняемых файлов (компиляторов и пр.). На выходе мы будем иметь платформеннозависимый `makefile` и конфигурационный файл системы или информацию о том, что каких-то компонентов не хватает, и их нужно доустановить.

Далее происходит обычная работа с `make`: `make all + sudo make install` установят платформеннозависимый софт на конечную систему.

Принципиальная схема:



51. Сервера сборки/непрерывной интеграции.

Для выполнения автоматической сборки существуют специальные автоматизированные сервера, политики непрерывной интеграции. Их основное назначение - сборка новой версии продукта при наступлении заданных администратором или разработчиком условий. Они очень гибкие в своей настройке и позволяют делать практически что угодно.

Таковыми условиями могут быть новое обновление исходных кодов в ветке "мастер". Собираем проект, генерируем артефакты, документацию, бинарные файлы для целевой системы, запускаем тесты, создаем отчеты, рассылаем их по электронной почте. Политика запускается каждый день в 00:00, после окончания рабочего дня.

Теперь мы можем с утра открыть отчеты, увидеть, как у нас все успешно собралось и мы можем с этим как-то работать.

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

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

Примеры: CruiseControl, Jenkins, Travis CI, GitLab CI/CD.

52. Основные понятия тестирования. Цели тестирования.

Основные понятия и цели тестирования задаются международной классификацией квалификации тестировщиков (International Software Testing Qualification Board - ISTQB)

Понятия:

- Mistake - ошибка разработчика. Человеческое деяние, которое в конечном итоге привело к получению неверного результата. В широком смысле – непреднамеренное отклонение от истины или правил.
- Fault - дефект, изъян. Неверный шаг в алгоритме (или неверное определение данных) в компьютерной программе. Следствие ошибки, потенциальная причина неисправности.
- Failure - неисправность, отказ или сбой - наблюдаемое проявление дефекта, в том числе, крах или падение программы.
- Error - невозможность выполнить с использованием программы задачу, получить верный результат.
 - BUG — неформальное использование. Может обозначать: дефект, изъян, сбой, невозможность выполнить задачу, что-то другое или вовсе ничего не обозначать.

Цели:

- Основная цель: повысить уровень пользовательского доверия к вашему программному обеспечению, дать пользователю понять, что Ваша программа работает корректно во всех необходимых обстоятельствах.
- А также:
 - Обнаружение и предотвращение дефектов
 - Предоставление информации для принятия решений

Выполняя тестирование, разработчик предлагает пользователю на основании объективных фактов (проведения набора тестов, выбора тестового покрытия для каждой части программы) поверить в то, что разработчик использовал все разумные средства, чтобы показать присутствие и исправление (а не отсутствие!) дефектов в продукте.

Невозможно гарантировать и доказать полное отсутствие ошибок в программе даже после тестирования.

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

Понятие "тестовое покрытие" включает в себя то, насколько код приложения покрыт тестами, которые могут находить известные и потенциальные дефекты. Полное тестовое покрытие подразумевает покрытие тестами всего кода и всех возможных вариантов развития событий, возможных в данном коде. Полное тестовое покрытие, если и достижимо, то ценой очень больших затрат.

Например, для проверки корректности работы умножителя двух 32 битовых чисел нам потребуется протестировать всевозможные результаты умножений для всех вариантов чисел А (2^{32} чисел) и В (2^{32} чисел) $\rightarrow 2^{64}$ вариантов. Пусть мы получили каким-то (альтернативным способом) тестовую базу на 2^{64} записей для каждого варианта умножения. Пусть эта запись содержит два числа А, В (32 бита каждый), ожидаемый

результат (64 бита), а также результат проверки теста (8 бит). Итого это 132 бита на каждую запись. Всего таких записей $2^{64} \rightarrow \sim 3.13$ петабайта будет занимать таблица с полной тестовой базой для нашего умножителя.

Пусть мы нашли хранилище для этого бессмысленного и беспощадного занятия. Пусть у нас одна проверка будет занимать 1 такт машинного времени. Тогда нам надо будет прогнать 2^{64} тактов на вычислительной технике. Пусть нам дан процессор на 3ГГц. Тогда проверка нашего умножителя будет занимать приблизительно 180 лет. И это речь идет о достаточно примитивной схемотехнике и алгоритме.

Считается, что полное тестовое покрытие недостижимо.

54. Статическое и динамическое тестирование.

Статическое тестирование - это тестирование, не связанное с запуском набора тестов, разработанных для ПО. Оно включает в себя методики по рецензированию и инспекциям кода. Его суть заключается в том, что нам вовсе необязательно иметь рабочую архитектуру и рабочий код + разработанный набор тестов для того, чтобы выполнить тестирование. Статическому тестированию можно подвергнуть не только программный код, но и другие артефакты, например, спецификацию. Проводится в ручном и автоматизированном режиме, например:

- Ручное статическое тестирование:
 - Звонок (опытному) другу — “Помоги найти мне ошибку...”
 - Инспекция — Несколько человек, которым дали простую работу, будут делать ее хорошо
- Автоматическое статическое тестирование:
 - Линтеры
 - Автокоррекция в ворде

Динамическое тестирование требует уже созданной программной архитектуры (осуществляется сборка и запуск модулей, групп модулей или всей системы). Однако недавно был предложен подход к разработке при помощи предварительного создания тестов (Test Driven Development). При этом на основе требований определяется тестовое покрытие, и разрабатываются тесты, которые первоначально все выполняются со сбоями. Разработка считается завершённой, когда все тесты выполняются успешно.

55. Автоматизация тестов и ручное тестирование.

Автоматизация тестов это хороший способ избавиться от монотонной рутинной работы ручных тестировщиков, но часто бывает такая ситуация, что у компании на это попросту не хватает ресурсов, и поэтому мы можем нанять максимально неквалифицированный персонал, который сможет нам штамповать ручные тесты пачками. Тем не менее, чем больше компания и ее продукт по размерам, тем больше нужно включать процесс автоматизации в тестирование.

Отдельный вид тестирования - это т.н. регрессионное тестирование, которое заключается в том, что при изменении программы запускаются старые тесты. Это позволяет проверить, не повлияли ли внесённые изменения на работу всей программы.

