1) **Программный продукт** – программная часть и связанные с ней документация и данные.

2) **Жизненный цикл ПО** – развитие продукта, начиная с разработки концепции (развития идеи) и заканчивая прекращением применения (выводом из эксплуатации).

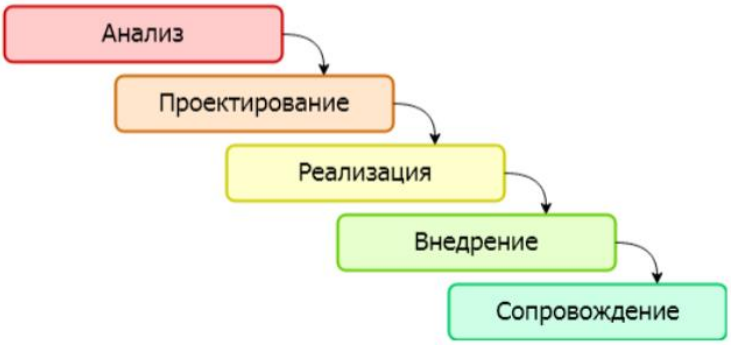
3) **Основные этапы жизненного цикла ПО**
    a. Разработка концепции, планирование
    b. Анализ
    c. Проектирование
    d. Реализация
    e. Тестирование
    f. Развертывание/внедрение
    g. Эксплуатация и сопровождение
    h. Вывод из эксплуатации

4) **Процессы жизненного цикла ПО**

    a. **Основные процессы**
        i. Анализ требований
        ii. Эскизное проектирование
        iii. Детальное проектирование
        iv. Кодирование
        v. Интеграция
        vi. Тестирование
        vii. Внедрение
        viii. Сопровождение
    b. Поддерживающие процессы
        i. Планирование
        ii. Документирование
        iii. Контроль качества
        iv. Управление процессом
        v. Создание инфраструктуры
5) Модели жизненного цикла ПО: достоинства, недостатки, применение
    a. **Модель жизненного цикла ПО** – структура, определяющая последовательность выполнения процессов, действий и задач на протяжении всего жизненного цикла.
    b. **Методология разработки ПО** – система правил и принципов по управлению разработкой ПО.

c. **Примитивная модель**

d. **Каскадная (Waterfall)**
Каждый новый этап начинается после того, как будет полностью завершен предыдущий. Это влечет за собой опасность маскировки действительных рисков до тех пор, пока не будет слишком поздно.



i. **Достоинства:**
   1. Выполняемые в четкой последовательности стадии позволяют уверенно планировать сроки выполнения работ и соответствующие ресурсы (денежные, материальные и людские).
   2. На каждой стадии формируется законченный набор документации, программного и аппаратного обеспечения, отвечающий критериям полноты и согласованности;

ii. **Недостатки:**
   1. Реальный процесс разработки информационной системы редко полностью укладывается в такую жесткую схему. Особенно это относится к разработке нетиповых и новаторских систем;
   2. Основана на точной формулировке исходных требований к информационной системе. Реально в начале проекта требования заказчика определены лишь частично;
   3. Основной недостаток – результаты разработки доступны заказчику только в конце проекта. В случае

неточного изложения требований или их изменения в течение длительного периода создания ИС заказчик получает систему, не удовлетворяющую его потребностям.

iii. **Применение**

1. Каскадная модель хорошо функционирует при ее применении в циклах разработки программного продукта, в которых используется неизменяемое определение продукта и вполне понятные технические методики.

e. **Итерационная**

Итерационная модель жизненного цикла не требует для начала полной спецификации требований. Вместо этого создание начинается с реализации части функционала, становящейся базой для определения дальнейших требований. Этот процесс повторяется. Версия может быть неидеальна, главное, чтобы она работала. Понимая конечную цель, мы стремимся к ней так, чтобы каждый шаг был результативен, а каждая версия — работоспособна.

