

## Вопросы к экзамену по дисциплине "МисПИ" 2021/2022

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

Жизненный цикл ПО – время существования программы от момента начального замысла и до окончательного вывода из эксплуатации. Все этапы описаны в стандартах ISO.

Все основные задачи при разработке ПО можно свести в определенные группы или этапы, а затем обсуждать характеристики задач отдельно, внутри каждого этапа.

Основные этапы: разработка требований, анализ, проектирование, разработка, тестирование, внедрение, эксплуатация, вывод из эксплуатации

Требования формулируются не только самими заказчиками, они согласуются с разработчиками через аналитиков, превращающих требование заказчика в способ решения поставленной задачи.

После анализа проектируется архитектура и шаблоны реализации ПО, а затем задание передается разработчикам. Одновременно с этим формируются подходы и производится тестирование, после чего продукт внедряется и эксплуатируется.

При эксплуатации осуществляется поддержка пользователей, исправление дефектов и обновление ПО. При выводе из эксплуатации производятся специальные процедуры связанные с выводом ПО из использования (или переходом на новую систему), например, обеспечение сохранности накопленных файлов.

Согласно ISO стандартам ЖЦ содержит 43 процесса:

- 2 согласования
- 5 орг. Обеспечения
- 7 проектов
- 11 тех. процессов
- 7 реализации программных средств (ПС)
- 8 поддержки ПС
- 3 повторного использования ПС

Каждый процесс описан по схеме Входные данные и ресурсы → совокупность действий → выходные данные и ресурсы. Совокупность действий по обработке данных и ресурсов не указывает конкретные действия, входные и выходные данные могут включать в себя требования по времени и финансам и т. п.

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

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

Простейшая модель – последовательная (водопадная/каскадная). Все стадии разработки выполняются последовательно и один раз, результаты разработки не меняются. Все требования определены, этап разработки один. Легко прогнозировать примерные сроки и предсказать стоимость для заказчика. Требования заказчика могут измениться – обратная связь будет слишком поздно – минус модели.

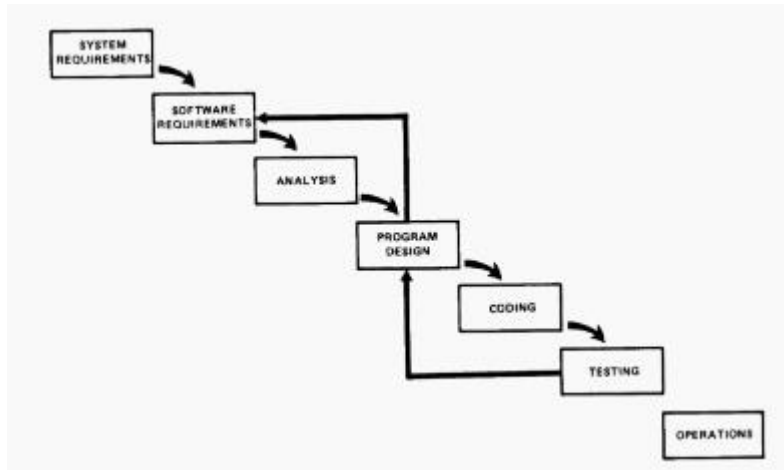
Инкрементная модель – ЖЦ разбит на несколько частей, основа разбиения – номенклатура функциональных требований. Все части приблизительно одинаковы с архитектурной точки зрения => устаревание архитектуры со временем.

Эволюционная модель – разрабатывается прототип, который с течением времени архитектурно и функционально развивается.

Чаще всего применяется инкрементно-эволюционная модель, хотя существует популярный SCRUM – эволюционная модель.

Также, модель формальных преобразований – модели ПО последовательно преобразуются друг в друга, а затем в программный код по определенным формальным принципам. Не распространена.

### 3. Водопадная (каскадная) модель.

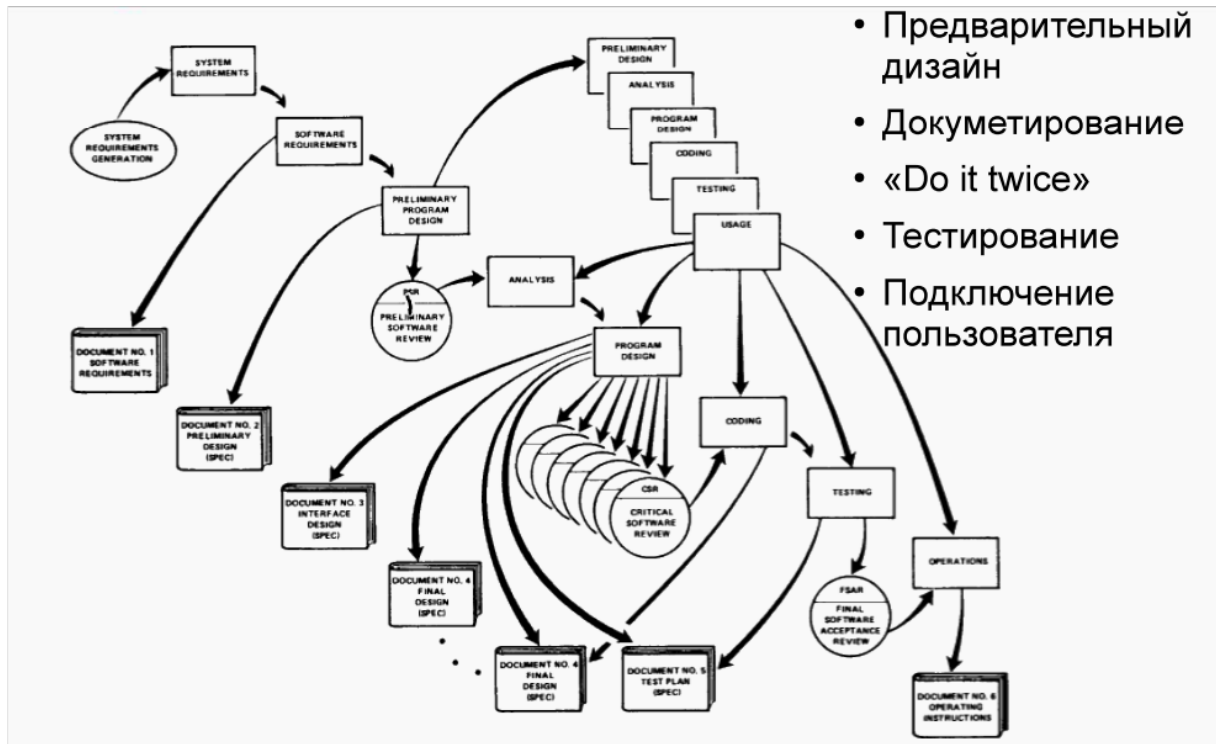


Стандартная последовательность шагов:

1. Определить системные требования
2. Требования к ПО
3. Анализ требований
4. Проектирование (возможен возврат к требованиям ПО)
5. Разработка кода
6. Тестирование (возможен возврат к проектированию)
7. Ввод в эксплуатацию

Возможен возврат к ранним стадиям, однако крайне затратен (увеличение сроков и стоимости в 2 раза).

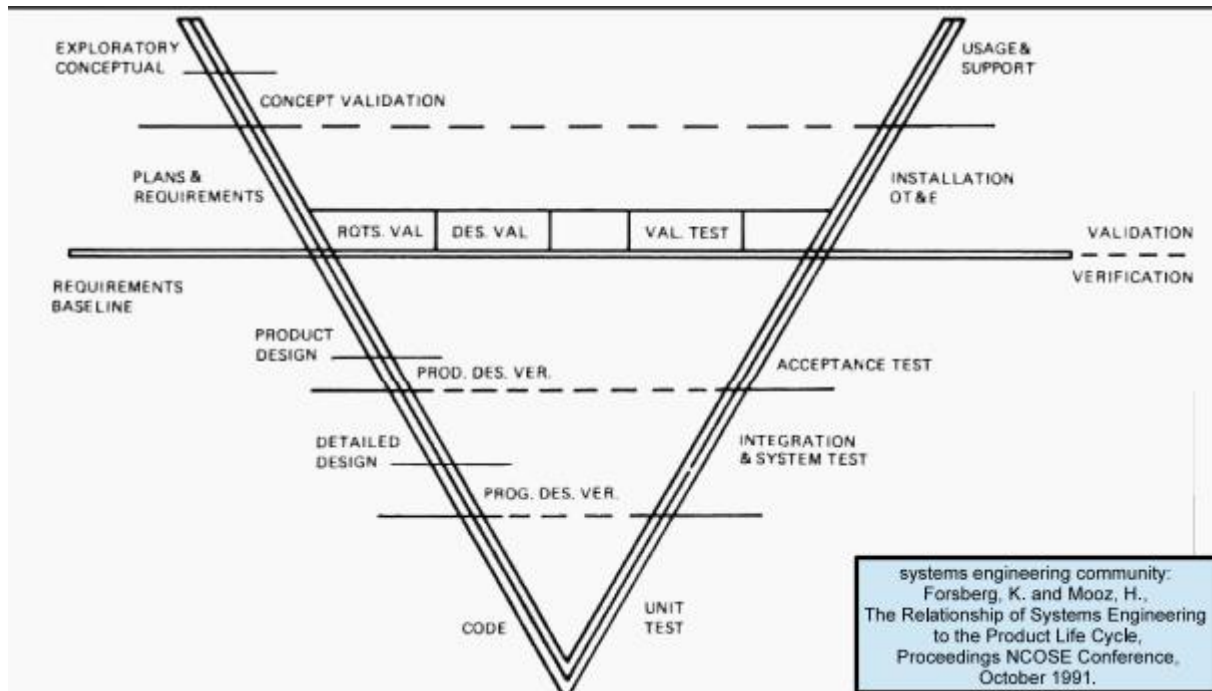
#### 4. Методология Ройса.



Основана на каскадной модели, расширяет ее пятью шагами, делающие модель менее рискованной.

1. Предварительный дизайн – стадия между программными требованиями и анализом. Дизайнер может убедиться в реализуемости требуемых характеристик ПО. Пред. Дизайн выполняется только дизайнерами, без участия программистов и аналитиков. Предлагается спроектировать, определить и создать модели обработки данных и разработать документ – обзор будущей системы.
2. Документирование дизайна – перечислены важнейшие документы для разработки ПО. Требования к системе, спецификация предварительного дизайна, спецификация дизайна интерфейсов, финальные спецификации дизайна системы, план тестирования и инструкция по использованию.
3. Do it twice – проведение тестовой разработки параллельно основной, использование ее как пилота с сокращенным временем разработки, для подтверждения или опровержения основных характеристик ПО.
4. Планирование, контроль и мониторинг тестирования – исключение дизайнера системы из процесса тестирования, проведение визуальной инспекции кода другим лицом, которое отметит визуально заметные дефекты. Протестировать каждый логический путь внутри программы (труднореализуемо). Провести проверку программы в тестовом окружении после исправления большинства простых ошибок.
5. Подключение пользователя на ранних этапах перед финальной поставкой продукта – выделены три точки, где необходим фидбэк пользователя – предварительный, критический и финальный просмотры.

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

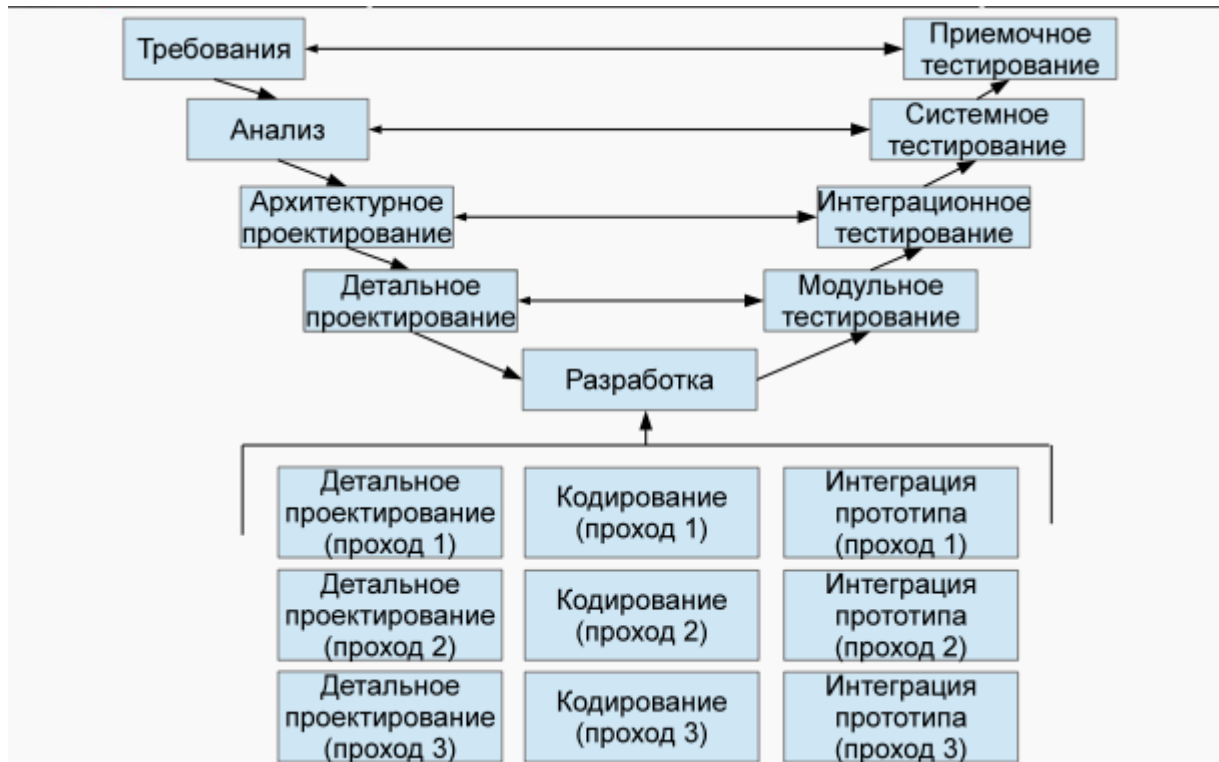


В основе та же последовательность шагов, что и в каскадной. Каждому уровню разработки сопоставлен свой уровень тестирования. Модульное, интеграционное и системное тестирования проводятся последовательно на основе критериев верификации соответствующих уровней разработки. Последнее – приемочное тестирование – проверка соответствия продукта всем основным функциональным требованиям.

Для проведения тестирования необходимо определить корректное поведение программы. Эталонное поведение должно быть задано вне кода разработанной системы.

Статическое тестирование выполняется на ранней стадии проекта для выявления грубых ошибок проектирования.

## 6. Многопроходная модель (Incremental model).



Продукт разбивается по отдельным функциональным и техническим требованиям, далее проектируется, реализовывается и интегрируется воедино в несколько проходов разработки в виде отдельных сборок – инкрементов функционала.

Существенно снижается стоимость изменений требований заказчика. Заказчик может использовать в работе частично разработанную систему. Гибкие современные методологии опираются на инкрементальную модель.

Недостатки – тенденция к устареванию и деградации архитектуры => необходимость рефакторинга (переработки), что сложно и дорого.

Из-за большой скорости изменений трудно управлять проектом и поддерживать документы по программным сборкам.

Заключение контрактов осложнено, т. к. традиционные подходы предполагают фиксированные суммы, сложно учесть потенциальные изменения и включить почасовую оплату.

## 7. Модель прототипирования (80-е).



Последовательность прототипов, каждый из которых уточняет архитектуру в рамках функциональных требований. В начале разработки программа представляет собой каркас для дальнейших разработок. ПО строится эволюционно.

Сначала планируется вся итерация, после чего проводится быстрый анализ требований и подходов к реализации, создается база данных и интерфейс пользователя, разрабатывается функционал и проверяется совместно с пользователями системы => проверка удовлетворенности пользователей прототипом, его архитектурой, основными параметрами. Если пользователь доволен – переход к разработке окончательной версии ПО.

Иначе – новый прототип. Новые прототипы создаются пока пользователь не протестирует все планируемые характеристики продукта.

В современных методах применяются средства макетирования для создания подобных прототипов, мокапов – простых визуальных моделей пользовательского интерфейса, и более сложных UX моделей – моделей интерфейса, которые можно опробовать.

## 8. RAD методология.





Появление средств разработки, способные автоматизировать повседневную деятельность программиста + потребность компаний в автоматизации крупного бизнеса => появление RAD.

Были созданы первые прообразы современных средств автоматизации, IDE, и предложена методология, учитывающая их использование и вовлечение бизнес-пользователей в разработку. Бизнес-пользователи могли сами проектировать бизнес функции.

Методология состоит в том, что сначала производится проектирование, а затем с помощью средств автоматизации ПО собирается как из кубиков.

Пользователь принимает участие в процессе разработки при помощи средств автоматизации – создает простейшие функции и интерфейсы. Даже если реализация не совсем удобная, она решает проблему реализации бизнес функции, которую нужно было разработать уже вчера.

Современный пример – Oracle E-Business Suite – пользователь может, зная бизнес логику, создавать формы данных и выполнять запросы к системе с помощью пользовательского интерфейса.

В RAD могут использоваться CASE системы – Computer Aided Software Engineering – системы, покрывающие полностью цикл разработки от определения требований и до отслеживания ошибок в ПО. Средства производят учет и анализ требований, построение моделей и генерацию первоначального кода на их основе. В них интегрированы модули для взаимодействия с системами контроля версий, проведения тестирования, управления дефектами, потоком задач и т. д.

Современный пример – продукты Rational.

## 9. Спиральная модель.



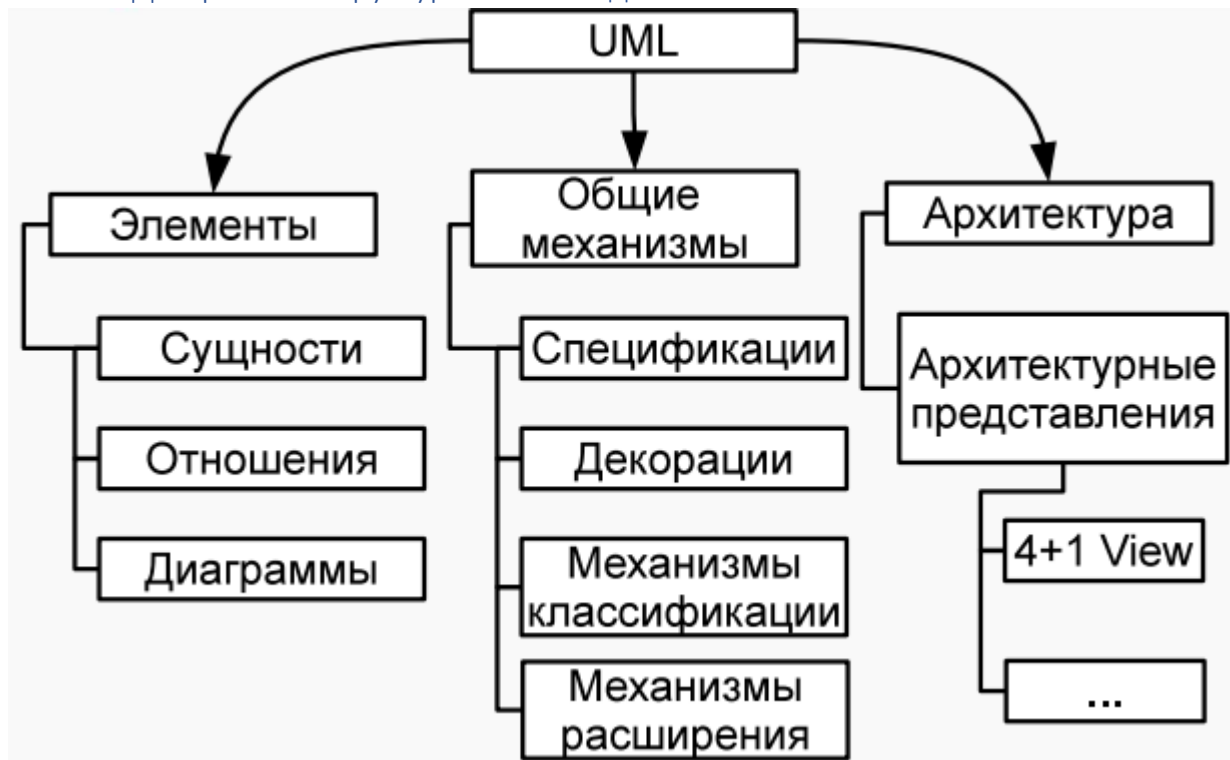
Каждый виток – фаза разработки ПО для построения очередной версии ПО или прототипа. На каждой итерации сначала ставятся цели, выявляются альтернативные решения и ограничения,

которые необходимо учитывать в данной фазе. Затем производится анализ рисков для минимизации числа действий, необходимых для разработки. Производится сверка с картой рисков (насколько вероятно в следующей итерации наступление риска). Анализируются все типы рисков.