Чтобы автоматизировать тесты и пользоваться регрессионным тестированием, необходимо обеспечить однозначное повторение тестового сценария (одинаковые входные данные должны порождать одинаковые выходные данные).

Важной является проверка одного и того же приложения в разных окружениях (т.н. тестирование совместимости). Необходимость повторять тесты в различных окружениях порождает большую сложность всего процесса. Необходимо выполнить много тестов, поэтому чем меньше будет сформировано изначальных функциональных тестов, тем меньше общая сложность и длительность выполнения таких тестов.

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

Выделяют два классических подхода к тестированию:

- Метод “черного ящика”. Исходные данные для тестов берутся из спецификации, требований, информации об архитектуре. Взаимодействие идет через публичный интерфейс системы и мы совершенно ничего не знаем о том, как это работает под капотом
- Метод “белого ящика”. Исходные данные опираются на основании анализа исходного кода. Исходный код анализируется и представляется в виде алгоритмического графа. После этого определяется цикломатическая сложность программы, которая указывает максимально необходимое число тестов, которые нужно выполнить для полной проверки программы. Конкретные тесты выбираются так, чтобы покрыть все пути полученного графа.

Дополнительно источниками данных для тестов могут быть модели (UML) и жизненный опыт тестировщика, который может заранее иметь на уме потенциально опасные места.

Роли и деятельности в тестировании делятся на несколько больших категорий:

- Проектирование тестов
 - Высококвалифицированные специалисты, разбирающиеся в предметной области, дискретной математике и программировании
- Автоматизация тестов
 - Непосредственно разработчики тестов. Принято, что модульные тесты пишут сами разработчики исходного кода, а интеграционные и системные тесты пишут отдельно выделенные для этого программисты
- Исполнение тестов
 - Исполнители тестов это очень аккуратные и внимательные люди, выполняющие работу четко по инструкции, которую им разработали проектировщики. Не требуют особой квалификации и их задача просто аккуратно провести тестирование системы.
- Анализ результатов
 - Аналитики тестовых результатов должны быть ознакомлены с предметной областью, в которой они работают, а также ряд технических знаний и их задача вразумить в себя полученные на прошлом этапе результаты тестирования.

57. Понятие тестового случая и сценария.

Тестовый случай представляет из себя набор входных тестовых значений, преобразователя тестовых данных (наша программа) и набора выходных значений (ожидаемый результат).

Input → Processing → Output

К значениям относятся данные + условия. В ожидаемом результате описано то, что мы ожидаем увидеть в качестве выходных данных, постусловий + иные последствия и/или изменения, которые происходят в программе.

Ожидаемый результат обязан быть определен до начала теста, иначе правдоподобные, но некорректные результаты будут засчитаны. Особенно здесь стоит подход TDD, при котором тесты определяются заранее и разработка считается завершенной тогда и только тогда, когда успешно выполнены все тесты.

Тестовый случай должен быть повторяем, как следствие, автоматизируемым. То есть при повторном запуске тестового случая мы не должны рассчитывать на изменение результатов. Это особенно важно, когда речь идет о регрессионном тестировании.

Тестовый случай должен учитывать состояния. Это расширяет тестовое покрытие и дает возможность классифицировать тесты по состояниям и собирать из этого тестовые сценарии.

Тестовый сценарий это последовательность тестовых случаев.

Пример: Банкомат, типичное использование системы.

// Начальное состояние // Ввод // Действие системы // Вывод // Итоговое состояние

1 // Готов // Пользователь вставляет карточку // Успешное чтение карты // Приглашение ко вводу PIN-кода // Ожидание PIN

2 // Ожидание PIN // Пользователь вводит верный PIN // Успешная проверка PIN-кода // Приглашение к действию с картой // Ожидание действия

3 // Ожидание действия // Пользователь выбирает опцию "Снять с карты 5000 рублей" // Проверка баланса, пересчет денег // Выдача 5000 рублей // Выдача денег и карты

4 // Выдача денег и карты // Пользователь забирает деньги и карту // Завершение выдачи // Благодарность за использование // Готов

Тестовый сценарий должен помимо этого отрабатывать некорректные или ошибочные варианты:

// Начальное состояние // Ввод // Действие системы // Вывод // Итоговое состояние

3 // Ожидание действия // Пользователь выбирает опцию "Снять с карты 5000 рублей" // Проверка баланса, пересчет денег // Выдача 5000 рублей // Выдача денег и карты

4 // Выдача денег и карты // Пользователь оставил деньги и карту в банкомате в течение 45 секунд. // Возврат денег на счет и в банкомат, блокировка карты // Сообщение об ошибке, информирование о необходимости посещения банка // Готов

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

При выборе количества тестов надо ориентироваться на следующий баланс:

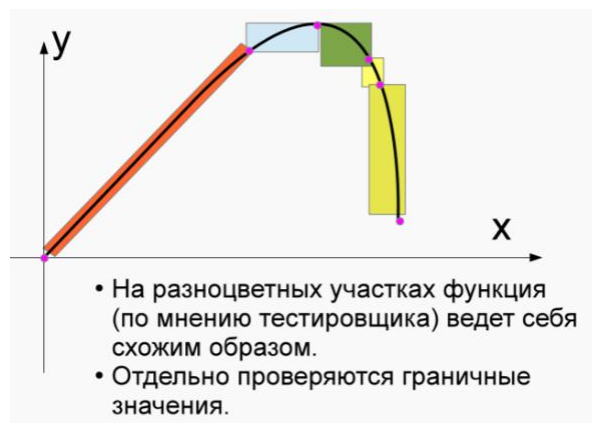
- Больше тестов → больше тестовое покрытие → выше качество и доверие
- Меньше тестов → продукт разрабатывается быстрее → продукт быстрее выйдет на рынок

Зачем нужно делать больше тестов +- очевидно, но вот отказываться от большего покрытия не лишено смысла. Очень часто вы можете столкнуться с суровыми реалиями конкурентной среды, где конкуренты дышат вам в спину и только спят и думают, как же отобрать у вас дорогого покупателя. Поэтому очень часто скорость выхода на рынок имеет очень серьезное влияние на разработку. Кстати, поэтому первые версии продуктов и получаются сырые, хоть это и приводит к тому, что каких-то пользователей вы рискуете потерять, как правило, эти риски не настолько существенны по сравнению с тем, что на рынок выйдет ваш конкурент и вам ничего по итогу не останется.