i. **Достоинства**

1. Позволяет быстрее показать пользователям системы работоспособный продукт, тем самым, активизируя процесс уточнения и дополнения требований;
2. Допускает изменение требований при разработке информационной системы, что характерно для большинства разработок, в том числе и типовых;
3. Обеспечивает большую гибкость в управлении проектом;
4. Позволяет получить более надежную и устойчивую систему. По мере развития системы ошибки и слабые места обнаруживаются и исправляются на каждой итерации;
5. Позволяет совершенствовать процесс разработки – анализ, проводимый в каждой итерации, позволяет

проводить оценку того, что должно быть изменено в организации разработки, и улучшить ее на следующей итерации;

6. Уменьшаются риски заказчика. Заказчик может с минимальными для себя финансовыми потерями завершить развитие неперспективного проекта.

ii. Недостатки

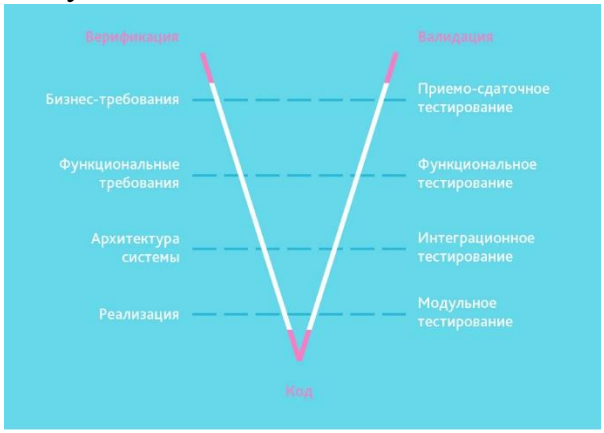
1. Увеличивается неопределенность у разработчика в перспективах развития проекта. Этот недостаток вытекает из предыдущего достоинства модели;
2. Затруднены операции временного и ресурсного планирования всего проекта в целом. Для решения этой проблемы необходимо ввести временные ограничения на каждую из стадий жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа выполнена. План составляется на основе статистических данных, полученных в предыдущих проектах и личного опыта разработчиков.

iii. Применение

1. Основная задача должна быть определена, но детали реализации могут эволюционировать с течением времени.

f. **V-model**

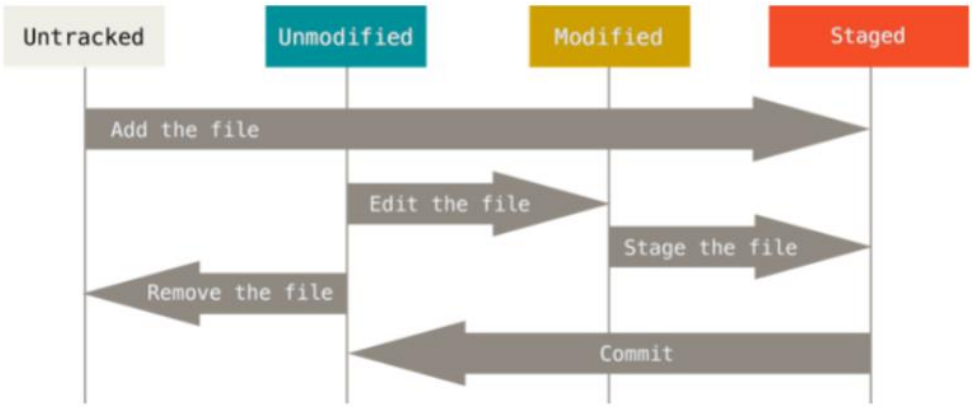
Особенностью модели можно считать то, что она направлена на тщательную проверку и тестирование продукта, находящегося уже на первоначальных стадиях проектирования. Стадия тестирования проводится одновременно с соответствующей стадией разработки, например, во время кодирования пишутся модульные тесты.