Далее производится разработка и валидация полученной версии ПО. На разных фазах разрабатываются основные концепции ПО, функционал, дизайн продукта, компонентов и их воплощение в коде. В конце последней итерации – тестирование разработанного ПО как в V модели.

Преимущество модели – принятие изменений как неотъемлемую часть разработки, определение того, что изменения могут быть приняты, отклонены или проигнорированы на основе возможных рисков для процесса разработки.

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







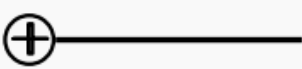


UML – графический язык моделирования общего назначения, для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем.

Основное назначение UML – графическое представление различных аспектов разработки программного обеспечения.

Основные элементы UML – сущности, отношения их связывающие и диаграммы – представления моделей (сущностей).

Отношения в UML:



- Зависимость 
- Ассоциация 
  - Агрегация 
  - Композиция 
  - Включение 
- Обобщение 
- Реализация 

Зависимость показывает, что изменение целевого элемента может повлиять на исходный.

Ассоциация показывает, что один элемент связан с другим, не уточняя смысловое наполнение.

Агрегации – вид ассоциации, где целевой элемент является частью исходного элемента и может существовать как отдельный элемент.

Композиция – строгая форма агрегации, часть не может существовать без целого и доступна только через точки взаимодействия.

Включение – исходный элемент содержит целевой, обычно используется в диаграмме пакетов.

Обобщение – исходный элемент является специализацией более обобщенного целевого элемента и может его замещать.

Реализация – исходный элемент реализует интерфейс целевого элемента.

## • Структурные

– Объект

Мой любимый пирожок

– Класс

Пирожок

Еда {abstract}

– Интерфейс

Подогреватель

<<interface>>  
Подогреватель

– Кооперация (взаимодействие)

Кухня

– Действующее лицо

Бабушка

Бабушка

– Компонент

http HA-proxy

<<component>>  
Postgresql

– Артефакт

<<artifact>>  
Концепция

<<library>>  
makesgood.jar

– Узел

Сервер

Сущности есть структурные и поведенческие.

- Поведенческие

- Прецедент использования
- Состояние
- Деятельность
- Действие

- Дополнительные

- Пакет
- Комментарий



Существуют структурные и поведенческие диаграммы.

Структурные – например диаграмма классов, компонентов, развертывания, пакетов и т. п.

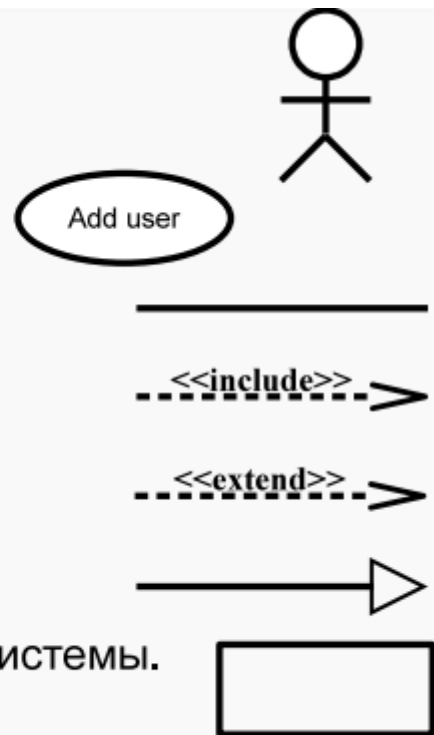
Поведенческие – диаграмма деятельности, состояний, Use-Case и др.

Наиболее часто используются:

- Диаграмма вариантов использования – описывает высокоуровневые требования к системе с помощью вариантов ее использования
- Диаграмма классов – при анализе - в виде доменной модели для описания предметной области без деталей реализации; также для иллюстрации архитектурных механизмов с деталями реализации.
- Диаграмма деятельности, последовательности и состояний.
- Диаграмма размещения – описывает архитектуру системы.

## 11. UML: Use-case модель.

- Actor — действующее лицо.
- Use Case — прецедент использования.
- Association — ассоциация, использование.
- Include — включение.
- Extend — точка расширения функционала.
- Generalization — обобщение.
- System boundary — границы системы.



В RUP Use-Case модель определена как инструмент графического отображения требований для упрощения взаимодействия заинтересованных лиц. Главный элемент – актер – пользователь системы. Под ним указывается его роль.

Внутри прецедентов пишется глагол или фраза, указывающая на действие системы.

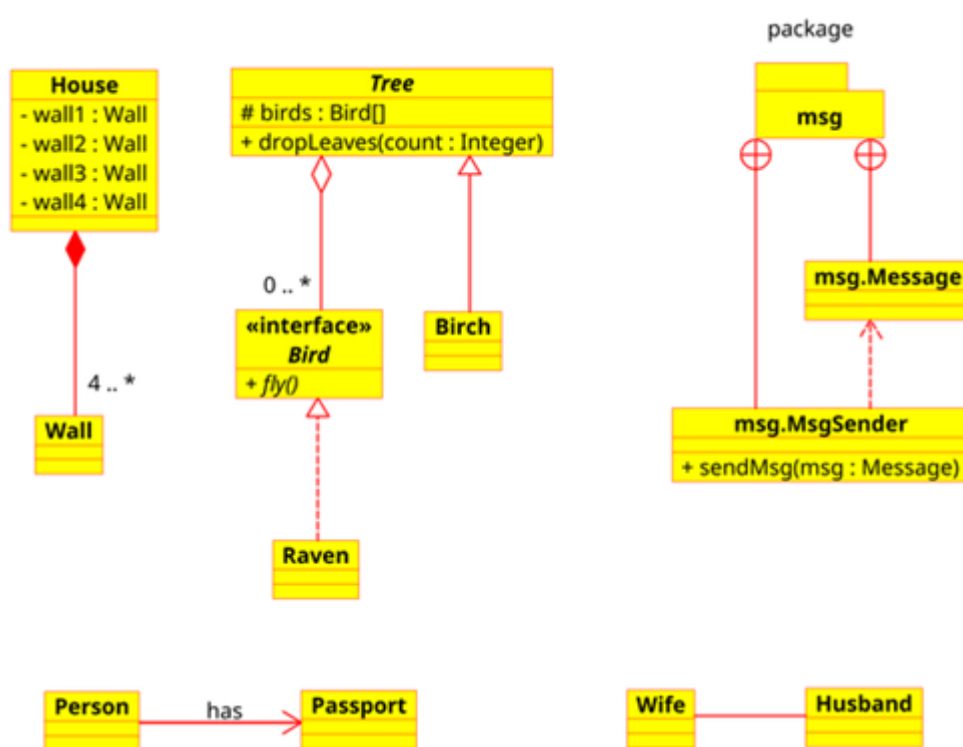
Ассоциация – связь, указывающая на выполнение кейса актером.

Используются стереотипы включения и расширения (точка расширения от базового функционала).

Обобщение указывает на то, потомком какого более высокого класса является объект.

Элемент границы системы важен, когда система разбита на подсистемы.

## 12. UML: Диаграмма классов.



Диаграммы классов для описания предметной области также называют доменной моделью. Преимущества подобной модели в наглядном изображении предметной области, возможность простого расширения и уточнения, например добавить новые типы данных, области видимостей и т. п.

Целью создания диаграммы классов является графическое представление статической структуры декларативных элементов системы (классов, типов и т. п.) Она содержит в себе также некоторые элементы поведения (например — операции), однако их динамика должна быть отражена на диаграммах других видов (диаграммах коммуникации, диаграммах состояний).

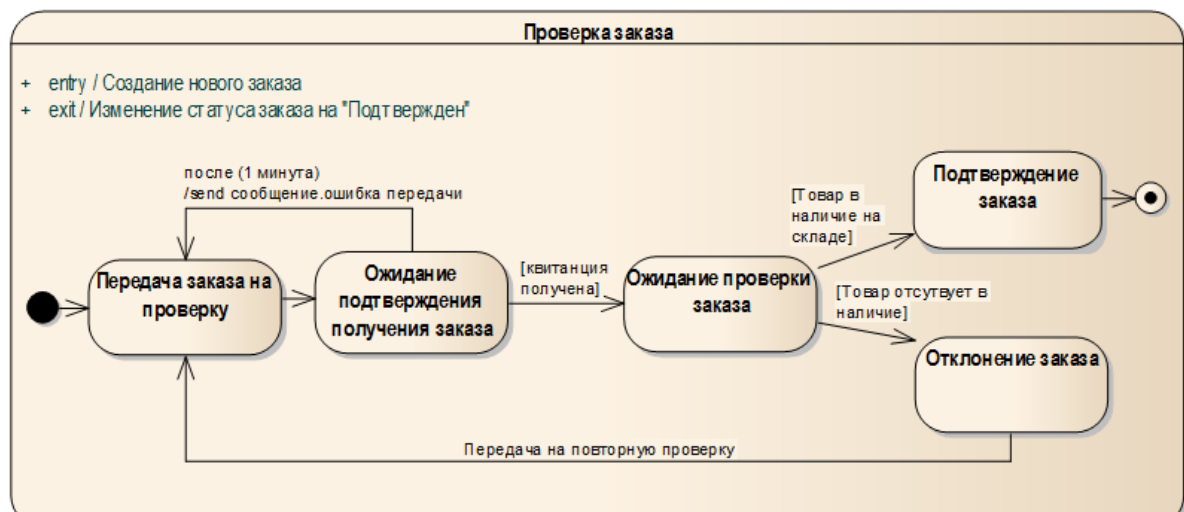
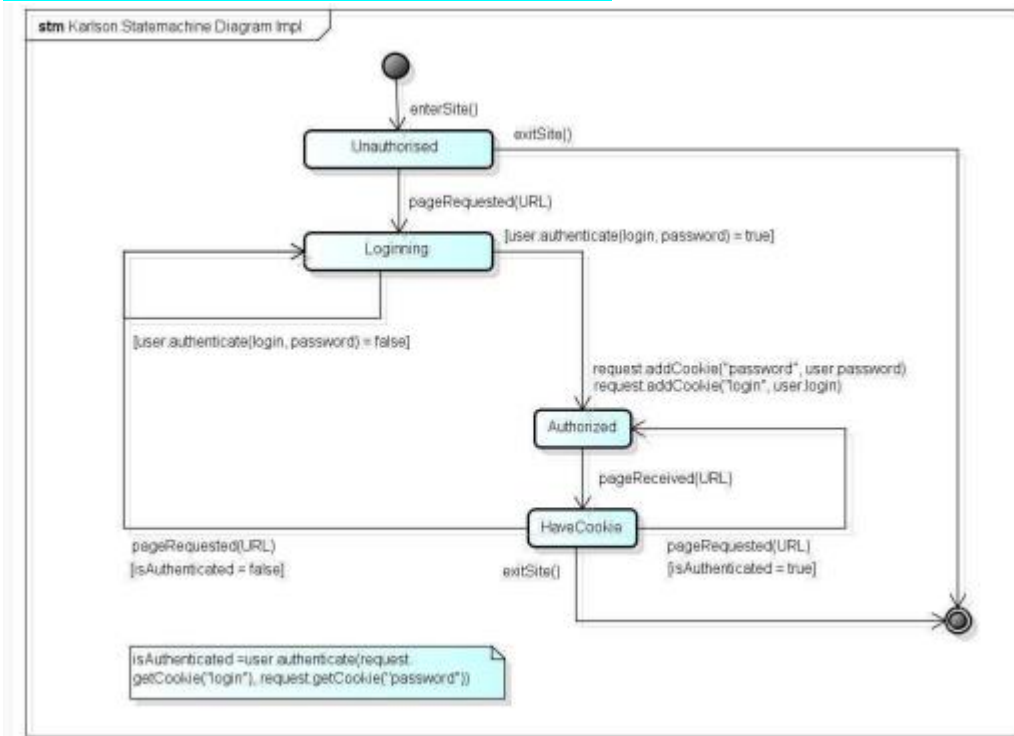
Для задания видимости членов класса (то есть — любым атрибутам или методам), эти обозначения должны быть размещены перед именем участника:

- + Публичный (Public)
- - Приватный (Private)
- # Защищенный (Protected)
- / Производный (Derived) (может быть совмещен с другими)
- ~ Пакет (Package)

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце. Различают следующие типичные случаи:

нотация	объяснение	пример
0..1	Ноль или один экземпляр	Кошка имеет хозяина.
1	Обязательно один экземпляр	у кошки одна мать
0..* или *	Ноль или более экземпляров	у кошки могут быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

### 13. UML: Диаграмма последовательности



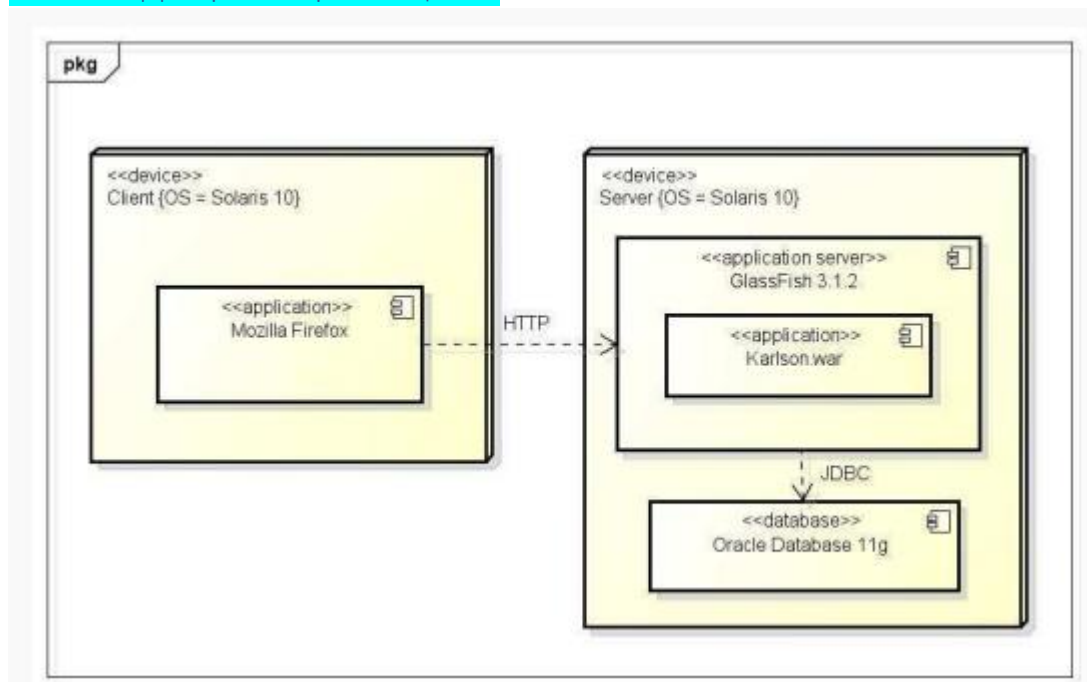
Также называют диаграммой состояний или диаграммой конечных автоматов. Диаграмма может содержать элементы деталей реализации, но не должна быть ими перегружена.

Используются следующие условные обозначения:

- Круг, обозначающий начальное состояние.

- Окружность с маленьким кругом внутри, обозначающая конечное состояние (если есть).
- Скругленный прямоугольник, обозначающий состояние. Верхушка прямоугольника содержит название состояния. В середине может быть горизонтальная линия, под которой записываются активности, происходящие в данном состоянии.
- Стрелка, обозначающая переход. Название события (если есть), вызывающего переход, отмечается рядом со стрелкой. Охраняющее выражение может быть добавлено перед «/» и заключено в квадратные скобки (название\_события[охраняющее\_выражение]), что значит, что это выражение должно быть истинным, чтобы переход имел место. Если при переходе производится какое-то действие, то оно добавляется после «/» (название\_события[охраняющее\_выражение]/действие).
- Толстая горизонтальная линия с либо множеством входящих линий и одной выходящей, либо одной входящей линией и множеством выходящих. Это означает объединение и разветвление соответственно.

#### 14. UML: Диаграмма размещения



Показывает физическую архитектуру размещения частей приложения на серверах, описывает архитектуру системы с деталями установки компонентов, их версий и структуры взаимной вложенности.

Узлы представляются как прямоугольные параллелепипеды с артефактами, расположенными в них, изображенными в виде прямоугольников. Узлы могут иметь подузлы, которые представляются как вложенные прямоугольные параллелепипеды. Один узел диаграммы развертывания может концептуально представлять множество физических узлов, таких как кластер серверов баз данных.

Существует два типа узлов:

- Узел устройства

- Узел среды выполнения

Узлы устройств — это физические вычислительные ресурсы со своей памятью и сервисами для выполнения программного обеспечения, такие как обычные ПК, мобильные телефоны.

Узел среды выполнения — это программный вычислительный ресурс, который работает внутри внешнего узла и который представляет собой сервис, выполняющий другие исполняемые программные элементы.

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

Собрав воедино широко распространяющуюся объектно-ориентированную парадигму и методики с учетом ОО подхода, был создан унифицированный процесс разработки. Процесс представляет разработку в виде инкрементально-эволюционного процесса и разбит на фазы — начало, проектирование, построение и внедрение.

Также введено понятие дисциплин — набора правил и указаний, необходимых для решения определенной задачи. Дисциплины предназначены для организации разработчиков, дать им характерные подходы, чтобы каждая роль процесса разработки могла выполнять необходимые для нее действия.

Существует график процесса, показывающий уровень активности разработчиков в каждой из фаз для каждой дисциплины. Например, работы по дисциплине Deployment активнее всего ведутся на фазах построения и внедрения. Фазы могут делиться на итерации.

RUP определяет более 30 различных ролей. Роль — группа обязанностей, которую берет на себя участник разработки для выполнения своей деятельности. У каждой роли есть свой набор деятельности. На входе и выходе деятельности используются, создаются и модифицируются артефакты — любые результаты труда роли. Артефакты создаются на основании инструкций и шаблонов.

Деятельность всегда производится согласно установленным правилам и при помощи принятых в компании средств. Компании могут изменять процессы, RUP предлагает их лишь в качестве основы.

Каждый элемент процесса детально описан и связан с другими.

RUP, как программный продукт — набор страниц HTML, описывающих элементы процесса разработки. Для визуального представления используется UML.

Каждая фаза заканчивается вехой — моментом времени для принятия решения о дальнейших действиях. Решение принимается стейкхолдерами (заинтересованными лицами).

## 16. RUP: Фаза «Начало».

Основное направление фазы — оценить проект, понять, сколько нужно потратить ресурсов и времени, какие проблемы пользователей и как проект сможет решить. Формируется артефакт Vision.

Цели в фазе:

- Определение границ проекта, решаемых и не решаемых им задач.
- Разработка и описание основных сценариев использования системы. Необходимо выяснить как система будет использована и определить последовательность действий пользователя
- Предложение возможных технических решений



- Подсчет стоимости и разработка графика работ
- Оценка рисков и подготовка окружения разработки

Веха Объекты ЖЦ – определяет согласие сторон в оценке сроков, стоимости, требованиях, приоритетах и технологиях, а также оценку рисков и стратегий смягчения последствий.

#### 17. RUP: Фаза «Проектирование».

Основная задача фазы – разработка и тестирование стабильной и неизменной архитектуры системы, создание прототипов системы, которые определяют исполняемую архитектуру – полностью законченные на базе выбранных технологий несколько характерных функций разрабатываемой системы. Как только реализация новых требований перестает создавать дополнительные элементы архитектуры – один из возможных способов построения готов.

Тестирование на этапе проверяет основные нефункциональные требования к системе.

Уточняются планы разработки, производится переоценка и контроль рисков, уточняются сроки и стоимость системы.