Существуют несколько методов выбора тестового покрытия:

- Партиции эквивалентности (эквивалентное разбиение)

В рамках этого метода производится анализ исследуемой функции на наличие участков, где она ведет себя примерно одинаковым способом.



Все, что нам осталось, это проставить тесты на граничных значениях, и думать, что мы все сделали правильно.

- Таблица альтернативных решений

Строим таблицу комбинаций входных и выходных данных, на этом основании строим тестовые сценарии, покрывающие группы этих сочетаний

- Таблица переходов

Все то же самое, что и в таблице альтернативных решений, только вместо комбинаций входных и выходных данных используем состояния (конечные автоматы), а покрываем тестами переходы между состояниями

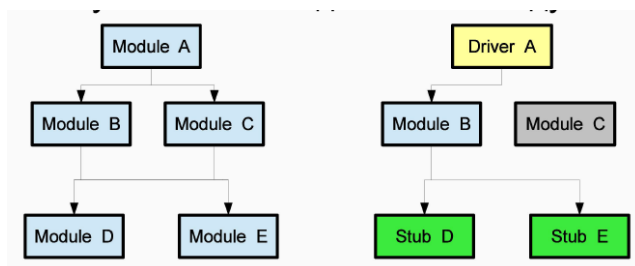
- Сценарии использования

В качестве топлива для тестового покрытия могут быть разработанные ранее сценарии использования. При этом в каждый описанный сценарий вводятся конкретные значения, которые мог бы подать пользователь. Обычно каждому такому сценарию использования соответствует несколько тестовых сценариев.

59. Модульное тестирование. JUnit 4.

Модуль является компонентом системы, который выполняет некую законченную функцию, и который может быть протестирован отдельно от всего вообще. При проектировании системы, мы ее делим сначала на слои, потом на модули. Таким образом, модульное тестирование является наименее масштабным тестированием среди всей системы.

Модуль для тестирования нужно изолировать. Для этого мы вводим *заглушки* (stub), чтобы исключить какое угодно внешнее воздействие в зависимых модулях. Вместо использования реальных вызывающих модулей, мы вместо этого используем *драйверы*.



JUnit это фреймворк для проведения модульного тестирования в Java. Он построен преимущественно на аннотациях, и вот его возможности:

- Обозначить метод как тестовый можно с помощью аннотации `@Test`
- Обозначить метод как исполняемый до/после тестов можно с помощью аннотаций `@Before/@After`. Можно также задавать порядок исполнения тестов, но подразумевается, что все тесты исполняются в параллель и по-хорошему модульные тесты не должны зависеть от порядка их вызова.
- Различные проверки на соответствие определенным условиям при помощи специальных функций (assertations), формирование тестового покрытия
- Предоставляется журнал тестирования.

Последовательность тестов в JUnit не регламентирована. JUnit самостоятельно определяет, каким образом будет запущено тестирование.

60. Интеграционное тестирование. Стратегии интеграции.

Интеграционное тестирование проверяет взаимодействие между программными компонентами и производится после модульного тестирования. В остальном очень похоже на модульное тестирование.

В рамках интеграционного тестирования проверяются:

- Вызовы API, сообщения между компонентами
- Нажатие кнопок на GUI
- Интерфейсы взаимодействия
- Базы данных

Системное интеграционное тестирование проверяет взаимодействие между программными системами или между аппаратным обеспечением и может быть выполнено после системного тестирования.

При составлении плана разработки ПО необходимо учитывать стратегию интеграции, поскольку она прямо коррелирует с календарным планом и сложностью разработки.

Стратегии интеграции:

- Сверху вниз. Сначала мы разрабатываем видимые для конечного пользователя модули, которые выполняют реальные запросы к заглушкам. Это очень распространенная стратегия разработки в бизнес-приложениях, потому что она сразу позволяет предоставить клиенту результаты работы (пользователь часто не видит, что находится под капотом). Из минусов, придется разработать большое число лишнего кода, нужного только для тестирования.
- Снизу вверх. Эта стратегия решает проблему интеграции “сверху вниз”, путем того, что сначала разрабатываются модули самого низкого уровня, далее разрабатываются модули более высоких уровней и интеграция таким образом происходит между реальными модулями, однако это усложняет процесс разработки. Данная стратегия нашла свое применение при производстве аппаратуры, там это наиболее актуально.
- Функциональная стратегия (e2e - end to end) — эта стратегия предусматривает наращивание приложений по функциям. Мы сделали функциональность для одного сценария, протестировали его, делаем дальше.
- Backbone (ядро) — здесь мы формируем минимальную рабочую функциональность, далее на ее основе наращиваем наше приложение.
- Big Bang (стратегия большого взрыва). самая примитивная стратегия. В ней всё собирается одновременно, поэтому в случае возникновения ошибок неясно, где именно они возникают, и что их вызвало.

61. Функциональное тестирование. Selenium.

Функциональное тестирование — подвид интеграционного тестирования, суть которого заключается в том, что мы полностью разрабатываем некоторую функциональность, которая покрывается тестовым сценарием, а затем проводим тестирование. Особенно популярно при разработке бизнес- и веб-приложений. Оно может быть как ручным, так и автоматизированным. Так, например, в случае обработки данных клиента через интерактивную форму добавить еще одно поле, все прошлые тесты сломаются и их

придется дорабатывать. Это заметит ручной тестировщик и сформирует соответствующий запрос на изменение.

Для функционального тестирования разработано множество средств автоматизации, которые могут имитировать ввод пользователя с клавиатуры, перемещать указатель и пр. Одно из таких является расширением для браузера Firefox — Selenium. Оно позволяет записать тестовую последовательность использования интерфейса, затем сохранить это как тестовую программу и использовать ее (даже в других браузерах).

62. Техники статического тестирования. Статический анализ кода.

Проблема динамических тестов проста — нет кода, нечего тестировать (но мы помним про TDD*)

Статические тесты (они же рецензирование) могут быть применены до или во время написания кода, и главное преимущество их в том, что они позволяют находить и предупреждать ошибки еще задолго до включения динамических тестов!