i. **Достоинства**
    1. В модели особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта на ранних стадиях его разработки.
    2. В модели предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта.
    3. В V-образной модели определение требований выполняется перед разработкой проекта системы, а проектирование ПО — перед разработкой компонентов.
    4. Модель определяет продукты, которые должны быть получены в результате процесса разработки, причём каждые полученные данные должны подвергаться тестированию.
    5. Благодаря модели менеджеры проекта могут отслеживать ход процесса разработки, так как в данном случае вполне возможно воспользоваться временной шкалой, а завершение каждой фазы является контрольной точкой
ii. **Недостатки**
    1. Модель не предусматривает работу с параллельными событиями.
    2. В модели не предусмотрено внесение требования динамических изменений на разных этапах жизненного цикла.
    3. Тестирование требований в жизненном цикле происходит слишком поздно, вследствие чего невозможно внести изменения, не повлияв при этом на график выполнения проекта.
    4. В модель не входят действия, направленные на анализ рисков.
    5. Некоторый результат можно посмотреть только при достижении низа буквы V.
iii. **Применение**
    1. Если требуется тщательное тестирование продукта, то V-модель оправдает заложенную в себя идею: validation and verification.
    2. Для малых и средних проектов, где требования четко определены и фиксированы.

3. В условиях доступности инженеров необходимой квалификации, особенно тестировщиков.

6) **Жизненный цикл файлов в гите**



7) **Модели распределенной разработки**

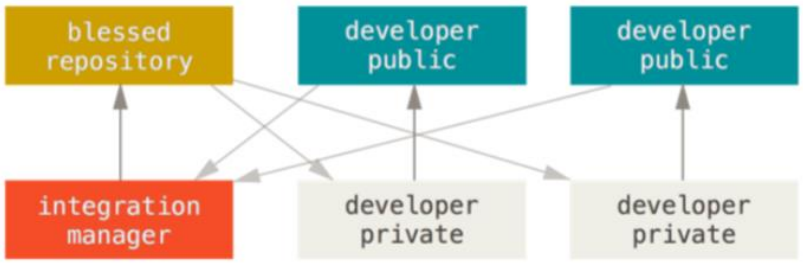
Определяют способы взаимодействия между разработчиками в рамках одного проекта, но не внутреннее устройство репозиториев. Выбираются в зависимости от величины проекта.

a. **Менеджер по организации**

Так как Git допускает использование нескольких удалённых репозиториев, то становится возможным организация рабочего процесса, где каждый разработчик имеет доступ на запись в свой публичный репозиторий и доступ на чтение ко всем остальным. При таком сценарии обычно существует канонический репозиторий, который представляет собой «официальный» проект. Для отправки своих наработок в этот проект следует создать его клон и отправить изменения в него. Затем вы отправляете запрос на слияние ваших изменений сопровождающему основного проекта. В свою очередь он может добавить ваш репозиторий как удаленный, протестировать ваши изменения локально, слить их в соответствующую ветку и отправить в основной репозиторий. Процесс работает в следующей последовательности:

i. Сопровождающий проекта отправляет изменения в свой публичный репозиторий.
ii. Участник клонирует этот репозиторий и вносит изменения.
iii. Участник отправляет свои изменения в свой публичный репозиторий.
iv. Участник отправляет письмо сопровождающему с запросом на слияние изменений.
v. Сопровождающий добавляет репозиторий участника как удалённый и сливает изменения локально.