Веха Архитектура ЖЦ – веха стабильности архитектуры. Определяется, что концепция, требования, архитектура проекта стабильны. Сформированы критерии тестирования прототипов. Тестирование показало отсутствие основных рисков. Планы разработки подробны и приемлемы по цене. Соотношение плановых и реальных расходов приемлемо, стороны подтверждают выполнимость проекта.

Заказчик может принять аргументированный перерасход средств.

#### 18. RUP: Фаза «Построение».

На этой фазе не может быть внесения кардинальных изменений в архитектуру проекта. Основная цель – экономически эффективно, качественно, как можно быстрее разработать продукт.

Ресурсы команды разработки не должны тратиться на неосновные работы и переделки. Изменения и улучшения в архитектуру отсутствуют или минимальны. Постоянно снижаются издержки команды.

Происходит создание необходимых выпусков продукта по плану проекта (альфа, бета...). Проводятся демонстрации заказчику. В конце фазы продукт подготавливается к передаче заказчику – обследование места установки, разработка учебных материалов, обучение пользователей.

Организовывается и проводится тестирование по плану, в котором описаны и метрики качества ПО.

Веха Начальная Операционная Способность – достаточно ли стабилен продукт для передачи пользователям, все стороны готовы и приемлемо ли соотношение затрат.

#### 19. RUP: Фаза «Внедрение».

Цель – запуск продукта в продуктивное использование.

Проводится бета-тестирование, сравнивается работоспособность версий. Переносится продуктивная БД, обучаются пользователи и поддерживающий персонал. Запускается реклама и продажи.

Процессы устранения сбоев, дефектов и проблем с производительностью отлаживаются.

Необходимо убедиться в самодостаточности пользователей и провести анализ соответствия продукта исходной концепции.

Веха Релиз продукта – удовлетворены ли пользователи и приемлемо ли финальное соотношение расходов.

## 20. Манифест Agile (2001).

- **Люди и взаимодействие важнее процессов и инструментов**
- **Работающий продукт важнее исчерпывающей документации**
- **Сотрудничество с заказчиком важнее согласования условий контракта**
- **Готовность к изменениям важнее следования первоначальному плану**

Способность удовлетворять постоянное изменение бизнес-требований явилось предпосылкой манифеста, декларирующего, что «Agile процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества»

Во главу подхода ставятся требования заказчика и реакция на них. Подход невозможен, если на проект выделен фиксированный бюджет без возможности расширения.

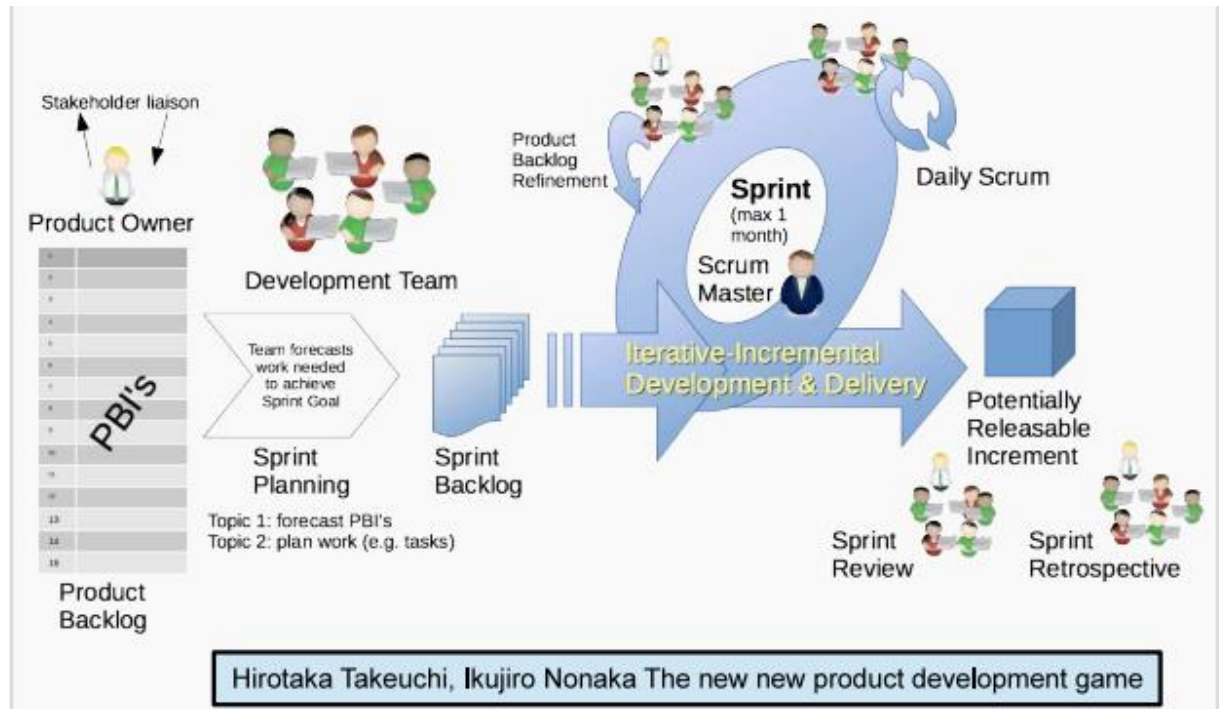
Гибкие методологии хорошо работают для внутренних проектов разработки в компаниях или когда заказчик покупает время разработчиков. Разработчики должны постоянно демонстрировать результат работ, результат оценивается, разработчики получают деньги и продолжают разработку.

Также важно сокращение расходов на труд разработчиков, не относящийся к созданию кода. В проектах с типовой архитектурой это не вызывает сложностей, в более сложных – порождает проблемы.

### 12 принципов Agile:

- Наивысшим приоритетом является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного ПО
- Изменение требований приветствуется даже на поздних стадиях
- Работающий продукт следует выпускать как можно чаще, от 2 недель до 2 месяцев
- Разработчики и представители бизнеса должны ежедневно работать вместе
- Над проектом должны работать мотивированные профессионалы, им нужно обеспечить поддержку, создать условия и довериться
- Непосредственное общение в команде
- Работающий продукт – основной показатель процесса
- Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно
- Техническое совершенство
- Простота – минимизация лишней работы – необходима
- Самоорганизация команд
- Систематическая коррекция стиля работы команды

## 21. Scrum.



Процесс разработки в скраме сильно упрощен. Основной и единственный служебный артефакт – бэклог – упорядоченный по приоритетам список требований с оценкой трудоемкости разработки. Есть бэклог продукта, обычно более общий и состоящий из бизнес-требований, и бэклог спринта – более детальный, с учетом технических особенностей реализации. Другой артефакт – инкремент продукта.

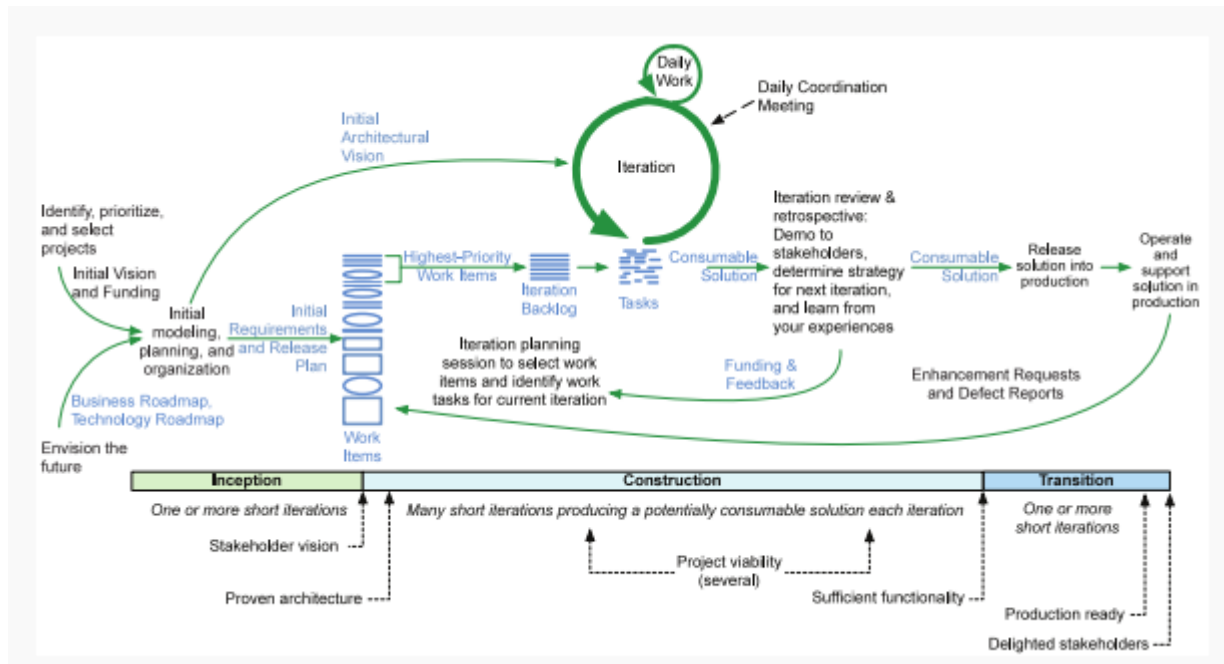
Спринт – время от 2 до 4 недель, за которое разработчики реализуют выбранный набор требований из бэклога спринта. Каждый спринт заканчивается демо версией, и проделанная работа показывается заказчику. Через каждые несколько спринтов проводятся ретроспективы, работа над ошибками и перераспределение обязанностей для повышения эффективности команды.

Команда в скрам небольшая, от 3 до 10 человек. Есть особая роль – владелец продукта – определяет порядок разработки требований из бэклога. Для каждого спринта задачи конкретизируются и передаются на разработку команде.

Также выделяется скрам мастер, ответственный за проведение скрам митинга, где планируется спринт, и следящий за отношениями в команде.

Достоинство скрама – простота, минимум административной работы и документов, концентрация на работоспособном коде. Лучше всего подходит для проектов с небольшой командой разработки.

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



Очевидная слабость гибких методологий – трудность масштабирования на команды с большим числом разработчиков. Решение этой проблемы – актуальная цель для рынка.

В 2013 году был предложен метод Disciplined Agile Delivery – подход к делению на фазы похож на RUP, но основной цикл разработки построен на базе гибких методов, в том числе Scrum. Методология находится в стадии развития.

Предлагается три фазы – Начало, построение и внедрение. Описывается каждая роль в разработке. Также рассматриваются процессы, выходящие за рамки процесса разработки: управление архитектурой, повторным использованием кода, персоналом, служба поддержки и текущих операций компании разработчика, управление портфолио компетенций, непрерывное улучшение процессов разработки и вспомогательных процессов и т. д.

Особое внимание в последнее время уделяется сбору и анализу большого объема данных, накопленных в рамках разработки большого количества разнородных проектов по технологиям BigData.

## 23. Требования. Иерархия требований.

Требования – условия или возможности, которым должна соответствовать система. Требование должно быть однозначно сформулировано, и грамотно и четко описывать то, что системе необходимо выполнять.

Требования могут быть общими или подробными. Требование не описывает как необходимо реализовать, оно лишь указывает на то, что нужно реализовать.

Первый способ описания требований – модель требований. Существует шаблон Software Requirement Specification. В дополнение к нему есть набор детализированных описаний прецедентов, описывающие последовательность действий пользователей. В них также могут быть предусмотрены прототипы графических интерфейсов взаимодействия с системой.

Второй способ описания – модели прецедентов (Use Case), описывающие требования в UML диаграммах. Лучше подходит для заказчиков, более нагляден.

По степени детализации:

- Потребности заинтересованных лиц. Пожелание для решения какой-то задачи. Вместе с пожеланиями заинтересованные лица предоставляют набор артефактов, которые можно использовать для иллюстрации информации, которую будет содержать система.
- Набор функций, формируется аналитиками на основе потребностей. Необходим большой формализм и четкость в формулировках, однако все еще должны быть ясны заказчику
- Требования к ПО: содержат детали реализации.

Требование должно быть корректным, однозначным, полным, непротиворечивым, ему можно задать приоритет, его можно проверить, изменить, отследить и оно должно ссылаться на свой источник (почему это требование нужно).

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

Ключевое слово **должна** указывает на безусловную необходимость.

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

В RUP используется модель FURPS+ для описания требований.

Требования делятся на 2 типа:

- Функциональные – что система должна делать
- Нефункциональные – характеристики и ограничения, накладываемые на систему

Нефункциональные требования делятся на:

- Требования к пользовательским характеристикам ПО (Usability)
- Требования к надежности системы и способности ее обеспечить требуемую готовность к выполнению операций (Reliability)
- Требования к производительности ПО (Performance)
- Требования к условиям поддержки (Supportability)

Также требования могут включать ограничения на использование интерфейсов, требования к реализации, физические требования.

#### 25. Формулирование требований. Функциональные требования.

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

ID указывает на тип и номер требования, например FR1 – первое функциональное.

Функциональные требования определяют:

- Feature Set – набор свойств продукта, необходимый для выполнения конкретной деятельности. Функционал продукта.
- Capabilities – возможности ПО
- Также могут определять требования к безопасности: метод аутентификации, список ролей, шифрование, меры по защите от злоумышленников и т. п.

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

В требования к юзабилити обычно входят особенности использования, которые необходимо учесть при разработке ПО.

Человеческий фактор, учет физических особенностей человека, например время отклика должно быть в диапазоне 1–5 секунд.

Эстетические требования, если определены, должны быть подробными и сложными. Часто в компаниях существуют справочники по корпоративному стилю.

Мастера настройки и ПО могут потребоваться для упрощения действий пользователя для типовых задач. Сюда же требования к подсказкам, документации, учебным материалам.

Для реализации этих требований существует подход User Experience Design (UX).

Требования к надежности предназначены для фиксирования способности ПО безотказно функционировать в течение определенного периода времени. Отказ системы – неспособность выполнять основную функцию. Также есть сбои – случаи неспособности выполнять отдельный функционал при сохранении общей работоспособности системы. Для разработки требований необходимо категорировать их по степени критичности для заказчика, а также определить реакцию системы и обслуживающего персонала на сбои и отказы.

Обычно указывают допустимое число отказов и сбоев за определенный промежуток времени, вносятся штрафы за превышение значений. Также указывается Recoverability – способность системы восстанавливать продуктивное функционирование за определенное время.

Accuracy – точность, например, проведения вычислений. Важно для мат. Моделей управления физ. Системами.

MTBF – среднее время между отказами, часто единый параметр характеристики надежности системы.

Коэффициент готовности системы – отношение времени исправной работы к сумме времен исправной работы и вынужденных простоев объекта за тот же срок.

## 27. Требования к производительности и поддерживаемости.

Требования производительности в первую очередь включают в себя скорость решения вычислительных задач. Особо важно в системах реального времени.

Требования к эффективности фиксируют процент времени, которое тратится на выполнение полезных задач, по отношению к времени на выполнение общесистемных.

Важное требование к готовности (Availability) быстро начать выполнение задачи, иногда называется реактивностью системы. Подразумевает учет архитектурных особенностей систем, которые предполагают определенные задержки в связи с диспетчеризацией процессов. Также архитектура ПО должна быть организована с целью уменьшения времени начала обслуживания.

Пропускная способность показывает, какой объем данных или запросов система может обработать за единицу времени. До точки насыщения в правильно построенной системе время отклика системы растет линейно, после чего рост экспоненциальный.

Различные параметры связаны между собой, ради большей реактивности приходится жертвовать пропускной способностью.

Также здесь есть требования к использованию ресурсов.

Требования к поддерживаемости включают в себя:

- Расширяемость
- Адаптируемость под конкретные задачи
- Поддерживаемость
- Совместимость
- Конфигурируемость



- Возможность проведения профилактик и обслуживания
- Требования к установке на разные системы
- Локализуемость и интернационализуемость

Одним из основных является способность к масштабированию и расширению. Она защищает инвестиции заказчиков в ПО, позволяя использовать одну и ту же систему в условиях возрастающей нагрузки.

Для обеспечения способности поддерживаемости группой обслуживания требуются специальные архитектурные решения.

Требования к совместимости позволяют использовать различные ОС, браузеры и т. д. с разработанным ПО.

## 28. Атрибуты требований.

Требования могут быть помечены атрибутами для сортировки.

Одним из основных является приоритет требования, часто по критерию MoSCoW.

M – Must Have – фундаментальные требования, без которых система не имеет смысла.

S – Should Have – важные требования, которые следует реализовать при наличии времени в процессе.

C – Could Have – потенциально возможные; улучшения

W – Will not have\Would like to have – могут быть реализованы в будущих версиях, но сейчас времени нет

Также приоритет может указываться цифровым значением.

Другой атрибут – статус: предложенное, одобренное, отклоненное и т. п.

Трудоемкость – часто указывается в человеко-часах, функциональных точках или use case точках. Также есть оценка попугаями – безразмерной единицей, одно из требований берется за эталонное и далее другие требования оцениваются относительно него. Время, потраченное на эталонное требование, берется за одного попугая.

В атрибутах также может быть риск – факторы, влияющие на разработку требования, стабильность – частота изменения требований, целевая версия.

## 29. Описание прецедента.

Прецедент: PieSelling
<b>ID:</b> 2
<b>Краткое описание:</b> Бабушка продаёт пирожки.
<b>Главные актёры:</b> Бабушка-продавец, Клиент (или любой другой пользователь).
<b>Второстепенные актёры:</b> нет.
<b>Предусловия:</b> Клиент знает, какой пирожок он предпочитает. Бабушка проверила весь ассортимент. У клиента достаточно средств.
<b>Основной поток:</b> 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.



В процессе дальнейшей разработки требований диаграммы прецедентов используются только как изначальная модель. Требования к системе формулируют текстовым описанием, графическим интерфейсом пользователя и др.

Есть основной поток событий. Ключевое слово extend выражается альтернативным сценарием.

### 30. Риски. Типы Рисков.

Риск – потенциально опасный фактор, влияние неопределенности на цели, сочетание вероятности события и его негативных последствий. Определения различны.

Риски могут быть прямыми и косвенными – прямыми можно в явном виде управлять: воздействовать на них, уменьшать вероятность, реагировать. Косвенные риски возникают по внешним причинам и не поддаются управлению.

Виды рисков по их источнику и особенностям:

- Ресурсные риски – недостаток времени, людей, денег, оборудования. Являются управляемыми.
- Бизнес-риски – взаимодействие с другими организациями и рынком, конкуренция, потенциальная убыточность решения. Сложно управляемы со стороны разработчиков
- Технические риски – управляемые разработчиками. Границы проекта, внешние зависимости, компетенции разработчиков по применяемым технологиям.
- Политические риски – изменение сфер влияния внутри компании. Иногда управляемы.
- Форс мажоры.

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

В процессе оценки риска специалист по работе с риском проводит знакомство с риском, его анализ, определяет степень серьезности и планы работы с риском. Внутри оценки производятся: идентификация, анализ и приоритезация риска.

После этого риск можно поместить под управление различных систем контроля риска.

Идентификация рисков может производиться по схеме источников рисков. Источники распределяются по иерархической структуре, первый уровень – классы рисков (окружение, разработка, ограничения). Классы состоят из элементов риска, а элементы из атрибутов, которые указывают на возможные источники возникновения риска.

Также риски делят на известные, неизвестные и непознаваемые. Место возникновения неизвестного риска можно предположить, но команда еще с ним не сталкивалась. Источник непознаваемых рисков нельзя предугадать.

Всесторонний анализ рисков выявляет скрытые взаимосвязи ситуаций и источников рисков. Методы анализа – стоимостной (расчет количественного выражения влияния риска), сетевой, качественных факторов.

Основные параметры риска – вероятность наступления и масштаб. Параметры обычно задаются неточно и субъективно, например по пятибалльной шкале.

Также можно задать риск шкалами по группам параметров в визуально понятной форме.

Приоритет риска может выставляться на основе экспозиции риска – произведения его вероятности на масштаб последствий. После этого создается документ Топ 10 рисков, он постоянно обновляется.

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

Управление рисками включает в себя планирование реакции, разрешение риска и мониторинг рисков.

Планирование реакции: первый способ реагирования – избежать риск, разработать комплекс мероприятий по отсрочке или исключения вероятности его наступления.

Второй способ – перенос риска, сделать так, чтобы под риск попал другой человек, организация и т. п.