Типичные дефекты, которые легче найти при рецензировании, чем при динамическом тестировании: отклонения от стандартов, дефекты в требованиях или дизайне, недостаточная пригодность к сопровождению и некорректные спецификации интерфейса.

Техники статического тестирования заключаются в следующем:

- “Звонок другу”. Мы зовем нашего опытного товарища, который свежим взглядом просмотрит на наш код и найдет там какой-то недочет или ошибку. Относится к варианту неформальных ревью. К сожалению, у коллеги тоже есть свои задачи, поэтому он будет доступен в течение очень небольшого числа времени
- Технический анализ является формальной (IEEE стандарт) техникой проверки артефактов. Он производится в виде собрания под руководством технического лидера, и происходит повторный просмотр с целью выявления плохих практик при разработке.
- Ревью менеджмента является схожей техникой с техническим анализом, только их проводит отдел менеджмента (это другие люди, но примерно та же заноза в заднице)
- Методика сквозного контроля подразумевает превентивные удары по плохому качеству проектирования и разработки. Они заключаются в проведении презентаций, семинаров, где эксперт “ведет” разработчиков по правильным способам разработки, предупреждая магическое мышление разработчиков.
- Методика инспекции берет собой в основу идею о том, что чем более простую задачу дать человеку, тем лучше он с ней справится. Мы берем несколько человек, которые занимаются примитивными вещами (как правило, двумя) и сажаем их рецензировать артефакты, генерируемые командой.

Также стоит упомянуть про автоматизированные средства статического анализа кода. К ним относятся всякие линтеры (Lint для C и FindBugs для Java.), они есть для практически любого языка программирования. Они строят грамматическое дерево на основании исходного кода и проверяют частосовокупаемые плохие практики использования ЯП

(например, неопределённое поведение (переменная не инициализирована), использование `fopen` без `fclose`, и пр.)

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

После окончания интеграции происходит тестирование системы в целом. Здесь проверяется соответствие заявленным характеристикам. Пререквест: пройденное интеграционное тестирование.

Оно состоит из нескольких частей:

- **Системное тестирование**
- Альфа-Бета тестирование
- Приемочное тестирование

Системное тестирование производится внутри компании разработчика. Формируется тестовое окружение, формируются сценарии, тестирование происходит от простых сценариев к сложным.

Сначала мы тестируем заявленные возможности ПО в соответствии с утвержденными требованиями. Тестируются только корректные варианты работы, используются только основные сценарии.

Далее мы пытаемся тестировать несколько пользователей в системе или повышенную нагрузку на систему от одного пользователя.

Далее мы тестируем усложненные сценарии. К пользовательскому взаимодействию добавляются заведомо некорректные входные данные, взламываются пользовательские файлы, проверяются умышленные попытки привести программу в состояние сбоя.

Далее мы тестируем, совместима ли наша система с Internet Explorer 6. Если нет, увы и ах, все переделываем. Это называется проверка на совместимость с реальным окружением заказчика. Помимо этого, здесь могут проверяться совместимости с различными версиями third-party библиотек.

И наконец, мы проверяем производительность системы. Смотрим на то, как система справляется в повышенных нагрузках, возможно, при ограниченных размерах памяти и/или процессорного времени.

Тестирование производительности включает в себя все виды тестов, которые образовались в англоязычной литературе в сокращение CARAT:

- Capacity — нефункциональные возможности

Они призваны протестировать соответствие нефункциональным требованиям по производительности. Они призваны нагрузить систему до предельно заявленных требований и посмотреть на результат.

- Accuracy — точность

Для некоторого ряда систем погрешность допустимых вычислений очень важна, и она проверяется здесь.

- Response Time — время отклика

Проверяется, насколько система быстро реагирует на поступающие ей запросы. В интерактивных системах это число должно в идеале варьироваться между 1 и 5 секундами.

- Availability — доступность системы

Выражается в виде коэффициента по формуле:

$Availability = (MTBF - MTBR) / MTBF$, где MTBF и MTBR — среднее время между отказами и среднее время между восстановлением соответственно.

Например, для достижения коэффициента 0,99999 (пять девяток), необходимо, чтобы система в среднем давала сбой каждый год (365 дней = 525600 минут) продолжительностью максимум 5 минут $((525600 - 5) / 525600 = 0.99999)$

- Throughput — пропускная способность

Пропускная способность показывает, сколько клиентских запросов может обработать система за единицу времени.

Существуют системы нагрузочного тестирования, которые позволяют автоматизировать этот этап тестирования. К таким относится, например, Gatling или Apache JMeter.

64. Тестирование системы в целом. Альфа- и бета-тестирование.

После окончания интеграции происходит тестирование системы в целом. Здесь проверяется соответствие заявленным характеристикам. Пререквест: пройденное интеграционное тестирование.

Оно состоит из нескольких частей:

- Системное тестирование
- **Альфа-Бета тестирование**
- Приемочное тестирование

Альфа и Бета тестирование производится уже непосредственно пользователями под контролем разработчика.

При альфа тестировании мы приглашаем пользователя к нам в компанию и даем ему пользоваться нашей системой. Все происходит за нашим подготовленным окружением, под нашим контролем. Пользователи могут обнаружить дополнительные косяки, которые не предусмотрели разработчики.

При бета тестировании мы устанавливаем систему уже на конечном компьютере пользователя, но тестирование все еще проводится под нашим контролем.

Как только мы собрали фидбек от пользователей, внесли возможные исправления, наступит приемочное тестирование.

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

- Системный и архитектурный аспект
 - Присутствует влияние архитектуры (монолитная, n-tier)
 - Свое влияние вносит кластеризация, виртуализация
- Низкоуровневый аспект
 - Характеристика аппаратуры (частота, количество ядер, скорости кешей, время доступа к дисковой подсистеме)
- Программный аспект
 - Выбор алгоритма
 - Многопоточность
- Человеческий фактор

Средства измерения могут быть

- Неинтрузивные
- Слабоинтрузивные
- Сильноинтрузивные

Разница в том, что неинтрузивные средства измерения не вносят никакого влияния на результаты работы. В физическом мире, если вставить градусник в воду, вода будет от него нагреваться, как только установится состояние теплового равновесия, тогда мы сможем получить температуру, но это средство измерения внесло серьезные изменения в результат. Поэтому почти все измерительные приборы в физике являются сильноинтрузивными.

В программной инженерии есть пример неинтрузивных средств измерения — счетчики операционной системы. Они все равно будут собираться, хотите вы того или нет, эти издержки были в вашей операционной системе всегда.

66. Ключевые характеристики производительности.

- Время отклика (время от момента запроса к системе до получения первых ответов от нее)
- Время полного обслуживания (время от момента запроса к системе до получения последнего ответа от нее)
- Пропускная способность (максимальное количество одновременно работающих пользователей в системе за единицу времени)
- Утилизация (%util, %busy, какую долю времени ресурс занят полезной работой) и ожидание (%wait %idle) ресурса

- Точка насыщения производительности (момент предельной нагрузки на систему, после которой производительность будет только падать с ростом нагрузки)
- Масштабируемость (показатель возможностей горизонтального (количественного) расширения системы)
- Эффективность, КПД (отношение полезной работы к общей)
- Java: Скорость запуска и memory footprints (стратегии работы с памятью)

67. Нисходящий метод поиска узких мест.

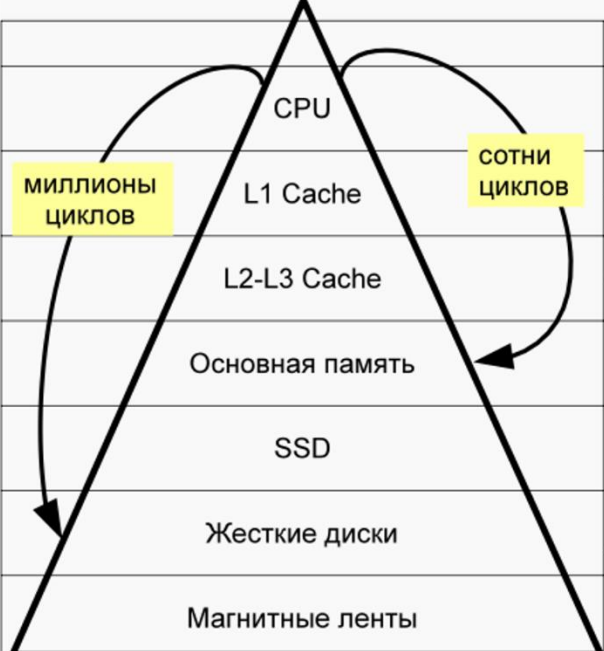
- Исключить ошибки аппаратуры и системного администратора. Проверяются системные журналы, файлы конфигурации системы и прикладного ПО. (Ой, а у нас тут диск перегорел уже 10 лет как) (Ой, а у нас администратор криворукий, настроил 16 Java-машин, каждая резервирует себе по 4GB памяти...)
- Исключить ошибки операционной системы. Средства системного, межузлового мониторинга и мониторинга виртуальных машин помогают наблюдать общую картину работы приложения и определить, на что в ней тратится основное время. Мониторинг CPU, I/O подсистемы, сети, каналов передачи, интерфейсов
- Исключить ошибки приложения. Пофиксить алгоритмические проблемы, проблемы API, исключить взаимные блокировки и пр. В качестве средств наблюдения используются средства мониторинга и профилирования приложений.
- Исключить ошибки микроархитектуры. Пофиксить кеш-промахи, избавиться от пузырьков конвейера и пр. Для улучшения ПО на этом уровне необходимо знать ассемблерный код, архитектуру процессора, принципы компиляции и пр. Обычно в пользовательском ПО до этого уровня не доходят из-за сложности внесения изменений.

Слева стрелку вниз нарисовать

68. Пирамида памяти и ее влияние на производительность.

При разработке программ учет скорости доступа к различным компонентам архитектуры вычислительной среды может существенно повлиять на итоговую производительность.

Пирамида памяти



	Объем	Тд	*	Тип	Управл.
CPU	100-1000 б.	<1нс	1с	Регистр	компилятор
L1 Cache	32-128Кб	1-4нс	4с	Ассоц.	аппаратура
L2-L3 Cache	0.5-32Мб	8-20нс	30с	Ассоц.	аппаратура
Основная память	0.5Гб-4Тб	60-200нс	3-10м	Адресная	malloc, free slab-alloc
SSD	128Гб-1Тб/drive	25-250мкс	5д	Блочн.	open read/write
Жесткие диски	0.5Тб-4Тб/drive	5-20мс	4мес	Блочн.	open read/write
Магнитные ленты	1-6Тб/к	1-240с	200л	Последов.	программно

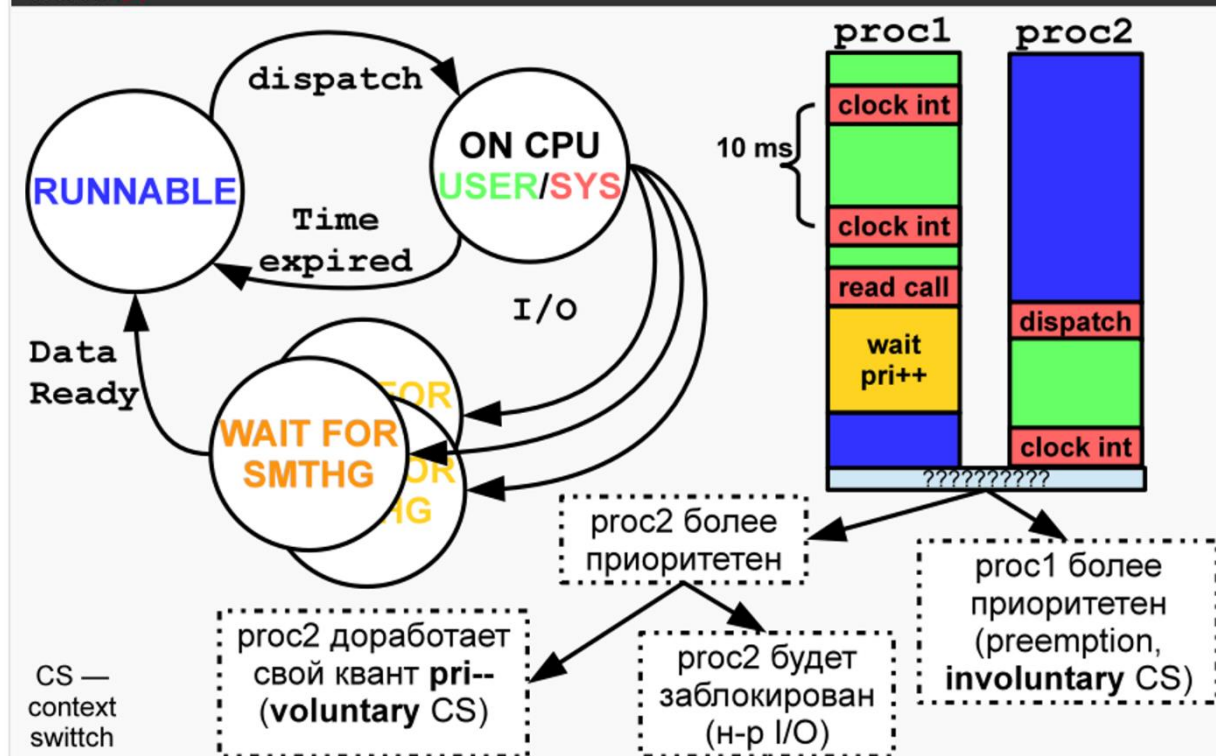
NB: при 2% cache miss производительность падает ниже 60% от идеальной!