Можно сократить вероятность наступления риска, также возможен прием риска. Тогда составляются планы действий в случае наступления риска.

Разрешение риска подразумевает мероприятия по предотвращению, возможно:

- Построение как можно большего числа прототипов системы. Тогда заказчик сможет более полно оценить то, что получит в результате. Число прототипов снижает вероятности рисков, однако затратно.
- Построение различных моделей функционирования ПО. Построение моделей дешевле, чем разработка ПО. Существует большое число средств симуляции пользовательского интерфейса.
- Аналитическая работа, в том числе над ошибками команды.
- Грамотный подбор персонала.

Мониторинг рисков подразумевает переоценку рисков в течение разработки. Принято концентрироваться на 10 наиболее вероятных и деструктивных рисках. Большую помощь могут оказать автоматизированные системы контроля рисков (плагины Jira).

### 33. Изменение. Общая модель управления изменениями.

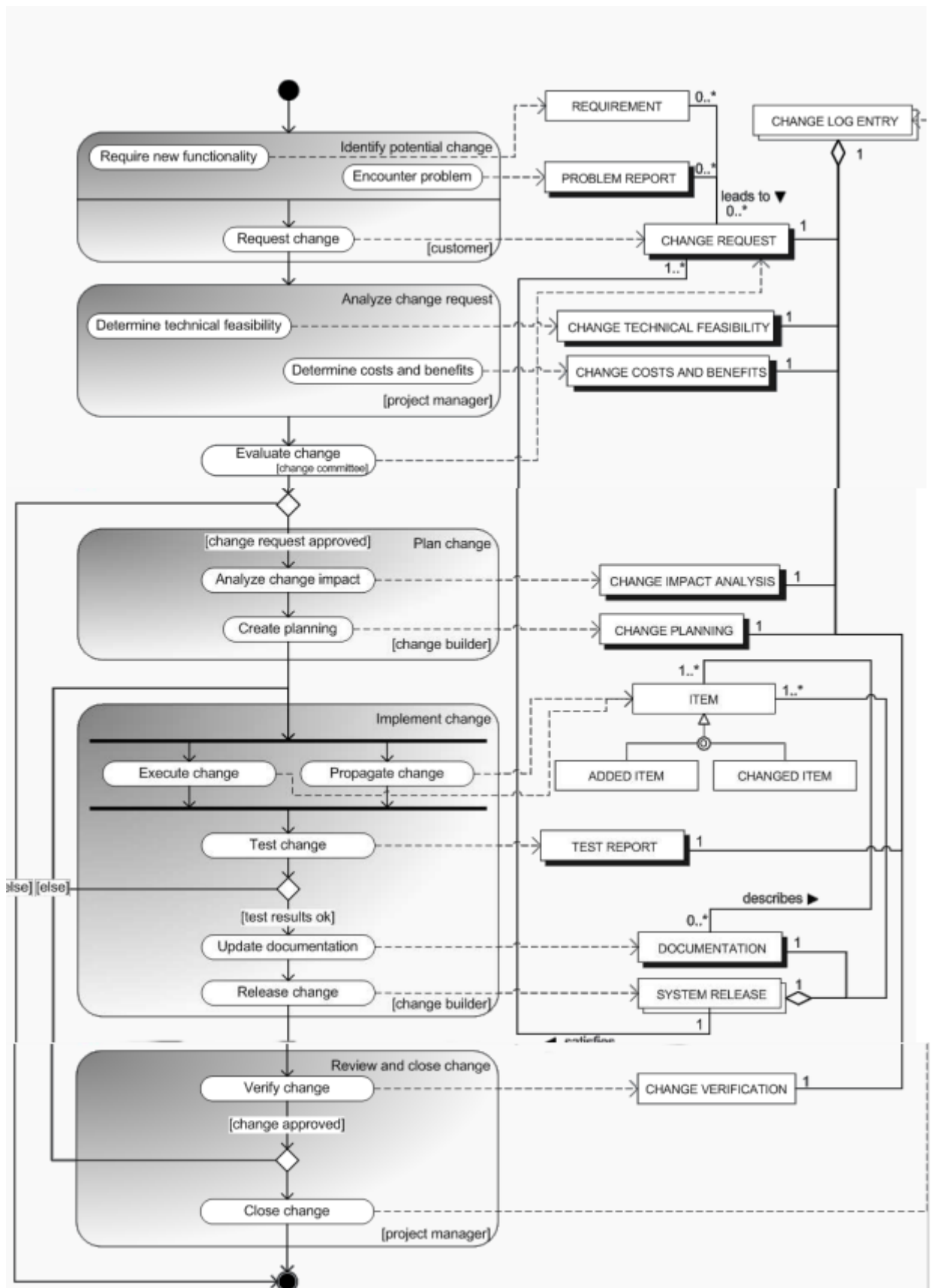
Изменение – контролируемое, журналируемое обновление системы.

Вносимые изменения выстраиваются в последовательность отдельных действий, которые необходимо контролировать и фиксировать атрибуты каждого изменения.

Используются системы контроля версий, с помощью них можно посмотреть историю правок и оценить их влияние.

Атрибуты изменения – идентификатор, дата, ответственный, описание, связанный журнал изменения и т. д.

Под контроль изменений попадают изменения не только кода, но и требований, дефектов, артефактов, анализа и дизайна.



В левой части схемы указаны активности ролей процесса формирования изменения. Справа артефакты изменения.

Заказчик вносит изменения в систему ради нового функционала или исправления проблемы. Деятельность приводит к созданию документа или артефакта – требования к системе, отчет об

ошибке или проблеме. Оба типа формируют запрос на изменения и на каждый запрос формируются записи в Change Log Entry.

Далее запросы поступают к менеджеру проекта, он определяет технологическую необходимость и осуществимость изменений, а также анализирует стоимость и преимущества для пользователя.

Далее запрос поступает комитету по изменениям, который оценив необходимость и стоимость, меняет статус изменения в Change Request. Если запрос подтвержден, изменение и его реализация анализируются.

Далее анализируется возможное влияние изменения на систему и последствия для пользователей. Организуется планирование проведения изменений в системе, определяются ресурсы, время и график действий.

Затем изменение передается на реализацию, Change Builder создает объекты, совокупность которых составляет реализацию изменения. Далее происходит тестирование, вносятся изменения в документацию и выпускается новая версия продукта. Создаются артефакты – отчет о тестировании, документация и выпуск системы.

В конце изменения передаются менеджеру проекта или заказчику, проверяются и утверждаются.

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

Системы контроля версий (СКВ) существуют для управления изменениями в программном коде и поддерживают групповую работу нескольких человек над кодом одновременно.

Есть три основных типа СКВ:

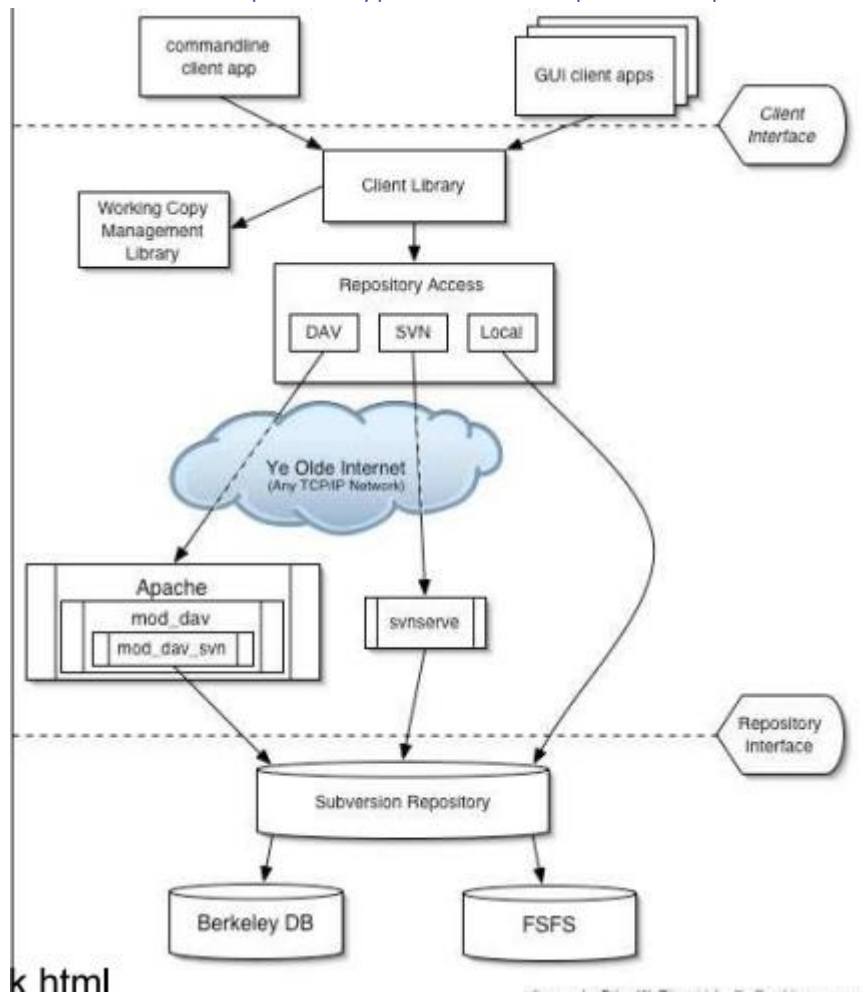
- На основе файловой системы – устаревший подход. Разработчики пользовались центральным сервером с общим доступом к файлам, создавались файлы слежения за текущей директорией, версии отслеживались только в рамках одной файловой системы. Можно было экспортировать файловую систему для доступа удаленных клиентов при помощи сетевых файловых систем (например, NFS)
- Централизованная, с единым репозиторием – хранилищем исходных кодов проекта на сервере, и удаленным доступом клиентов по специальным протоколам.
- Распределенная – существует центральный репозиторий, из которого пользователи скачивают данные в локальные репозитории. Обратно в центральный репозиторий данные попадают после локальных проверок.

В первых СКВ действия производились в командной строке, в большинстве современных СКВ есть встроенные клиенты, автоматизирующие работу с версиями и организующие необходимые действия в набор пунктов меню.

Главной проблемой при работе с СКВ является одновременное редактирование одних и тех же файлов. Всего утвердились два подхода:

- Lock-Modify-Unlock, где при работе одного пользователя с файлом он блокируется для других и они не могут его модифицировать. Работа команды замедляется. Был характерен в СКВ на файловой системе.
- Сорудify-merge, где каждый пользователь копирует себе репозиторий и работает с ним, а изменения всех пользователей сливаются. В данном случае проблема именно в процессе слияния, так как приходится решать конфликты, возникающие между разными версиями файла.

### 35. Subversion. Архитектура системы и репозиторий.



SVN является централизованной СКВ. В архитектуре существует уровень хранения репозитория, который может иметь 2 технические реализации – БД Berkeley или в простейшем случае в файловой системе.

Доступом управляет демон `svnserve`, получающий команды пользователя и выполняющий соответствующие изменения в репозитории. Второй вариант доступа – сервер Apache и его модули, реализующие необходимую логику.

Удаленный доступ обычно осуществляется по протоколу `svn`. Удаленный доступ к репозиторию может осуществляться также по протоколам `svn+ssh` и `svn+https`. Этими протоколами может пользоваться клиентская библиотека, которую используют приложения с интеграцией Subversion.

Клиент SVN также осуществляет управление локальной копией файлов. Когда вы скачиваете выбранную версию из репозитория, локальный набор файлов должен быть изменен в соответствии с содержимым выбранной версии.

Репозиторий – набор файлов проекта, организуемый определенным иерархическим образом для удобной работы.

- **trunk** — основная ветвь разработки.
- **branches** — хранение модификаций продукта:
  - Releases — значимые версии продукта (3.0,3.5).
  - Features — выполнение работ над версиями со существенными изменениями без влияния на trunk.
  - Vendor — версии с модификациями сторонних библиотек. *Vendor drop*.
- **tag** — функционально целостные изменения:
  - Исправления дефектов ПО (3.5.0-B2947219).
  - Минорные версии (с исправлением нескольких дефектов — 3.5.0.1).

В SVN каждая фиксация изменений повышает версию репозитория на 1. Ревизия репозитория всегда целое уникальное число. Каждая фиксация обязана включать все файлы, изменения которых должны быть помещены в репозиторий. Основной процесс разработки происходит в каталоге trunk, туда вносят ежедневные результаты труда в виде целостного набора изменений.

В branches хранятся отдельные модификации и структурно-значимые версии продукта. Также там можно вести разработку отдельных ветвей реализации, в которых внутренний функционал изменен по сравнению с trunk. После завершения изменений содержимое веток сливается в главную ветку.

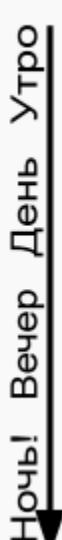
Также есть Vendor – с модификациями сторонних библиотек, используемых в проекте.

В tags хранятся помеченные версии продукта, например для работы с дефектами.

Для создания новой ветки в SVN производится полное копирование файлов старой ветки в новый каталог.

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

**svn checkout** — первоначальное создание рабочей копии.

- 
- Обновить рабочую копию:
    - **svn update**
  - Изменить необходимые файлы:
    - **svn (add, delete, copy, move, mkdir)**
    - Просмотр изменений: **svn status** и **svn diff**
    - Откат изменений: **svn revert**
  - Фиксация изменений на сервере:
    - **svn update** для загрузки изменений других.
    - Разрешить конфликты содержимого и структуры файлов.
    - **svn commit** — собственно фиксация.

Для начала работы необходимо обновить свою рабочую копию, при этом скачивается и управляющая информация для обеспечения целостности и определения того, куда производятся коммиты.

Затем происходит работа. Для этого используются команды со слайда.

Вечером проходит фиксация изменений на сервере, необходимо загрузить новую версию и решить конфликты с другими разработчиками. На основе вечерних коммитов создаются ночные сборки.

### 37. Subversion: Конфликты. Слияние изменений.

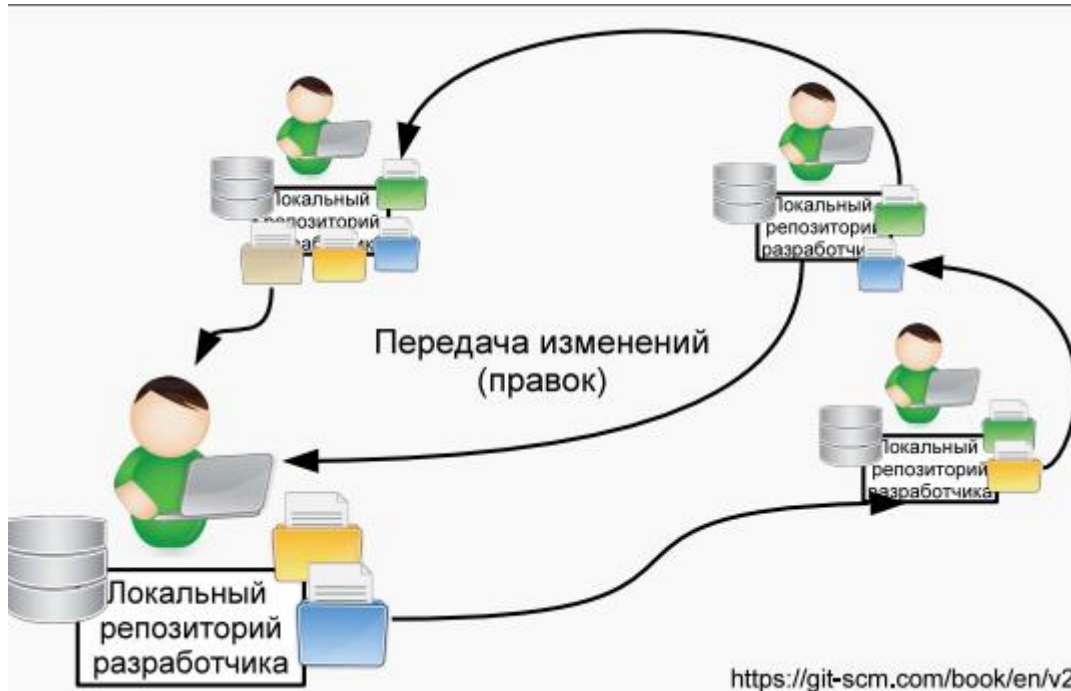
Основная проблема при фиксации изменений – конфликты. Могут быть конфликты содержимого, когда файлы содержат различный код не сливающийся друг с другом. Могут быть конфликты структуры, когда, например, в репозитории перемещаются файлы. Не совпадает структура программ и файлы могут быть переименованы, перемещены или удалены. Для разрешения конфликтов могут использоваться следующие средства:

- (e) edit — изменить файл в редакторе.
  - (df) diff-full — показать список несовпадений между версиями.
  - (r) resolved — подтвердить, что конфликт разрешен в текущей (м.б. отредактированной) версии файла.
  - (dc) display-conflict — показать все конфликты.
  - (mc) mine-conflict, (tc) theirs-conflict — подтвердить мои (или из репозитория) версии для всех конфликтов.
  - (mf) mine-full, (tf) theirs-full — подтвердить версии полностью для файла.
  - (p) postpone — отложить «на потом».
  - (l) launch — запустить внешнее средство.
- ```
$ svn update
Updating '.':
U    Pokedex.xml
G    Slowimperator.java
Conflict discovered in 'Slowking.java'
Select:
```
- filename.mine — до update  
filename.rOLDREV — до правок  
filename.rNEWREV  
— в репозитории

Слияние между ветками производится командой svn merge.