Сверху пирамиды находится самая дорогая память с наименьшим объемом. Внизу находится самая дешевая память с максимальным объемом. Чем ниже память в пирамиде, тем дороже в плане затрат времени обходится обращение к ней. Например, обращение к основной памяти может занимать в сотни раз больше времени, чем обращение к регистрам процессора или к кэшу первого уровня. Поэтому минимизация количества обращений к нижним уровням пирамиды во много раз повышает производительность.

Для иллюстрации такого повышения скорости работы программы рассмотрим простой пример. Если отождествить такт процессора со скоростью диалога в процессе межличностного общения (колонка *), то на вопрос, заданный одним собеседником, ответ, данный собеседником, придет через 1 секунду (если он отвечает со скоростью процессора). Если данные, необходимые для ответа, содержится в кэш-памяти первого уровня, то время ответа будет все еще приемлемым, всего 4 секунды. Если в кэше второго уровня — то уже 30 секунд, что для беседы выглядит куда как страшно. Обращение к постоянной памяти, SSD и жестким дискам даст задержку в дни или месяцы, что сильно замедляет быстроедействие процессора.

Учет архитектуры ЭВМ позволяет сделать разрабатываемые программы существенно (на несколько порядков) быстрее.

Диспетчер процессов CPU



Процесс может исполняться на CPU (On CPU), в пользовательском (User) или системном (System) режиме (он же режим ядра), быть готовым к исполнению (Runnable) или ждать чего-то/находиться в блокировке (Wait). Допустим, у нас есть 2 процесса. Пусть в некоторый момент начала наблюдения исполняется процесс 1. На CPU исполняется пользовательская программа. Раз в некоторое время генерируется прерывание от часов и начинает исполняться системный код. Во время работы системного кода наращиваются счетчики производительности, производится подготовка к следующему кванту времени. Далее управление возвращается пользовательской программе.

Как только процессу понадобилось чего-то подождать (например, работу устройства ввода-вывода), сразу после этого, во избежание простаивания ресурсов процессора происходит процесс диспетчеризации. На место работы CPU встает второй процесс, которому отдадут поток управления, и который будет работать до следующего прерывания от часов. Как только от контроллера ВУ поступит прерывание о готовности данных, процесс 1 вернется в состояние Runnable. Как только наступит следующее прерывание от часов, системе придется выбрать, какой из процессов стоит дальше пустить на CPU. В современных ОС это делается при помощи честной системы приоритезации. Если процесс отработал весь квант времени, его приоритет понижается, приоритет остальных процессов, наоборот, повышается за каждый квант времени, за который процесс простаивает в состоянии Runnable.

В случае на картинке может быть 3 сценария:

- Процесс 1 оказался более приоритетным. Произойдет involuntary context switch (вытесняющая многозадачность), мы отбираем у процесса его время и отдаем ее другому процессу.

- Процесс 2 оказался более приоритетным и он доработал до блокировки. В таком случае процесс передачи управления при помощи `dispatch` от процесса 1 к процессу 2, описанного ранее, повторится с точностью до обратного.
- Процесс 2 оказался более приоритетным и доработал полностью. В таком случае произойдет `voluntary context switch`.

70. Мониторинг производительности: виртуальная память.

Виртуальная память это механизм управления памятью компьютера, при котором программа работает с виртуальным адресным пространством, которое привязывается к физическим адресам, давая исполняемой программе иллюзию того, что она единственная исполняется на системе, но более того, виртуальная память позволяет разделять общие участки кода (например, `libc` или ядро операционной системы) между разными процессами, позволяя экономить физическую память. К тому же, механизм виртуальной памяти позволяет перемещать неиспользуемые страницы памяти на более медленный накопитель, экономя место в оперативной памяти (своппинг).

Современная операционная система старается держать некоторое количество страниц памяти всегда свободными (для каких-то сверх экстренных нужд). Когда количество свободных страниц уменьшилось до определенного порога, запускается процесс, который начинает сканирование виртуальной памяти на предмет неиспользуемых страниц для записи и для чтения (с помощью проверки обращений к странице). Неиспользуемые страницы сбрасываются в своп или даже удаляются из памяти вовсе.