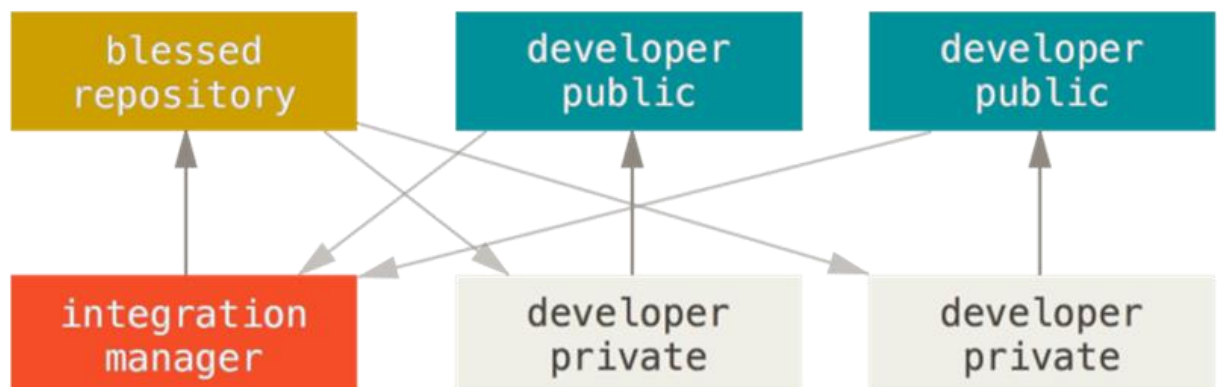


### 38. GIT: Архитектура и команды.



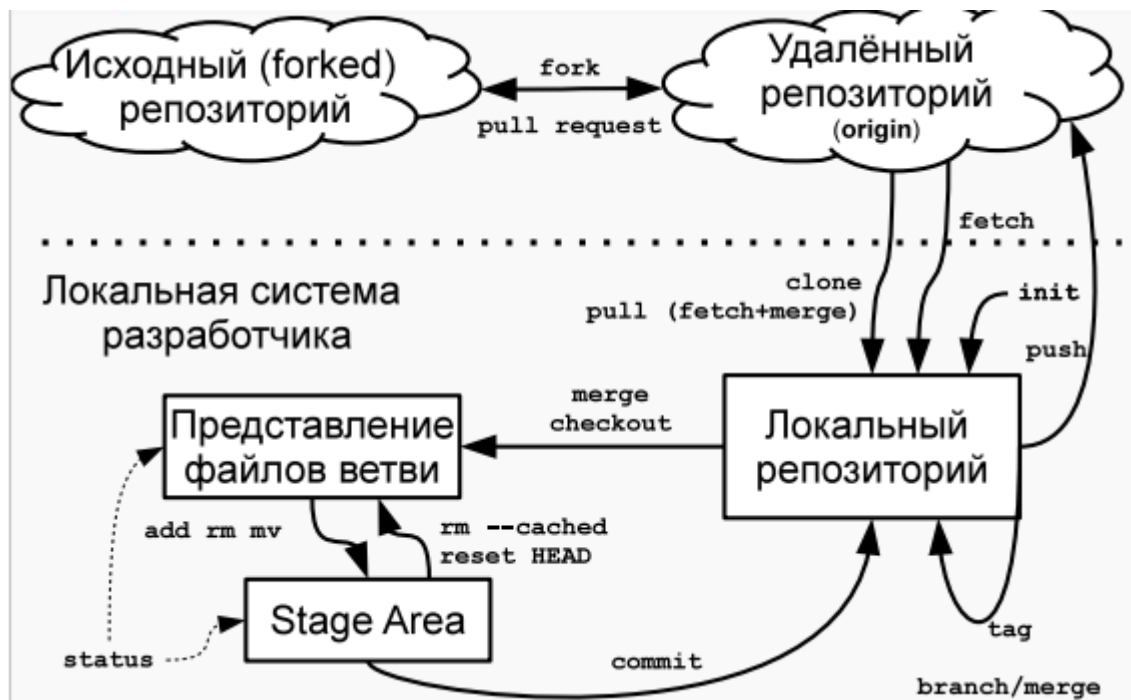
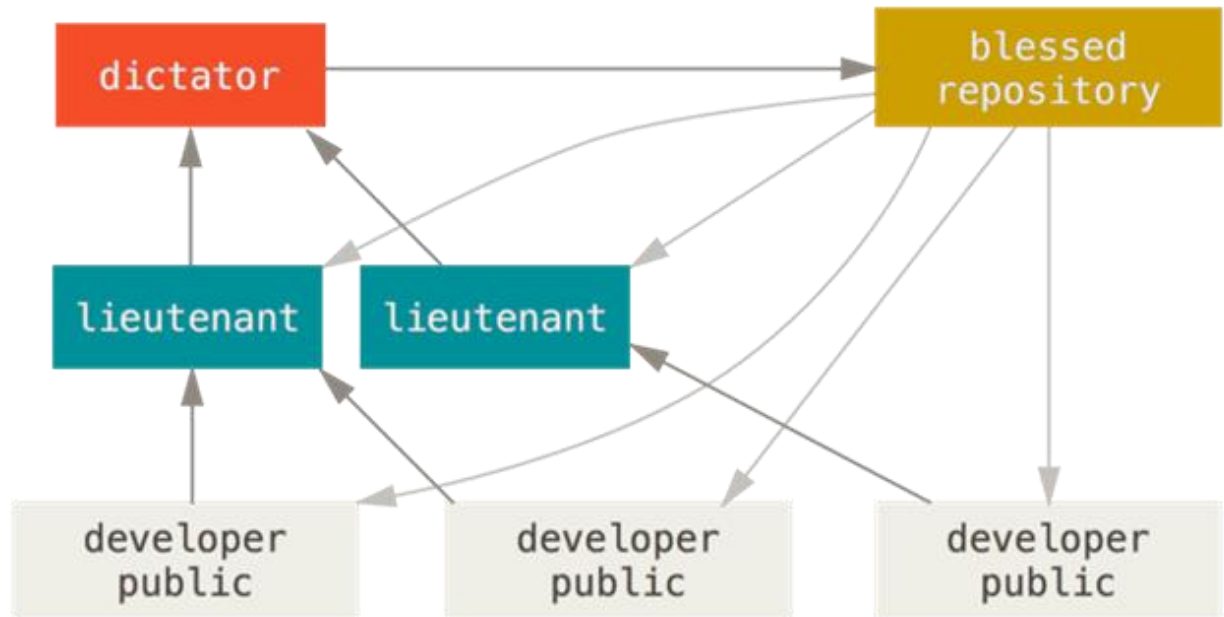
Git - распределенная СКВ. Репозитории существуют локально у каждого разработчика. Программист может предложить свои правки другим разработчикам. Каждый репозиторий имеет свой URL, по которому он доступен. Архитектура позволяет адаптировать процесс для каждой конкретной команды. Сейчас определены три основных рабочих процесса с Git:

- Централизованный – существуют единственный центральный репозиторий и разработчики синхронизируют работу с ним.
- С менеджером по интеграции – каждый разработчик изменяет локальную копию, которую он открывает другим для чтения, после завершения правок запрашивается интеграционный менеджер, который загружает правки из копии для чтения в канонический репозиторий.



- С диктатором и лейтенантами – используется на огромных проектах. Лейтенанты – интеграционные менеджеры, отвечающие за отдельные части репозитория. У лейтенантов есть свой интеграционный менеджер – доброжелательный диктатор, его репозиторий

выступает как эталонный, откуда все участники процесса должны получать изменения.



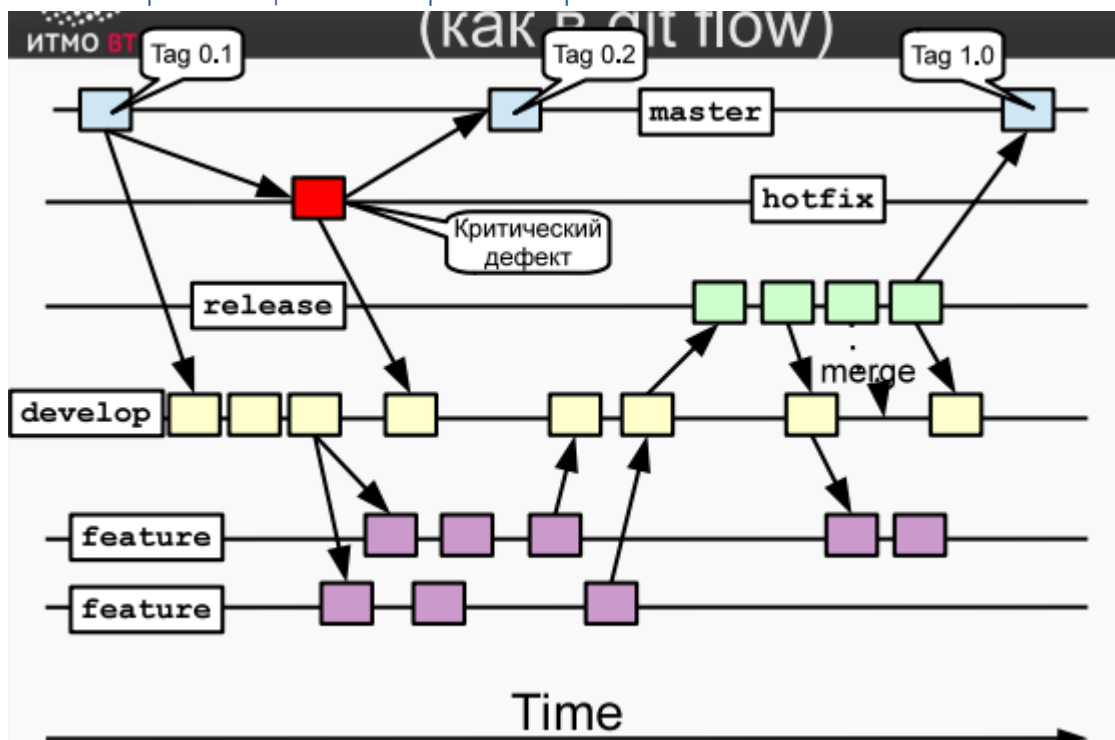
Центральный репозиторий может быть получен с помощью fork. Во время работы с локальной копией можно откладывать изменения с помощью stash, тогда файлы будут убраны из Stage Area. Все содержимое Stage Area включается в коммит.

Способ фиксации изменений в удаленном репозитории – pull request – изменения будут сначала подтверждены владельцем ветви.

Каждая команда создает изменения и подсчитывает хеш изменений.

- `git status` — показывает статус текущих состояний файлов в файловой системе и информацию о ветви, в которой производится редактирование.
- `git log` показывает журнал коммитов, а опция `--graph` выводит в графическом виде ветви, в которых производились данные изменения.
- `git diff` — покажет все изменения, которые были сделаны относительно последних зафиксированных изменений.
- `git reset --hard HEAD` сбросит все изменения, которые были сделаны в текущем локальном репозитории.
- `git branch` — показывает ветви.
- `git checkout` — переключает разработчика между ветвями
- `git merge` — объединяет несколько ветвей в текущую ветвь
- `git commit` — фиксирует изменения в текущей ветке. Опция `-m` задает сообщение коммита, которое будет показываться пользователю
- `git add` — добавляет измененные файлы к последующему коммиту, помещая их в Stage Area.

### 39. GIT: Организация ветвей репозитория.



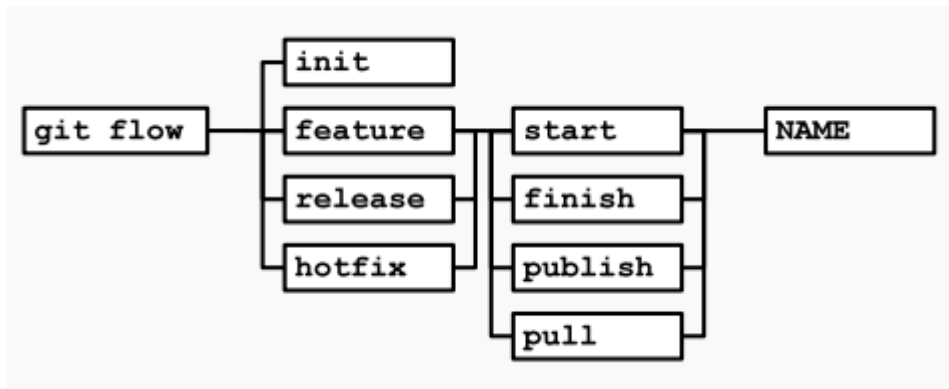
В ветви Master находятся версии, готовые к поставке пользователю – основные версии продукта с определенным функционалом и известным списком дефектов.

Основная разработка происходит в ветви Develop. Она использует Master как базовую. Все разработчики вносят изменения именно в эту ветвь. Для каждого из требований создается своя ветвь Feature, по мере завершения разработки функционала коммиты подгружаются в ветвь разработки.

Когда накоплено достаточно изменений формируется ветвь Release для исправления дефектов, препятствующих выпуску новой версии продукта. Далее готовая версия копируется в основную ветвь. Исправления из Release также вливаются в Develop.

Ветвь Hotfix нужна для исправления критических ошибок в версии пользователя. Исправления передаются в ветви основную и разработку.

#### 40. GIT: Плагин git-flow.



Чтобы упростить работу с Git был создан Git Flow, позволяющий работать с версиями именно в терминах версий, а не операций, без точного знания команд и последовательности действий.

Для использования плагина необходима определенная начальная настройка репозитория, есть функция `init`, которая задает пользователю несколько вопросов и проводит настройку.

Для создания новой feature в плагине используется команда `git flow feature start FEATURE_NAME`.

Когда нужно указать конец изменений - `git flow feature finish FEATURE_NAME`. Остальные команды аналогичны.

Это позволяет пользоваться репозиторием на более высоком уровне.

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

Даже в простейшем случае при сборке продукта приходится многократно повторять последовательности из нескольких команд, а если приложение состоит из модулей и имеет внешние зависимости последовательность становится еще сложнее и зависит от разных условий.

Можно создавать скрипты на уровне операционной системы, однако их сложно поддерживать, любое изменение в структуре проекта нужно отразить в скрипте. Системы сборки же генерируют скрипты автоматически.

Также, среда разработки часто отличается от среды, где будет использоваться продукт. Отличия могут состоять в аппаратном обеспечении, например отличия в системах с разной разрядностью, что влечет различия в размере минимальной единицы информации, что влияет на быстродействие программы и на необходимость генерации другого кода для целевой системы. Отличия в разрядности обуславливают отличия в разрядности адреса, что влияет на возможности работы с памятью, что повлияет на быстродействие некоторых алгоритмов.

Большое значение имеют размеры и организация кэшей. При попадании данных в кэш производительность увеличивается резко.

Также есть различия, вносимые операционной системой. Стандартные библиотеки ОС могут находиться в разных местах целевой системы и иметь отличные версии. Из попытки создать универсальные библиотеки ОС появились стандарты POSIX.

Также нужно учитывать, где размещается конфигурация системы для системных служб, для обеспечения платформ независимости ПО.

Сборка проекта может быть долгим процессом, особенно много времени может уходить на оптимизацию кода компилятором. Для ускорения процесса можно его распараллелить, однако невозможно одновременно компилировать файлы с зависимостями друг от друга.

Есть также системы многомашинной сборки, здесь сложность возникает с организацией доступа сборочных узлов к общим исходным кодам, использованием одинаковой версии компиляторов и других утилит сборки, размещением результата в одном репозитории.

Длительность процесса повлияла на расписание полной сборки больших программных продуктов, появились ночные сборки.

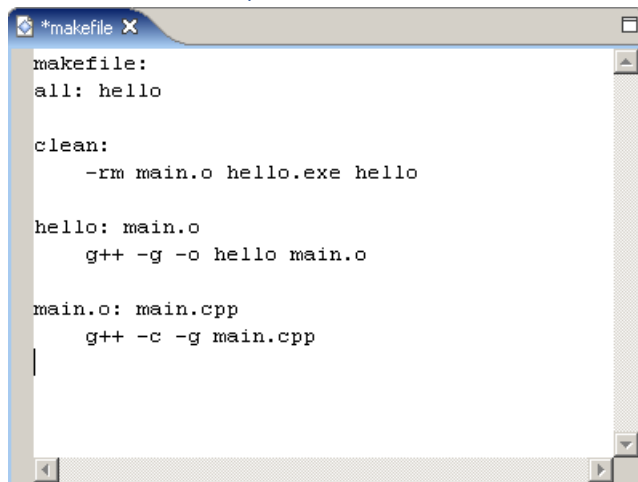
Современные системы сборки обеспечивают выполнение процесса сборки под управлением специальных языков, уменьшающих трудоемкость поддержки процесса разработчиками. В идеале скрипты должны быть сгенерированы автоматизировано.

Системы сборки также могут задавать и определять конфигурацию и архитектуру сборочной и целевой систем, возможности максимально эффективной утилизации ресурсов системы и т. п.

Для возможности непрерывной интеграции разработаны специальные сборочные серверы, которые запускают сборку при внесении изменений в репозиторий, проводят тесты системы и хранят все сборки ПО.

Все это позволяет сделать процесс сборки более простым, понятным и контролируемым.

## 42. Системы сборки: Make и Makefile.



Make впервые появилась в Unix экосистеме и остается актуальным как средство сборки низкого уровня. В файле сборки Makefile указываются опции компиляции и зависимости, необходимые для работы make.

В Makefile прописываются цели, которые нужно собрать, после них через двоеточие указаны файлы, от которых зависят эти цели, а на следующей строке – последовательность действий для сборки цели. Так make строит граф зависимостей для сборки проекта.

При запуске утилита анализирует файл, начиная со стандартной по умолчанию цели all, после чего собирает программы с учетом зависимостей. Также учитывается время последней модификации файлов, измененные после последней сборки файлы будут пересобраны.

Так, Make – язык, в неявном виде позволяет определять последовательность действий при сборке, и шаги, которые необходимо при этой сборке выполнить. К сожалению, он не очень удобен, необходимо следить за многими макропеременными и действиями по умолчанию, определенными в самой системе.

There are many [automatic variables](#), but often only a few show up:

```
hey: one two
# Outputs "hey", since this is the first target
echo $@

# Outputs all prerequisites newer than the target
echo $?

# Outputs all prerequisites
echo $^
```

#### 43. Системы сборки: Ant. Команды Ant.

```
<project name="World" basedir=".>
  <target name="init">
    ...
  </target>
  <target name="build" depends="init">
    ...
  </target>
  <target name="dist" depends="build">
    <antcall target=""/>
    <ant antfile="" dir="" target=""/>
  </target>
</project>
```



Ant – императивная система сборки, управляется файлом сборки build.xml

Используется широко для сборки java программ. Проект в файле сборки описан в формате XML.

Язык не очень удобен для чтения человеком, однако идеален для машинной обработки. Описание проекта состоит из именованных целей с явным указанием зависимостей – других целей, что позволяет создать граф зависимостей. Внутри каждой цели определены действия, которые необходимо выполнить для сборки.

Цели можно вызывать в явном виде (antcall). Возможен вызов целей из другого файла, что позволяет собирать систему из модулей.



- **Неизменяемые значения. Могут задаваться:**

- **Значением:**

```
<property name="build.dir" value="/tmp"/>
```

- **Переменной окружения:**

```
<property environment="env"/>
```

```
<echo message="Shell path: ${env.PATH}" />
```

```
<echo message="Build to: ${build.dir}" />
```

- **Загружаться из файла:**

```
<property file="build.properties"/>
```

Подстановка переменной



Есть переменные (свойства) с информацией о проекте, которые необходимо менять, чтобы собирать проект в разных условиях.

Свойства могут задаваться прямым значением, ссылкой на переменную окружения, загружаться из файла. После указания значения переменной, на нее можно ссылаться с использованием специального синтаксиса.

В Ant есть множество команд (task). Все собственные команды Ant задаются в виде XML. Это делает сборку независимой от особенностей поведения команд операционной системы, однако синтаксис команды надо каждый раз уточнять.

#### 44. Системы сборки: Ant-ivy.

- **Легко добавить в build.xml:**

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

- **Зависимости можно задать в файле ivy.xml:**

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

Ant Ivy – менеджер зависимостей для Ant. До него в Ant зависимости приходилось собирать вручную. Сторонние библиотеки скачиваются через Ivy, в частности, можно работать с репозиториями Maven2.

Ivy легко добавляется в проект, после чего он скачивает отсутствующие зависимости из удаленного репозитория. Зависимости задаются в отдельном файле – ivy.xml.

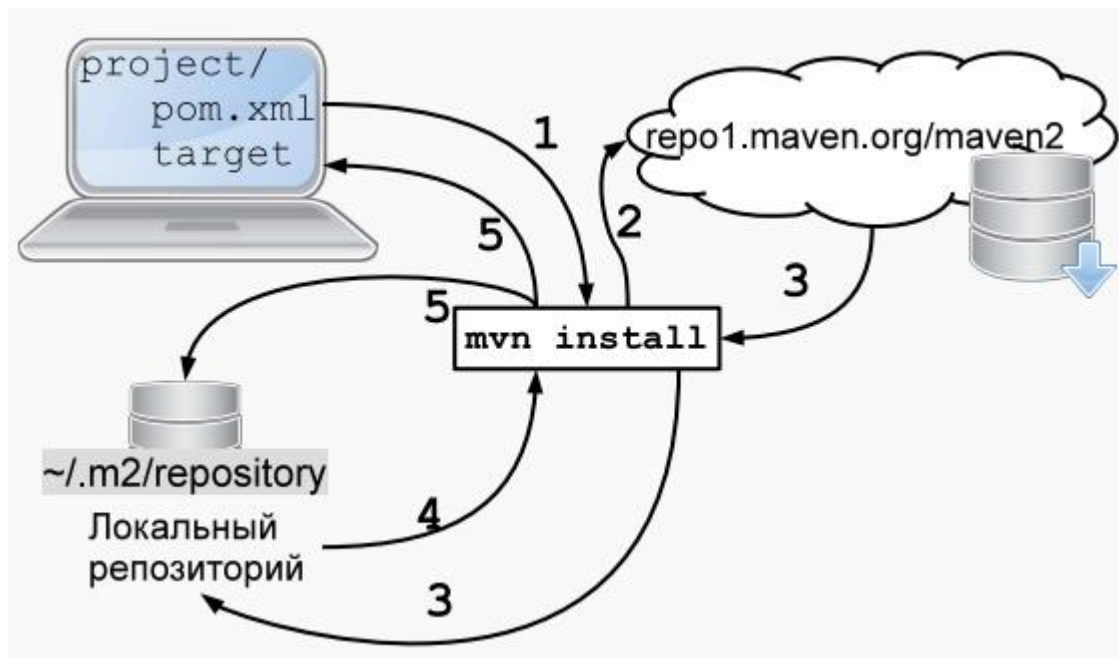


#### 45. Системы сборки: Maven. POM. Репозитории и зависимости.

Императивные системы сборки хороши, пока удобно последовательно описывать сборку каждой из частей программы. Однако со временем правильную последовательность становится сложно отслеживать.

Декларативные средства сборки – решение. Большинство действий выполняется не прозрачно для разработчика, а вызываются через интерфейс верхнего уровня декларативно. Мы указываем, что мы хотим собрать, а не как. При этом возможность модификации выполняемых команд остается.

Maven – декларативная система сборки. Управляется на базе описания проекта в POM.xml. POM – Project Object Model – содержит имя, версию и тип программы, местонахождение исходников, зависимости, используемые для сборки плагины и альтернативные конфигурации проекта – профили.



Декларативный подход стал популярен, когда появилась концепция репозиториев.

Принцип работы Maven на картинке. Цель по умолчанию – install. Ее назначение в том, чтобы в локальном репозитории пользователя была собрана версия продукта.

Сначала читается файла pom.xml с информацией о проекте и определяются зависимости проекта от внешних продуктов. После этого необходимые файлы скачиваются с удаленного репозитория и сохраняются в локальный репозиторий. Затем происходит сама сборка и в итоге готовый продукт помещается в директорию target и в локальный репозиторий, после чего можно отправить продукт в удаленный репозиторий.

Все цели в Maven частично скриптовые, а частично описаны внутри кода модулей на основе своих POM. По умолчанию сборка кода обычная и последовательная, но ее можно распараллелить при необходимости.

#### 46. Maven: Структура проекта. GAV.

В Maven существует система каталогов проекта по умолчанию.

- target — рабочая и целевая директории.
- src/main — основные исходные файлы:
  - src/main/java — исходные файлы java.
  - src/main/webapp — web-страницы, jsp, js, css...
  - src/main/resources — файлы, которые нет необходимости компилировать.
- src/test — исходные файлы для тестов:
  - src/test/java
  - src/test/resources

Система наименования модулей строится по принципу GAV – groupId:artifactId:version. Любая внешняя зависимость и продукт пользователя описывается таким способом.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>javax.portlet</groupId>
  <artifactId>portlet-api</artifactId>
  <version>2.0</version>
</project>
```

Таким образом в репозитории все проекты хранятся иерархически.

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

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

**Bold — по умолчанию**

Зависимости в Maven описываются в GAV синтаксисе и скачиваются автоматически.

У зависимости есть scope – область действия. Указывает, в какой момент жизненного цикла приложения применяется зависимость. По умолчанию – compile – во время компиляции. Test – во время тестирования. Runtime – в рантайме.

Scope=provided используется в Enterprise Java. Данный скоуп указывает, что контейнер приложений предоставит зависимость при размещении. System – похож на provided, однако необходимо указывать конкретный путь к зависимости в системе.

Зависимости в Maven транзитивны, то есть если включенный подпроект зависит от набора библиотек, то включающий проект тоже зависит от этого набора библиотек.

- `generate-sources/generate-resources`
- `compile`
- `test-compile`
- `test`
- `package`
- `integration-test (pre and post)`
- `install`
- `deploy`

Жизненный цикл Maven – приложения – последовательность целей и стадий сборки.

Generate sources/resources применяется, если в приложении часть исходного кода авто генерируется (например на базе DSL грамматики собирается компилятор предметно ориентированного языка). Далее код компилируется, затем компилируются тесты и производится тестирование. Далее происходит сборка в пакеты и интеграционное тестирование. Приложение установится в локальном репозитории и, если указан сервер приложений, будет размещено на этом сервере.

- Все операции над проектом выполняются плагинами.
- Плагины в качестве точек входа содержат цели, связанные с ЖЦ:
  - Core: `clean compiler deploy failsafe install resources site surefire verifier`.
  - Packaging: `ear ejb jar rar war app-client/acr shade source verifier`.
  - Reporting, Tools, and thirdparty.

Плагины используются для управления сборкой, их надо в явном виде включать в POM.

Вызов плагинов управляется общей логикой сборки проекта, плагины в качестве точек входа содержат цели, связанные с ЖЦ сборки. В число целей, например, входят цели упаковки.

В случае необходимости выполнения нестандартных действий в определенной фазе, например, на стадии генерации исходников «generate-sources», можно добавить вызов соответствующего плагина в файле *pom.xml* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>имя-плагина</artifactId>
  <executions>
    <execution>
      <id>customTask</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>pluginGoal</goal>
      </goals>
    </execution>
  </executions>
</plugin>
...
```

Goal – это конкретное выполняемое плагином действие. Плагин привязывает свои голы к фазам. Например, когда мы вызываем `mvn clean`, работу по удалению файлов сборки делает не сама фаза `clean`, а привязанная к ней цель `clean:clean` из встроенного `maven-clean-plugin`.

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

Maven является по сути ограниченно-декларативной системой, то есть, если по какой то причине нужно собрать что либо, не описываемое плагином, начнутся сложности.

Gradle – успешный последователь Ant и Maven. Основные его достоинства:

- Нейтральность к языкам программирования, Gradle никак не ограничивает вас в языках программирования. Можно указать, как собрать приложение, для различных языков.
- Описание зависимостей подобно Maven и возможность использования репозитория зависимостей.
- Подобно Maven использует плагины и жизненный цикл
- Использует лаконичный проблемно-специфичный язык для описания сборки, использует возможности Groovy, также возможно использование аналога на Kotlin
- Инкрементальная и параллельная сборка – позволяет запускать плагины только при наличии действительной необходимости выполнения сборки, экономит время.

Управляется файлом `build.gradle`

```
apply plugin: "java"
apply plugin: "application"

mainClassName = "ru.ifmo.cs.Bcomp-NG"

repositories {
    mavenCentral()
}

dependencies {
    compile "log4j:log4j-core:2.12.1"
}

jar {
    manifest.attributes("Main-Class": mainClassName);
}
```

Задание конфигурации простое.

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

Утилиты, как GNU autotools, выполняют сходный набор действий менеджеру зависимостей, но, в основном, не скачивают удаленные зависимости, а автоматизируют сборку продукта, делая ее более декларативной.

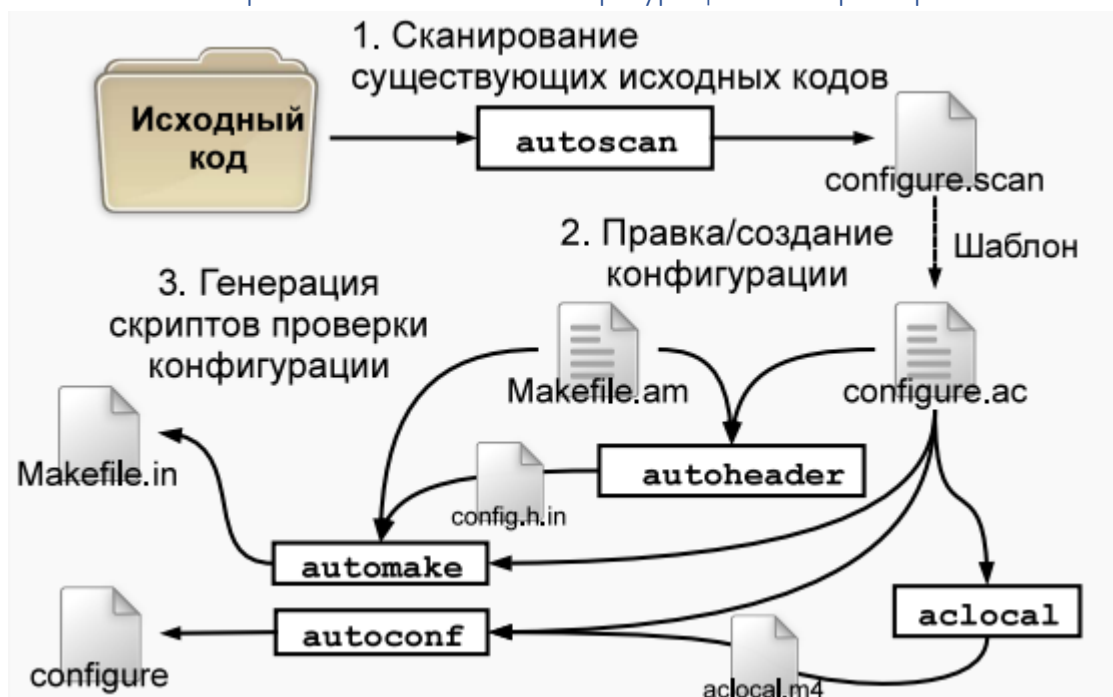
Системы используются для генерации корректного Makefile. На процесс генерации оказывают влияние режимы работы программы, заданная конфигурация, операционная система и фактическое наличие библиотек.

GNU autotools основаны на макропроцессоре общего назначения m4 – программе, преобразующей входной текст в выходной, руководствуясь правилами замены последовательностей символов, называемых правилами макроподстановки.

Исторически используется для распространения программ с открытым кодом, для сборки из исходных материалов пользователю нужно знать всего три команды: ./configure, make и sudo make install.

Полностью платформ независимы.

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



С помощью утилиты autoscan сканируется существующий исходный код и выделяются участки кода, которые могут зависеть от особенностей платформы и операционной системы. По результатам сканирования формируется шаблон конфигурационного файла configure.scan, который затем редактируется вручную и получается файл configure.ac.

Также вручную создается файл Makefile.am. В нем в явном виде указываются названия исполняемых программ, из каких исходных файлов они должны быть собраны, и другие зависимости. Такой файл создается в каждом из подкаталогов исходных файлов.

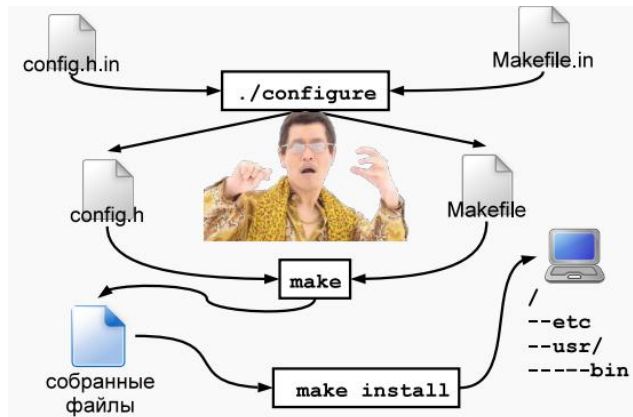
После этого вызывается autoheader, создается шаблон config.h.in для файла config.h. Команда aclocal проверяет, что установлено на локальной системе разработчика и должно быть использовано в проекте.

Затем запускаются automake и autoconf, в результате чего в дистрибутиве появляются файлы Makefile.in и configure, которые позже используются при определении текущей конфигурации на целевой системе, где будет собираться ПО.

В файле configure.ac сначала описывается продукт (имя, версия, путь к исходникам, заголовкам). Так autoscan подготавливает шаблон для продукта. Затем проверяется наличие компиляторов и необходимых системных библиотек, и отдельных функций в этих библиотеках.

На этапе конфигурации будут создаваться небольшие тестовые программы и проверять наличие системных функций. Если они есть, то в config.h будет указано их наличие. В тексте программы можно указать реакцию на наличие или отсутствие тех или иных функций в соответствии с config.h.

Makefile.am содержит описание исходников, название модуля и ключи компиляции для каждого каталога сборки.



После скачивания ПО запускается ./configure и создаются заголовочный файл и Makefile. После чего используется make и все собранные файлы распределяются в необходимые системные каталоги с помощью make install.

## 51. Сервера сборки/непрерывной интеграции.

Для сборки продукта в автоматическом режиме существуют сервера сборки или сервера непрерывной интеграции. Их основное назначение – сборка новой версии продукта при наступлении заданных администратором или разработчиком условий. Такими условиями могут быть, например изменения исходного кода в ветке Мастер, наступление определенного времени и т. п.

Билд сервер может также проводить тесты, снимать метрики с программного кода, производить автозапуск.

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

Примерами таких систем для Java являются Jenkins и Travis CI.

## 52. Основные понятия тестирования. Цели тестирования.

Основные понятия тестирования:

- Mistake – ошибка разработчика. Действие человека, которое привело к получению неверного результата.
- Fault – дефект, изъян. Неверный шаг в алгоритме (неверное определение данных) в компьютерной программе. Следствие ошибки, потенциальная причина неисправности.
- Failure – неисправность, отказ, сбой – наблюдаемое проявление дефекта, в том числе крах, падение программы.
- Error – невозможность выполнить с использованием программы задачу, получить верный результат.
- Bug – неформальный термин.

Цели тестирования по ISTQB:

- Обнаружение дефектов

- Повышение уверенности в уровне качества
- Предоставление информации для принятия решений
- Предотвращение дефектов

Цель тестирования – увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах.

Корректное поведение – программа должна соответствовать предъявленным к ней требованиям, особенностям рынка, стандартам в бизнес среде и т. п.

Пользовательское доверие зарабатывается при помощи демонстрации измеряемых показателей, существуют метрики, позволяющие измерить уровень дефектов в коде.

Необходимые обстоятельства – требование реального окружения.

Валидация – мероприятие по проверке корректности требования к программному обеспечению – правильную ли вещь сделали. Верификация – проверка соответствия ПО требованиям – правильно ли сделали вещь.

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

Тестовое покрытие включает в себя то, насколько код приложения покрыт тестами, которые могут находить известные и потенциальные дефекты. Полное тестовое покрытие – весь код и все варианты развития покрыты тестами. Если оно и достижимо, то ценой больших затрат. Например, для тестирования функции умножения двух чисел существует  $2^{64}$  вариантов входных данных. Тогда, на 3 ГГц ЦПУ, полное тестовое покрытие будет проверяться около 181,5 лет.

Самый простой способ организации тестирования – по таблице входных данных и эталонного результата.

### 54. Статическое и динамическое тестирование.

Статическое тестирование – тестирование, не связанное с запуском набора тестов для ПО. Это методики по рецензированию и инспекциям кода, проверки спецификаций, архитектурных принципов, требований и т. д. Оно способствует раннему нахождению дефектов, что экономит время и средства. Стоимость исправления дефекта по мере перехода от одной фазы к другой растет с фактором 10.

Динамическое тестирование требует уже созданной программной архитектуры. Осуществляется сборка и запуск модулей, групп модулей или системы.

Подход Test Driven Development – тесты разрабатываются перед кодом. По мере написания кода тесты начинают проходить, и разработка завершается, когда все тесты пройдены успешно.

### 55. Автоматизация тестов и ручное тестирование.

Автоматизация тестов – положительное явление, однако часто ручное тестирование дешевле и проще, особенно при простых задачах. Легче нанять низкоквалифицированный персонал, чем написать программу, формирующую и меняющую сценарии тестирования.

Регрессионное тестирование – проверка того, что после изменения программы, старые тесты все еще проходят. Позволяет проверить влияние изменений на программу.

Для автоматизации тестирования и регрессивного тестирования необходимо обеспечить однозначное повторение тестового сценария.

Важно проверять приложение в разных окружениях – тестирование совместимости.



Чем меньше изначально сформированных тестов, тем меньше общая сложность и длительность тестирования.

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

Первый классический подход – метод «черного ящика» - внутреннее содержимое программы скрыто. При таком тестировании работа идет на уровне спецификации или на основе опыта составителя тестов. Тесты подаются на вход исходные данные и сравнивают результат с известным эталоном. Источники данных – спецификации, требования и дизайн.

Метод «белого ящика» - можно исследовать исходный код. Можно оценить размер тестового покрытия – из приложения строится граф, где все части кода – узлы графа, ветвления и циклы – переходы. Определяется цикломатическая сложность программы – число путей обхода существующего кода программы, максимально необходимое число тестов для полной проверки программы. Тесты выбираются так, чтобы покрыть все пути полученного графа.

Дополнительный источник данных – опыт разработчика, позволяет решать стандартные ситуации типовыми методами, минимизируя вероятность ошибки.

Источником данных для тестов могут являться модели, в том числе UML диаграммы и их описание. Выбираются основные и альтернативные пути, а затем данные для их тестирования.

Роли и деятельности в тестировании:

- Проектирование тестов – высокая квалификация персонала, знание предметной области, особенностей пользовательского интерфейса, дискретной математики, программирования и тестирования
- Автоматизация тестов – обычно программисты, разработчики тестовых скриптов и программ. Модульные тесты обычно пишут сами разработчики кода, а интеграционные и системные – отдельные сотрудники
- Исполнение тестов – не требует высокой квалификации, нужно знать тестовую инфраструктуру, принцип работы с приложением и навыки работы с компьютером
- Анализ результатов – высокая квалификация персонала, знания предметной области и технические знания.

#### 57. Понятие тестового случая и сценария.

Тестовый случай – набор входных значений, пред и пост условий выполнения, ожидаемых результатов.

Входные значения – набор данных, на которых разрабатываемое ПО должно вести себя определенным образом с точки зрения спецификации требований к ПО или других источников информации о поведении программы.

Ожидаемые результаты должны создаваться как часть спецификаций тестовых сценариев и включать в себя выходные данные, изменения в данных и состояниях, другие последствия теста.

В идеальных условиях ожидаемые результаты должны быть определены до выполнения теста. Эта концепция отражается в TDD подходе к разработке.

Тестовый случай должен быть повторяемым, одинаковый набор входных значений и состояний = одинаковые выходные данные и состояния.

Желательно автоматизировать проведение теста, особенно важно для регрессионного тестирования.

Должны учитываться состояния внутри ПО и переходы между ними.

Тестовый сценарий есть последовательность тестовых случаев. В них должно быть предусмотрено появление как положительной, так и отрицательной реакции системы на действия пользователей. Для неправильных действий тоже создаются тестовые сценарии.

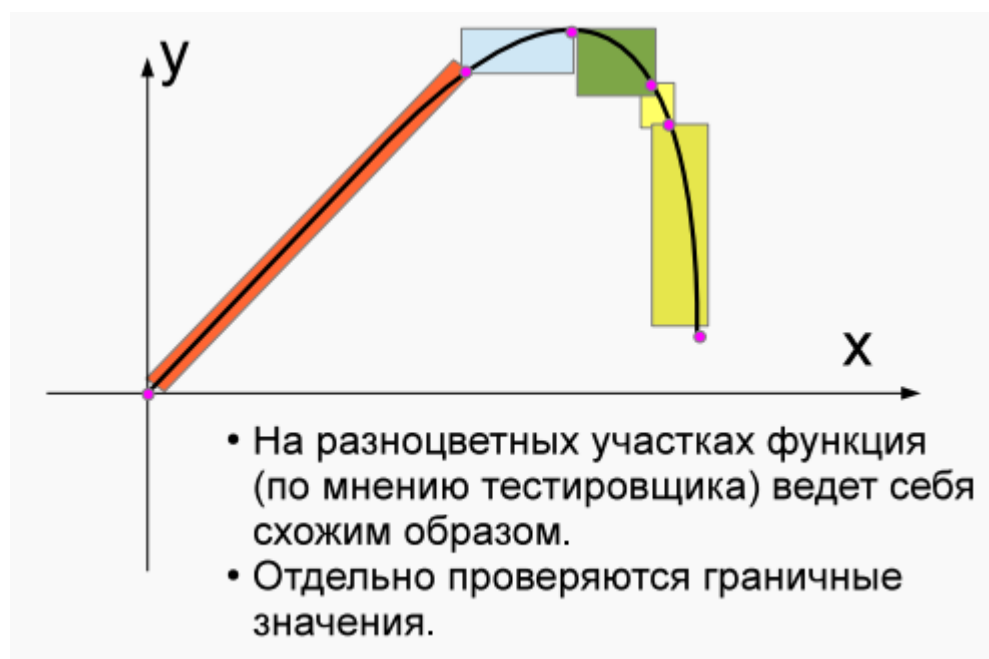
#### 58. Выбор тестового покрытия и количества тестов. Анализ эквивалентности.

Следует соблюдать баланс между качеством и скоростью вывода продукта в эксплуатацию. Если тестов много, то растет покрытие и соответственно качество. Однако долго тестируемая программа может стать никому не нужной т. к. конкуренты уже захватили рынок.

Полное тестовое покрытие недостижимо.