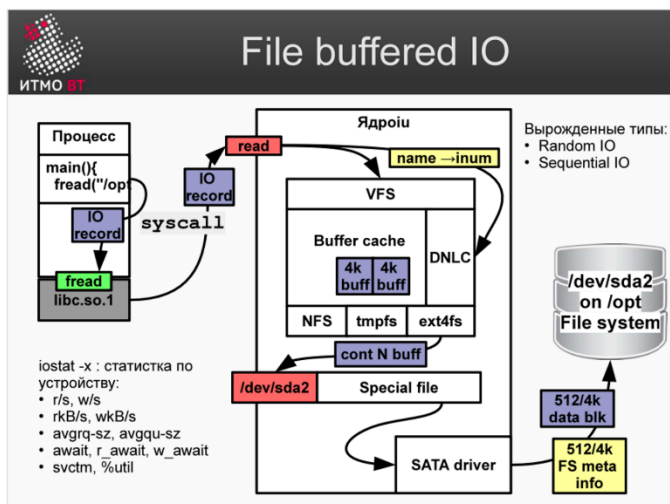
Если свободной памяти все равно становится меньше, то система увеличивает т. н. `scan rate`, т. е. количество проверок виртуальной памяти становится все больше и больше, система пытается сбросить как можно больше страниц памяти.

Если это все равно не помогает, операционная система начинает подвешивать приходящие ей запросы на аллокацию памяти, операционная система постоянно работает со страницами в памяти и увеличивается количество `page fault`'ов.

Если это тоже не помогает, и количество памяти уменьшается, система начнет отвергать запросы на аллокацию памяти. Это означает, что `malloc` в C будет возвращать `NULL`, а в JVM будет сгенерирована ошибка `OutOfMemoryError`.

В Linux мониторинг виртуальной памяти производится при помощи команды `sar -B`. Эта команда даст вам знать о количестве загружаемых/выгружаемых/освобождаемых страниц в файл подкачки в секунду, количество промахов в страницах, `scan rate` и другую полезную информацию о виртуальной памяти.

71. Мониторинг производительности: буферизированный файловый ввод-вывод.



При чтении данных вначале указывается количество байт, которые нужно прочитать (10 record size). Интенсивность чтения данных определяется требованиями программы.

Пусть в нашей программе появилось обращение к файлу, чтобы что-то оттуда прочитать. Рано или поздно это дойдет до системного вызова к подсистеме VFS в ядре (VFS - Virtual File System). Она обратится к Directory Name Lookup Cache (DNLC) для преобразования имени файла в его номер inode в необходимой файловой системе. Далее операция чтения будет работать с кешем. Этот кеш хранит в себе подгруженные с диска данные, любые изменения раз в какое-то время будут сгружаться обратно на диск. Если был обнаружен промах в кеше, загружается специальный файл драйвера работы с конкретной файловой системой и далее уже ведется работа по чтению данных с конкретной аппаратуры.

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

Получить эту статистику в Linux можно с помощью команды `iostat -x`

72. Мониторинг производительности: Windows и Linux.

Мониторинг производительности в Windows представляет собой встроенное средство просмотра всей основной информации об использовании компьютера — Windows Task Manager, где выводится статистика использования процессора, памяти, дисковой подсистемы и др. Для более детального исследования поставляются также Resource Monitor & Performance Monitor, Reliability Monitor и т.д.

Более подробная статистика с продвинутыми счетчиками есть в коммерческом и свободно распространяемом ПО, наиболее продвинутым средством является разработка Microsoft SysInternals.

В ОС семейства Linux ситуация на первый взгляд выглядит сложнее, чем с Windows. В Linux существует большое число средств мониторинга, наблюдающих за определёнными подсистемами ОС и учитывающих особенности архитектуры этих подсистем. В общем случае они являются неинтрузивными.

Для того, чтобы получить информацию из счетчиков Linux, вам для этого помогут команды в терминале, например, `vmstat`, `top`, `perf`, `sar` и другие:

- `vmstat` соберет основную информацию из всех основных подсистем Linux (CPU, Swap, IO, Memory), раз в определенное время в зависимости от аргументов командной строки
- `top` позволяет следить за процессами в системе, например, за их приоритетом, занимаемой CPU и памятью (виртуальная, физическая, разделяемая с другими процессами) и другую информацию. В статусной строке можно увидеть большое число общесистемных характеристик
- Утилита `sar` является одной из первых утилит в ОС Unix (по состоянию на 2024, в macOS Ventura этой команды не найдено даже в POSIX-совместимом shell) по мониторингу производительности, позволяет мониторить основную системную информацию ядра. Некоторые метрики собираются устаревшим способом, поэтому команду стоит использовать с осторожностью
- Отдельно стоит выделить утилиту `perf`, которая позволяет собирать огромное число метрик не только ядра, но и приложения, запущенного под этой утилитой. `Perf` умеет работать с большим числом счетчиков производительности процессора для сбора и поимки таких событий, как, например, промахи кеша.
- `Star` позволяет установить точки в ядре, чтобы агрегировать информацию о работе подсистем ядра.

73. Системный анализ Linux "за 60 секунд".

Практический подход “Системный анализ Linux за 60 секунд” представляет собой последовательность команд, исполнение которых быстро введет человека в курс дела о том, что происходит в системе:

- `uptime` покажет `load average` — количество готовых к выполнению процессов в очереди на диспетчеризацию за 1, 5 и 15 минут. Если процессов много, система не справится с нагрузкой
- `sudo dmsg | tail` — часто в системе есть какие-то явные сбои, взглянув на которые можно понять, куда двигаться дальше. Эта команда позволяет получить логи системных сбоев.
- `vmstat 1` — следом мы получаем представление о том, есть ли у нас свободная память, происходит ли процесс пэйджинга/своппинга, как утилизируется центральный процессор
- `mpstat -P ALL 1` — смотрим на распределение процессов по центральному процессору, так как возможная ситуация, когда занят только один процессор (или ядро), остальные простаивают.
- `pidstat 1` — проверяем наиболее “горячие” процессы, смотрим статистику
- `iostat -x 1` — проверяем работу подсистемы ввода-вывода
- `free -m` — проверка исчерпания кешей или буферов
- `sar -n DEV 1` — смотрим сетевую статистику по интерфейсам
- `sar -n TCP, ETCP 1` — смотрим сетевую статистику по соединениям
- `top` — в завершение, эта команда является швейцарским ножом (шутка, швейцарским ножом является утилита `perf`) для системного администратора, многие названные выше характеристики можно посмотреть в едином месте с помощью как раз данной утилиты

74. Создание тестовой нагрузки и нагрузчики.