Методы выбора тестового покрытия:

- Метод эквивалентного разбиения – анализ граничных значений, внутри которых тестируемая функция ведет себя одинаково
- Таблица альтернативных решений – составляется таблица комбинаций входных данных с соответствующими выходными данными, которая может быть использована для проектирования сценариев.
- Таблица переходов – выделяются явные состояния внутри системы, определяются переходы между ними, покрываются тестами.
- Сценарии использования могут служить источником информации для формирования покрытия. Добавляются конкретные значения, вводимые пользователем. Учитываются и основные и альтернативные сценарии.



При анализе эквивалентности тестируемая функция или модуль разбивается на участки, где программа ведет себя одинаково (эквивалентно). Внутри каждого участка свой набор тестовых случаев. Отдельные тесты составляются для граничных значений участков.

#### 59. Модульное тестирование. Junit 4.

Модуль – компонент, который необходимо протестировать отдельно от остального программного продукта. Модуль выполняет некую законченную функцию. Модуль может быть методом, классом, или совокупностью методов и классов, формирующих программный модуль. Модули определены в дизайне программы.

Для проведения модульного тестирования модуль необходимо изолировать из системы.

Изолирование производится ради исключения сторонних воздействий. Изолирование предполагает замену модулей, осуществляющих вызов драйверами (управляющими тестированием), а подчиненных модулей – заглушками. Количество таких сборок обычно равно количеству модулей в системе.

Драйвер – компонент, вызывающий модули и обеспечивающий последовательность тестирования. Должен последовательно вызывать тестируемый модуль с разными входными параметрами и условиями.

Заглушка – сильно упрощенная реализация подчиненного модуля, ведущая себя ему подобно. При вызове возвращает заранее заданные значения, не предусматривает ввод входных значений, не предусмотренных в заглушке.

Часто для создания заглушек используется табличный метод из-за наглядности.

В Java для модульного тестирования используют Junit – простейший фреймворк для модульного тестирования. Построен на аннотациях. Метод, организующий тестирование, помечается аннотацией @Test. При этом фреймворк последовательно просматривает загруженные классы (reflection API) и ищет в них аннотацию. Все методы с ней запускаются (возможно в разных потоках).

Внутри тестового метода проверяется тестовое покрытие на соответствие определенным условиям с помощью методов проверки – методов assertion. В Junit 4 множество таких функций, способных проверять равенство, совпадение, появление исключения и т. д. Результаты тестов журналируются.

Также есть аннотации Before, After, BeforeClass, AfterClass, содержимое, которое выполняется перед каждым тестом, после каждого теста, один раз перед выполнением всех тестов загружаемого тестового класса и один раз после выполнения всех тестов выгружаемого тестового класса.

Последовательность тестов не регламентирована (можно определить искусственно, но по умолчанию тесты выполняются параллельно и независимо). Junit определяет порядок сам.

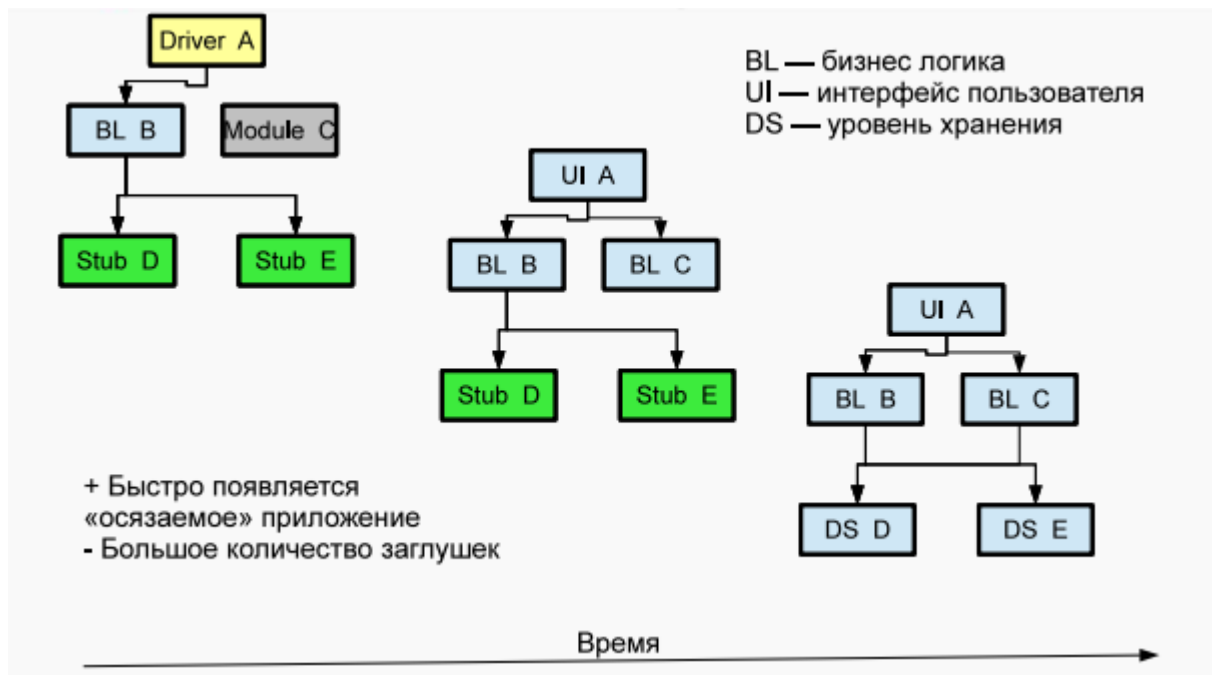
## 60. Интеграционное тестирование. Стратегии интеграции.

По средствам и подходам похоже на модульное, но интеграционное тестирование проверяет взаимодействие между программными модулями и выполняется после модульного. Смысл – проверка интерфейсов взаимодействия модулей на правильную последовательность вызовов и соответствия протоколов такого взаимодействия требованиям спецификации.

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

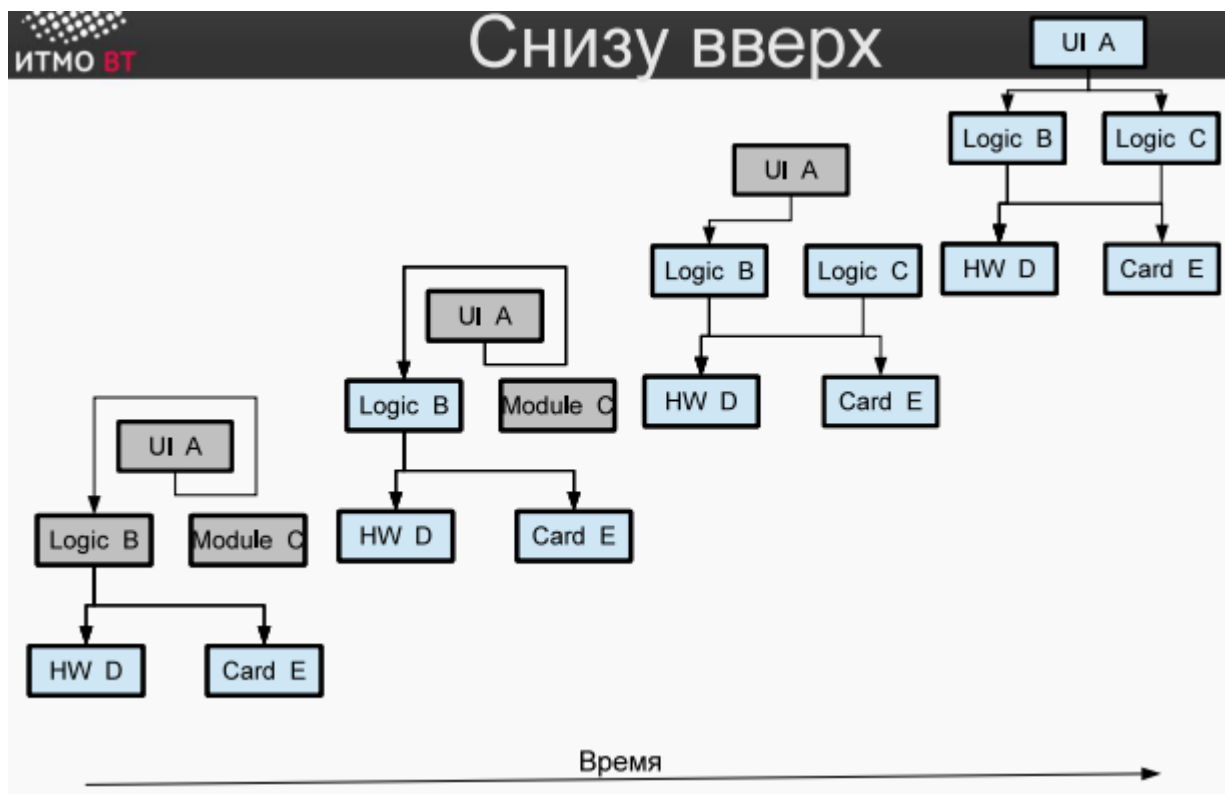
Интеграционное тестирование может проводиться, когда два компонента уже разработаны. Остальные компоненты добавляются по мере их реализации, при этом важна последовательность интеграции модулей, определяемая стратегией интеграции.

При составлении плана разработки ПО стратегию нужно учитывать и подготовить в соответствии с ней последовательность разработки модулей.



Стратегия «Сверху вниз» - самая распространенная, применяется в основном для бизнес-приложений. Сначала проверяется бизнес логика с использованием драйвера и заглушек. После этого подключается пользовательский интерфейс, а потом блоки хранения данных.

Стратегия позволяет быстро продемонстрировать приложение конечному заказчику, однако ведет к разработке большого числа заглушек.



Стратегия «Снизу вверх» чаще используется, когда приложение сильно связано с аппаратурой. Вследствие этого интеграция начинается с блоков нижнего уровня, взаимодействующих с аппаратурой.

Цикл разработки аппаратуры обычно более длительный и трудоемкий. Тестирование остальных частей приложения, использующих отдельное аппаратное обеспечение, может начаться только после того, как будут готовы первые прототипы. Компании, разрабатывающие аппаратуру, обычно подготавливают тестовые версии своих продуктов и предоставляют их смежным компаниям еще до их широкого запуска в производство.

Другие стратегии:

- Функциональная – по одной функции (1 сценарий UI – Логика - БД, затем другой и так далее)
- Ядро – сначала формируется минимальный работоспособный функционал, а потом возможности приложения расширяются
- Большой взрыв, больше добавить нечего

## 61. Функциональное тестирование. Selenium.

Обычно рассматривается как разновидность интеграционного. Проверяется функционал, заложенный в программу. Осуществляется на базе сценариев использования, где в явном виде описаны действия пользователя в системе. Обычно проверяются бизнес-процессы целиком, они могут включать в себя использование различных ролей пользователей.

Основной элемент управления – графический интерфейс пользователя. Функциональные тесты могут быть как ручными, так и автоматизированными. Разработка тестируемого функционала должна быть полностью завершена на всех уровнях приложения.

Полностью избежать ручного тестирования не получится, особенно явно это при изменении функциональности приложения с уже готовыми тестами.

Есть большое число средств автоматизации, которые берут на себя управление интерфейсом пользователя. Например Selenium – дополнение к браузеру Firefox – позволяет сначала записать тестовую последовательность использования интерфейса, а затем экспортировать ее в виде тестовой программы и для других браузеров.

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

Статическое тестирования – вид тестирования ПО, проводящийся перед динамическим тестированием и не подразумевающий исполнение кода. Цели типов тестирования одинаковы – обнаружение дефектов. Статическое тестирование находит причины сбоя, а не сами отказы.

Рецензирование может проводиться вручную или с помощью специальных средств. Главная составляющая ручного процесса – исследование и комментирование продукта.

Одной из техник является неформальное ревью коллегой, который сможет посмотреть на ваш код своими глазами и заметить какие-то очевидные дефекты. Однако не точный метод, т. к. коллега тоже человек и может не видеть ошибок или быть субъективным.

Существуют несколько формальных техник, проводимых в виде отдельно организованного собрания под управлением различных ролей.

- Технический анализ, метод повторного просмотра – управляет технический лидер проекта
- Management review – управляет менеджер разработки
- Метод сквозного контроля – просмотр решений специально выделенным экспертом, он ведет аудиторию через рассматриваемый артефакт, фиксируя недочеты и дефекты, и контролирует процесс их исправления.
- Инспекции – человек может контролировать не более двух параметров одновременно => каждый инспектор обладает двумя ролями и контролирует только их.

Рецензирование может проводиться для любого продукта, связанного с разработкой ПО: код, спецификации, дизайн, планы тестирования, руководства пользователя и т. п.

Исправление дефектов на раннем этапе часто обходится значительно дешевле заказчику. Во время рецензирования могут быть выявлены дефекты, которые трудно найти динамическими методами.

Итого, преимущества рецензирования:

- Раннее обнаружение и исправление дефектов
- Улучшение продуктивности
- Уменьшение времени разработки
- Уменьшение времени и стоимости тестирования
- Сокращение стоимости ЖЦ
- Уменьшение числа дефектов
- Улучшение коммуникации в команде
- Передача информации в команде в образовательных целях

Типичные дефекты, которые проще найти при рецензировании: отклонения от стандартов, дефекты требований и дизайна, некорректные спецификации, недостаточная пригодность к сопровождению.

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

Примерами статического анализа кода являются Lint (Си) и FindBugs (Java). Они строят синтаксическое дерево и проводят строгую проверку на неопределенное поведение, нарушение алгоритмов использования библиотеки, сценарии некорректного поведения, разрушение кроссплатформенности и т. п.

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

Тестирование системы в целом начинается после окончания интеграции. Необходимо проверить заявленные характеристики системы. Состоит из нескольких частей:

- Системное тестирование – внутри организации разработчика без сторонних лиц
- Альфа и бета тест – пользователями на оборудовании разработчиков и в реальном окружении под наблюдением.
- Приемочное тестирование – пользователем в его собственном окружении без контроля разработчика.

Системное тестирование производится от простых сценариев к сложным.

Сначала – заявленные возможности ПО. Обычно те же сценарии, что и в функциональном тестировании, но на корректность проверяется вся система целиком. Тестовые случаи, приводящие к ошибкам или перегрузке – не используются.

Далее проверяется стабильность работы, например при одновременном обращении нескольких пользователей.

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

Затем – аспекты системы, связанные с совместимостью – кросс-браузерность, возможность использования с программой стороннего ПО.

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

Тестирование производительности включает в себя виды тестов CARAT:

- Capacity – нефункциональные возможности. Характеристики, связанные с объемом обрабатываемой информации. Заключается в последовательном доведении до предела каждого из имеющихся параметров и наблюдении за поведением системы.
- Accuracy – точность. В основном точность математических расчетов с заданной погрешностью. Может быть критична для систем моделирования физ. Процессов. В системах реального времени ПО должно обеспечить заданную точность в ограниченный промежуток времени.
- Response Time – время ответа системы на запрос пользователя.
- Availability – готовность, обычно выражается в коэффициенте готовности – отношении разности среднего времени до отказа и среднего времени до восстановления к среднему времени до отказа  $(MTBF - MTTR)/MTBF$ .
- Throughput – пропускная способность – сколько запросов обрабатывает система за единицу времени.

Существуют средства для нагрузочного тестирования. Нагрузка может быть стационарной, сформированной заданным количеством запросов в единицу времени, а также более сложной – изменяющейся по заданному закону. Все средства обеспечивают замер времени ответа и журналирование.

Для ПО на Java наиболее популярен Apache JMeter, позволяющий удаленно формировать нагрузку на тестируемое ПО с использованием большого числа протоколов, а также организовывать распределенную нагрузку.



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

Тестирование системы в целом начинается после окончания интеграции. Необходимо проверить заявленные характеристики системы. Состоит из нескольких частей:

- Системное тестирование – внутри организации разработчика без сторонних лиц
- Альфа и бета тест – пользователями на оборудовании разработчиков и в реальном окружении под наблюдением.
- Приемочное тестирование – пользователем в его собственном окружении без контроля разработчика.

Альфа и Бета тестирование – выполняется пользователем под контролем разработчика. За счет этого разработчики получают полезные для завершения разработки отзывы. Кроме того, так как тестирование выполняют не разработчики, то тестирующие могут пойти нестандартными путями, которые разработчики не учли. При этом альфа тестирование производится на окружении разработчиков, а бета тестирования – в реальном пользовательском окружении.

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

В первую очередь – системный и архитектурный аспекты. При размещении программы в вычислительной среде, там уже работают другие программы. Вычислительная среда построена согласно определенной архитектуре, которая задает структурные особенности выполняемых в ней программ, их параметры и характеристики. Архитектура может быть:

- Распределенной, где нужен баланс производительности на всех уровнях обработки информации
- Кластерной, где нужен баланс между различными узлами кластера
- Виртуализированной, где на одной серверной ферме работает много различных подсистем, взаимно влияя на друг друга из-за общих ограничений самой фермы

Низкоуровневый аппаратный аспект связан с техническими характеристиками аппаратуры. Например, от тактовой частоты процессора зависит линейная скорость выполнения каждого потока программы, от размера кэшей и уровней конкуренции разных потоков зависит то, как много данных могут быть помещены в кэш второго уровня и как долго они могут там оставаться, что также влияет на скорость обработки данных.

Программный аспект связан с выбором подходящих алгоритмов для разрабатываемых программ. Зачастую сроки сдачи в эксплуатацию важнее оптимальности решения задач. Например, выбор между алгоритмами сортировки. Также важными могут быть пул, реализация многопоточности, блокировки и т. п.

Важный фактор – человеческий – ошибки и неправильные архитектурные решения, узкие знания в области и т. д.

Любая работа по анализу производительности – научный эксперимент, в котором выбираются критерии оценки, по которым можно оценить производительность программы. После этого выбираются средства измерения. Не интрузивные – не влияют на результаты, слабо интрузивные вносят небольшие искажения, интрузивные – напрямую влияют на результаты.

Практически любое измерение производительности влияет на результат измерения, даже счетчики операционной системы тратят процессорное время.

После выбора средства и критериев нужно выбрать нагрузку – она должна быть максимально приближенной, эквивалентной реальной пользовательской нагрузке. Во время работы под нагрузкой происходит сбор результатов – системные утилиты производят измерения счетчиков ОС. Периодичность сбора и объем получаемых журналов также могут влиять на результаты.

Затем проводится анализ результатов и исключение данных, наведенных средствами измерения. После анализа выбирается один из параметров программы или участков кода и переписывается. Эксперимент повторяется и так с разными параметрами до получения нужного результата.

#### 66. Ключевые характеристики производительности.

Все параметры производительности тесно связаны между собой.

- Время отклика системы – время от выдачи запроса до получения первых данных. Полное время обслуживания – до получения всех ответных данных.
- Пропускная способность – максимальное число запросов за единицу времени способных пройти через канал, систему или ее узел.
- Утилизация ресурса показывает какую долю времени ресурс занят полезной работой. Ожидание ресурса учитывает среднюю длину очереди, % ожидания показывает сколько времени очередь не пуста.
- Точка насыщения – момент, когда нагрузка достигает предельного значения, которое может обработать устройство или программа. Достижение максимальной производительности. После этого производительность будет резко падать, обычно экспоненциально. Если снижение более плавное, то компоненты системы хорошо сбалансированы. Масштабируемость характеризует то, насколько можно количественно расширить и нагрузить систему.
- Эффективность ПО – соотношение полезной работы системы к общему количеству работы. Например, свопинг загружает систему, но полезной работы не происходит. Ускорение работы и прирост производительности показывают, насколько больше полезной работы выполняет программа после внесения изменений.
- В Java - Memory Footprints – стратегии использования памяти. Если работа с памятью некорректно ведется, скорость работы резко снижается.

#### 67. Нисходящий метод поиска узких мест.

1. Администратор системы ищет ошибки аппаратуры и конфигурации системы, проверяются системные журналы, файлы конфигурации системы и ПО. Сбойный блок на диске или дефект кабеля может привести к повторному чтению участка памяти. В системных журналах ошибки видны сразу. Возможно было выделено слишком мало памяти для виртуальной машины, в результате чего слишком частые сборки мусора останавливают систему.
2. Далее наблюдение за системой. ОС обладает развитыми средствами наблюдения за подсистемами, средства системного, межузлового мониторинга и мониторинга виртуальных машин помогают наблюдать картину приложения и определить, куда тратится время.
3. Далее наблюдение за приложением – фиксируются алгоритмические проблемы, проблемы API, проблемы многопоточности, блокировок и синхронизации. Используются средства мониторинга и профилирования приложений.
4. Если результат не достигнут – мониторинг микроархитектуры: проверка количества инструкций за один такт, выравнивание данных, оптимизация кэшей, предсказание переходов и т. д. Здесь нужно знать ассемблерный код, архитектуру процессора, принципы компиляции и т. п. Обычно в пользовательском ПО до сюда не доходят из-за сложности внесения изменений.

68. Пирамида памяти и ее влияние на производительность.



Сверху пирамиды – самая дорогая память с наименьшим объемом. Снизу – самая дешевая с максимальным объемом. Разница между обращением к основной памяти и обращению к регистру процессора колоссальная. Промах мимо кэша в 2% снизит производительность на 40% от эталонной.

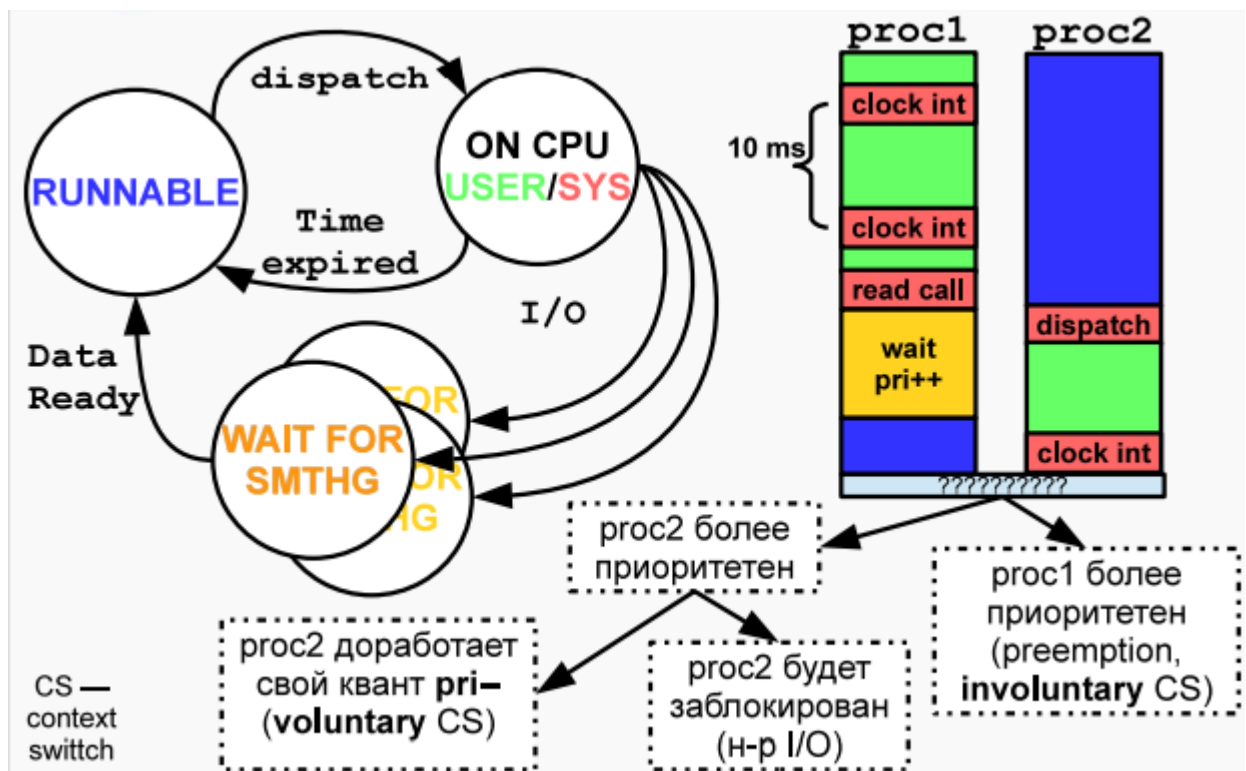
Учет архитектуры ЭВМ позволяет сделать разрабатываемые программы существенно, на несколько порядков, быстрее.

## 69. Мониторинг производительности: процессы.

Параметры системного мониторинга:

- CPU: user%, system%, idle%, ctxt\_sw, interrupts, load average.
- IO: b, kb, mb, blk read & write/s, op/s, wait time.
- VM: free, buff, cache, pagein/s, pageout/s, scan time.
- Network : up (TX) & down (RX) streams, errors, collisons.
- Детальные данные других подсистем.

Диспетчеризация:



Здесь видна упрощенная схема диспетчеризации процессов CPU.

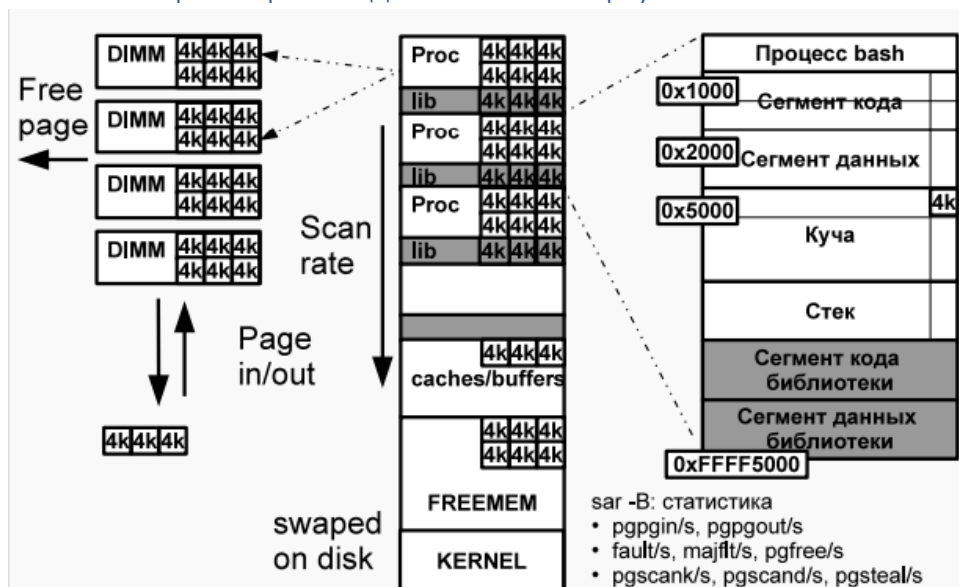
Процесс или поток внутри процесса может находиться в трех состояниях: готов к исполнению, находится на исполняющем устройстве на уровне ОС или уровне пользователя, или ожидать ввода/вывода, освобождения блокировки и т. п.

Пусть proc1 находится на процессоре и работает программа пользователя. Раз в 10 мс происходит прерывание от часов для сбора данных счетчиками производительности, во время прерывания ОС наращивает счетчики и проверяет в очереди на исполнение наличие процесса с более высоким приоритетом, чем у текущего, а также проверяет не исчерпал ли текущий процесс свой квант времени. Пусть не исчерпал, тогда proc1 продолжит выполнение и proc2 останется в состоянии Runnable. Так будет пока не будет исчерпан квант времени, либо не будет сделан вызов из Proc1 к устройству ввода-вывода.

При появлении такого вызова `proc1` перейдет в режим ожидания и произойдет context switch – процессор переходит к исполнению другого процесса, сохраняя окружение предыдущего. Внутри ядра произойдет диспетчеризация, в ходе которой будет выбран процесс с высшим приоритетом, готовый к выполнению. Второй процесс начинает выполняться. Процесс 1 отправляется в ожидание и у него повышается приоритет – чем больше время ожидания, тем больше увеличивается приоритет и наоборот.

В момент прерывания несколько вариантов действия – если приоритет второго процесса выше, то он либо дорабатывает свой квант времени, либо выполняет операцию ввода-вывода. Если 1 процесс более приоритетен, то произойдет involuntary context switch и он снова перейдет на процессор.

## 70. Мониторинг производительности: виртуальная память.



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

В работе виртуальной памяти задействованы физическая память, устройство подкачки (своп устройство) и виртуальные страницы, которые могут принадлежать физической или виртуальной памяти. Сегментно-страничная структура процессов типична для современных ОС, где каждая страница может находиться в нескольких состояниях: быть в памяти, быть на своп устройстве, быть зарезервированной. Вся память процесса поделена на страницы от 4 Кб. Размер влияет на количество мэпингов (отображений структур виртуальных адресов ОС на страницы внутри банка памяти). Используемые для чтения страницы необязательно перемещать на своп устройство, их можно удалить из памяти т. к. они могут быть загружены из исполняемого файла программы. Страницы, связанные с файлами – именованная память. Динамически созданные в куче – анонимная память.

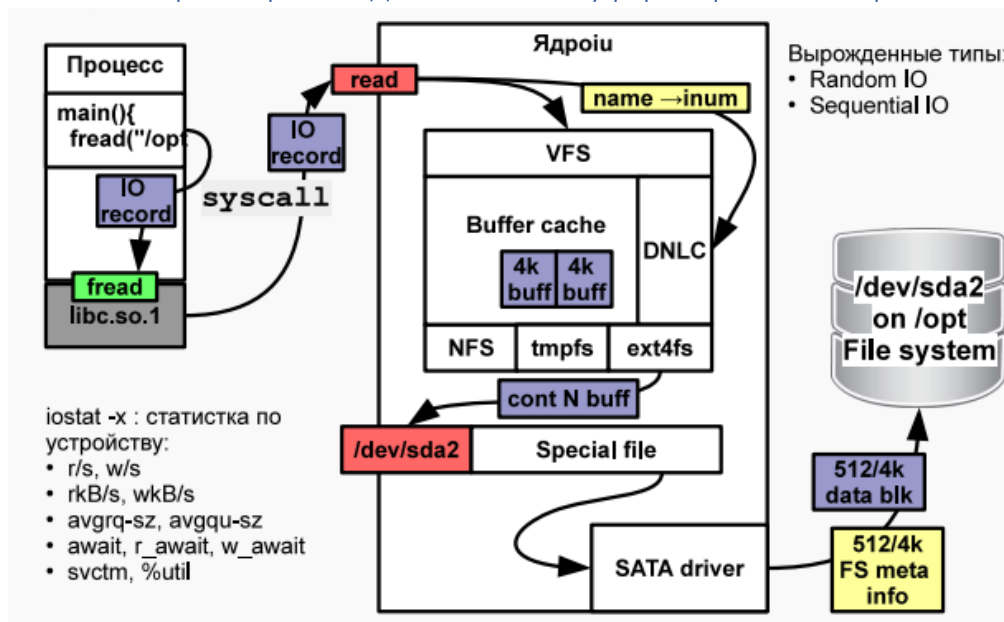
Параметры виртуальной памяти:

- Scan rate – число отсканированных страниц за единицу времени, если значение высоко, то есть недостаток оперативной памяти и ОС постоянно манипулирует страницами. При недостатке памяти ОС последовательно сканирует редко используемые, неизменные страницы и принимает решения об их удалении или записи на диск.

- Если процессор обратился к отсутствующей в памяти странице – страничный сбой. Минорный – если нужно создать маппинг в таблицах адресации, мажорный – если страница должна быть загружена из диска подкачки.

Сбор статистики виртуальной памяти в ОС \*nix осуществляется при помощи `sar -B`.

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



Пусть внутри программы производится обращение к `fread`, которая находится в системной библиотеке. Функция формирует запрос к ядру и вызывает соответствующую функцию ядра `read`. Ввод-вывод в ядре попадает в подсистему Виртуальной файловой системы (VFS) и запрос попадает на уровень, не связанный с конкретной файловой системой. Имя файла преобразуется в Directory Name Lookup Cache в номер inode.

ОС экономит ресурсы, и операция `read` сначала будет работать с буферным кэшем. В нем содержатся в виде блоков данных файлы в ОЗУ, формируя промежуточное хранилище. Каждые 30 секунд данные, которые были помечены как измененные, сохраняются на диск.

Под VFS существуют реализации модуля ядра физических файловых систем. К точке монтирования подключается устройство с использованием специального файла, после чего работу ведет драйвер и данные перемещаются в дисковое устройство.

Для каждой файловой системы наблюдаемые параметры – количество чтений, записей, объем читаемых и записываемых данных, средние времена запросов, время ожидания, время обслуживания и процент занятости устройства.

К вырожденным типам обмена с дисковой подсистемой различают случайный доступ – каждый раз к новому месту – и последовательный доступ – данные записываются и читаются большими группами последовательно. Скорость чтения и записи во втором случае гораздо выше, чем в первом. Для случайного доступа повысить скорость может использование SSD, т. к. в них нет движущихся частей.

Команда `iostat` рассматривает упомянутые параметры.

## 72. Мониторинг производительности: Windows и Linux.

В Windows самым распространенным средством является Task Manager, где выводится статистика использования процессора, памяти, дисковой подсистемы и т. д.



Для более детального исследования есть системные оснастки – Resource Monitor, Performance Monitor, Reliability Monitor и другие.

Наиболее подробное средство – Microsoft SysInternals, базируется на внутрисистемных счетчиках.

В Linux существует большое число средств мониторинга. В общем случае они неинтрузивны.

Для просмотра счетчиков производительности стандартных подсистем – vmstat, mpstat, iostat, netstat.

Утилита top – динамическое наблюдение за характеристиками запущенных процессов, такие как приоритет, занимаемая память.

Утилита sar имеет множество опций для вывода разной информации, но некоторые ее подходы к сбору информации устарели.

Perf может собирать и показывать большое число характеристик ядра и запущенных под управлением утилиты процессов. Умеет работать со счетчиками производительности системы для сбора таких событий, как промахи мимо кэша.

Для детального наблюдения за процессом – strace – проводит трассировку системных вызовов к ядру и библиотекам. Интрузивен.

Stap позволяет установить точки сбора информации в ядре, собрать и агрегировать информацию о подсистемах ядра. Средство использует свой язык программирования.

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

- Uptime – показывает среднюю загрузку процессорной подсистемы – количество готовых к выполнению процессов в очереди на диспетчеризацию за разные промежутки времени. Если таковых много, то система не справляется с нагрузкой.
- Dmesg | tail – последние ошибки.
- Vmstat 1 – есть ли свободная память, paging
- Mpstat -P ALL 1 – распределение процессов по CPU (например один процесс занят, остальные простаивают)
- Pidstat 1 – статистика по процессам, горячие процессы
- Iostat -xz 1 – характеристики ввода-вывода
- Free -m – проверка исчерпания кэшей и буферов
- Sar -n DEV 1 – сетевая статистика по интерфейсам
- Sar -n TCP,ETCP 1 – сетевая статистика по соединениям
- Top – онлайн мониторинг параметров, швейцарский нож.

### 74. Создание тестовой нагрузки и нагрузчики.

Из-за страха внесения искажений в нормальную работу системы, наблюдение реальных систем запрещено и необходимо создать тестовую систему, для которой нужно создать нагрузку, близкую по характеристикам к реальной пользовательской. В тестовой системе также можно использовать интрузивные средства мониторинга.

Нагрузку близкую к реальной можно создать с помощью средств создания синтетической нагрузки или средства записи реальной нагрузки. При этом синтетическая нагрузка всегда будет отлична от реальной, и в средствах создания такой нагрузки есть большое число параметров для гибкой настройки.



## 75. Профилирование приложений. Основные подходы.

Для исследования приложений и поиска алгоритмических дефектов существуют профилировщики приложений. С их помощью можно узнать время исполнения функции/метода, объем созданных объектов в памяти, проследить за потоками приложений и борьбой потоков за захват блокировки, временем ожидания ими освобождения блокировок и т. п.

Два основных подхода: первый – диагностические точки можно внедрять в сами функции из указанного набора. Является интрузивным методом, стоит сначала определить проблемные места.

Второй подход – использование прерываний с заданной периодичностью для сбора интересующей информации. Для определения времени исполнения функций используется состояние стека программы в момент прерывания. Собирается информация о куче.

Интервал выбирается так, чтобы не сильно снижать производительность и включить в данные даже методы с малой продолжительностью работы.

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

Борьба за производительность сопряжена компромиссами. Изменения в одном месте ведут к изменениям в другом. Чем быстрее мы хотим получить доступ к данным, тем больше нам потребуется на это памяти. Последовательный поиск в массиве всегда противоречит в плане скорости/занимаемой памяти с индексированием, т. к. для индекса нужна память, но скорость поиска увеличивается.

Так, время, потраченное CPU, связано с тем количеством памяти, которое требует программа: если сделать требования скромнее, она будет требовать больших ресурсов CPU, а программа с большими требованиями к памяти будет работать быстрее за счет алгоритмов.

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

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

1. Высокая, непродуманная нагрузка на подсистему ввода-вывода. Нужно реже читать/писать, стараться сжимать данные при помощи алгоритмов сжатия, использовать бинарное представление данных числа вместо строкового. Помогают также буферизация и замена устройств на более быстрые. Можно синхронизировать размер блока передачи с ОС или устройствами хранения.
2. Недостатки в работе планировщика: слишком частая диспетчеризация, избыточное переключение контекстов. Нужно проверить не слишком ли много используется потоков и чем занимается ОС.
3. Избыточная подкачка страниц – выдать больше памяти системе за счет процессов. Также можно запретить выгрузку некоторых критически важных процессов из памяти, вследствие чего они не будут свопаться.
4. Трата времени процессора в других системных функциях и процесса ядра. Тогда можно попробовать найти и исключить лишние системные процессы или настроить параметры ядра. Нужна тонкая настройка, понимание структуры ядра.

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

1. Проблемы с самими приложениями. Следует оптимизировать запросы к диску – меньше/реже читать/писать. Если предполагается активный обмен данными – обмениваться большими порциями за одну операцию. Проверить, согласован ли блок приложения с ОС и дисками: например, если при размере stripe-size 16 Кб, читать данные блоками по 8 Кб, то происходит двойное прочтение и преимущества RAID массива сходят на нет.

2. Буфера/кэши, т. е. в системе выделено мало памяти под промежуточное хранение данных. Тогда стоит расширить память или настроить использование системных кэшей.
3. Проблемы с аппаратурой. Можно купить новую, более совершенную дисковую подсистему. Для ускорения операций эффективны SSD или отдельные flash карты устанавливаемые прямо в шину вычислительной системы. Также эти устройства будут хорошо работать с ПО, осуществляющим случайный доступ к памяти.

#### 79. Рецепты повышения производительности при высоком %idle.

1. В системе мало процессов в стадии выполнения – может помочь распараллеливание алгоритмов в приложении, особенно в многопроцессорных системах. Если есть пулы потоков рабочих, то в них можно добавить дополнительные потоки для равномерной загрузки всех ядер ЦПУ. Также источником проблем могут быть внутренние блокировки в приложении, когда много потоков ждут один и тот же участок. Нужно оптимизировать блокировки, держать блокировки как можно меньше времени, использовать более легкие типы блокировок или использовать алгоритмы без блокировок.
2. Проблемы в самой ОС. Зачастую дефекты в ОС на системных блокировках. Следует проверить багтрекер ОС и сопоставить написанное там с симптомами проблемы. Также может помочь настройка параметров подсистем ядра.

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

Если процессор загружен на все 100, это не значит, что он работает эффективно. Для определения узких мест нужно воспользоваться средствами профилирования. Они помогут найти наиболее затратные функции, объекты с частым созданием или удалением в памяти и другие аномалии ПО.

1. Использование алгоритмов с меньшей алгоритмической сложностью (не забывая о компромиссах)
2. Принцип повторного использования объектов – крупные объекты и структуры не нужно удалять и создавать повторно, можно установить новые данные в старый объект. Так, например происходит в Unix для структур процессов, когда процесс умирает он входит в состояние зомби и ядро может сложить часть зомби на кладбище, и когда нужно создать новый процесс воскресить старую структуру.
3. На уровне микроархитектуры избавляться от кэш промахов и промахов мимо Translation Lookaside Buffer. Кэш промахи можно исправить работой над структурами данных – группировка данных для попадания в одну строку кэша, паддинг и т. п. Промахи мимо TLB исправляются использованием больших страниц памяти. Для исключения остывания кэшей можно закреплять потоки за процессорами (Биндинг). Кроме того, биндинг позволит обращаться к локальной памяти процессора при NUMA архитектуре, минимизируя запросы через шину к другой процессорной плате  
Для ценителей – попробовать переписывать части кода на ассемблере и использовать специальные средства аппаратуры – GPU и криптопроцессоры.