- Средства мониторинга — сильноинтрузивные

В корпоративных системах бывает строго настрого запрещено прикасаться к боевой системе:

- Они могут создать дефекты в работе системы → BSOD/Kernel Panic

Решение: создать такую же систему как основную и нагрузить ее “так же, как и основную”:

- Для этого есть средства создания синтетической нагрузки (Apache JMeter, Gathling)
- Средства, которые записывают реальную нагрузку (присутствуют на боевой системе), чтобы помочь ее воспроизвести

Все-таки стоит отметить, что синтетическая нагрузка **всегда** будет отличаться от реальной, насколько близко (с помощью большого числа гибко настраиваемых параметров) мы бы не пытались это настроить.

75. Профилирование приложений. Основные подходы.

После того, как мы выяснили, что вся наша аппаратура в порядке, а операционная система и все ее подсистемы настроены хорошо, но производительность нас по-прежнему не устраивает, в дело вступает профилирование приложения. Если у нас есть исходные коды, мы можем в них провалиться в поиске алгоритмических дефектов.

С помощью профилировщика можно узнать:

- Время исполнения метода
- Объем созданных объектов в памяти
- Состояние многопоточности, захваченные блокировки и пр.

Существуют два основных подхода к профилировке:

- Диагностические точки можно внедрять в сами функции из указанного набора (в начале и в конце функции). Это интрузивный подход, мы меняем исходный код и заставляем тратить время на дополнительное исполнение. К тому же мы тратим время на поиск этих самых точек для внедрения профилирования
- Используем прерывания программы, и во время прерывания собираем интересующую нас информацию, например, анализируем стек вызовов, смотрим на состояние кучи.

Интервал прерываний выбирается так, чтобы не слишком сильно снижать производительность, и в то же время включить в выбранные данные даже методы с

малой продолжительностью работы. Так работают коммерческие профилировщики, например, Performance Analyzer из Oracle Solaris Studio.

76. Компромиссы (trade-offs) в производительности.

Любая борьба за производительность сопряжена с т. н. trade-off'ами (компромиссами):

- Различные подсистемы ЭВМ взаимосвязаны: чем быстрее мы имеем доступ к данным, тем больше памяти они занимают ((linear seach) 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)
- Выбор алгоритма и архитектуры может существенно ускорить приложение, либо сделать его максимально компактным

Примеры:

- Добавление кеша не обязательно делает доступ к памяти быстрее (нам требуется время на перегон памяти между промежуточным звеном между процессором и памятью)
- Бесконечное увеличение кеша бесполезно — настанет момент времени, когда добавление кеша не будет добавлять прирост к производительности, при этом будет расти стоимость такого решения.
- Выбор алгоритма сортировки на маленьких и больших объемах данных может существенно отличаться. Иногда нам будет рациональнее взять асимптотически более тяжелый алгоритм в угоду его простоты или компактности.

77. Рецепты повышения производительности при высоком %SYS.

- Высокая нагрузка на ввод вывод (сеть, диск,...)
 - Рассмотреть возможность меньше обращаться к подсистеме ввода-вывода. Сжимать данные, оптимизировать систему.
 - Купить буферы, более быстрые устройства ввода-вывода
 - Синхронизировать размер блока передачи между ОС и устройством ввода-вывода
- Высокая степень работы планировщика (постоянные контекст свичи, высокий uptime)
 - Уменьшить количество потоков, наладить работу ОС
 - Забиндить потоки к своим ядрам
- Paging/Swapping
 - Выдать больше памяти системе, ограничить память процессам
 - Запретить swapping
- Ядро пишут тоже люди
 - Попытаться найти и исключить лишние системные процессы
 - Попытаться разобраться в конфигурации ядра
 - Сообщить о какой-то проблеме при ее обнаружении в багтрекер

78. Рецепты повышения производительности при высоком %IO wait.

- Проблемы приложений
 - Рассмотреть возможность меньше обращаться к подсистеме ввода-вывода. Сжимать данные, оптимизировать систему.
 - Проверить, согласован ли блок приложения с ОС и диском (например, если при размере stripe-size в 16 Кб, читать данные блоками по 8 Кб, то происходит, фактически, двойное прочтение, и преимущества RAID-массива сходят на нет)
- В системе выделено мало кеша
 - Расширить кеш
 - Настроить его грамотное использование, исключить лишние промахи
- Проблема аппаратуры
 - Купить более быстрые устройства ввода-вывода (новая дисковая подсистема, SSD или отдельные карты flash памяти, непосредственно устанавливаемые в шину (например PCIe) вычислительной системы)

79. Рецепты повышения производительности при высоком %Idle.

- Программа убивается в один поток на одном ядре (или около того)
 - Распараллелить все, что можно (особенно, если система многопроцессорная)
 - Добавить потоки в пулы приложений (если существуют пулы потоков-worker'ов, то в них можно добавить дополнительные потоки для равномерной загрузки всех ядер ЦПУ)
 - Анализ блокировок, исключить потенциальные взаимные блокировки
 - По возможности использовать lock-free алгоритмы
- Проблемы внутри ОС, ядро тоже пишут люди
 - ОС имеет слишком много блокировок — мониторинг средствами ОС
 - Настройка параметров в ядре
 - Поиск дефектов ядра в багтрекере, сообщить о проблеме, если таковая была обнаружена

80. Рецепты повышения производительности при высоком %User.

Проблемы приложений: не всегда это означает полезную работу программы

- Использовать алгоритмы с меньшей алгоритмической сложностью
- Не создавать по новой лишние объекты в памяти, аллоцирование памяти это невероятно дорогая операция, повторное использование объектов
- Избавиться от активных опросов и ожиданий в цикле (busy-waiting)
- Искать промахи в кешах (заменять структуры данных) или в трансляторе виртуальной памяти (использовать больший размер страничек памяти)
- Учитывать размер и организацию кэшей:

- • Размер кэшей ограничен.
- • Паддинг и разрывание объектов.
- • Использовать более слабые примитивы синхронизации.
- Для исключения остывания кэшей — биндить потоки и процессы к конкретным ядрам/процессорам
- Оптимизации на уровне компилятора или языков более низкого уровня
- Повысить аппаратные характеристики, увеличить тактовую частоту, количество ядер, использовать GPU (прим. ред.: NVIDIA CUDA) или иную вычислительную технику