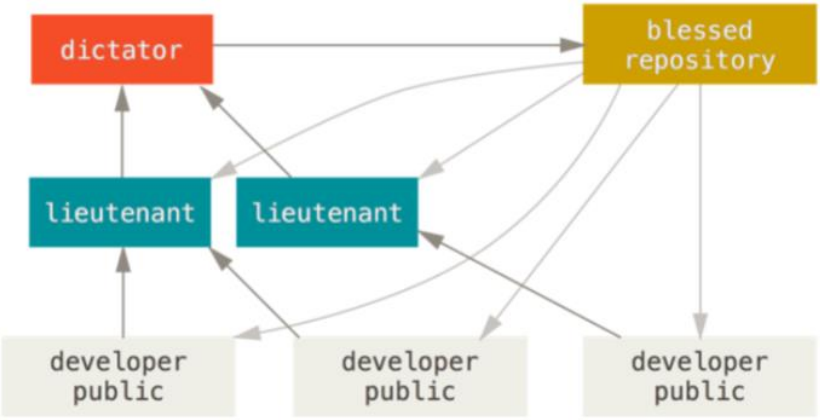
vi. Сопровождающий отправляет слитые изменения в основной репозиторий.



Это достаточно распространённый вид организации рабочего процесса, в котором используются хабы, такие как GitHub или GitLab, где легко создать свой репозиторий как ответвление проекта (fork) и отправить туда свои изменения. Одним из основных преимуществ этого подхода является то, что вы можете продолжать работать, а сопровождающий основного репозитория может слить ваши изменения в любое время. Участники не должны ждать пока основной проект примет их изменения — каждый может работать в своём темпе.

b. **Диктаторы и лейтенанты**
Это вариант организации рабочего процесса с использованием нескольких репозиториев. В основном такой подход используется на огромных проектах, насчитывающих сотни участников; самый известный пример — ядро Linux. Помощники (lieutenants) — это интеграционные менеджеры, которые отвечают за отдельные части репозитория. Над ними главенствует один диспетчер интеграции, которого называют великодушным диктатором. Репозиторий доброжелательного диктатора выступает как эталонный (blessed), откуда все участники процесса должны получать изменения. Процесс работает следующим образом:
   i. Обычные разработчики работают в своих тематических ветках и перебазируют свою работу относительно ветки master. Ветка master — это ветка эталонного репозитория, в которую имеет доступ только диктатор.
   ii. Помощники сливают тематические ветки разработчиков в свои ветки master.
   iii. Диктатор сливает ветки master помощников в свою ветку master.

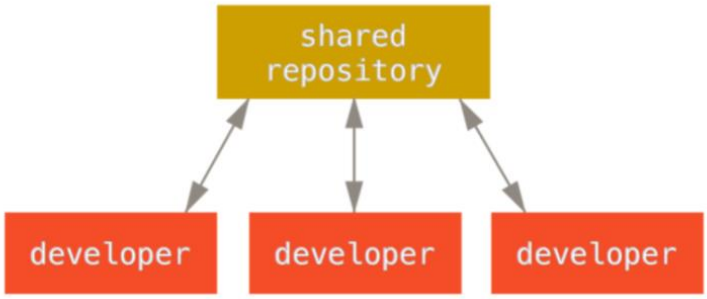
iv. Наконец, диктатор отправляет свою ветку master в эталонный репозиторий, чтобы все остальные могли перебазировать свою работу на основании неё.



Такой вид организации рабочего процесса не является обычным, но может быть применён к очень большим проектам или в сильно иерархической среде. Это позволяет лидеру проекта (диктатору) делегировать большую часть работы и собирать большие подмножества кода в различных точках до того, как они будут интегрированы.

c. **Централизованная**

В централизованных системах, как правило, присутствует только одна модель взаимодействия — централизованный рабочий процесс. Центральный хаб или репозиторий может принимать код, а все остальные синхронизируют свою работу с ним. Все разработчики являются узлами (пользователями хаба) и синхронизируются только с ним.



Это означает, что если два разработчика клонируют репозиторий и каждый внесёт изменения, то первый из них сможет отправить свои изменения в репозиторий без проблем. Второй разработчик должен слить изменения, сделанные первым разработчиком, чтобы избежать их перезаписи во время отправки на сервер.

8) **Git-hooks**

Пользовательские скрипты, срабатывающие при возникновении определенного события (Скрипт для pre-commit, pre-rebace и т.д.).

   a. **Виды**
      i. Клиентские
      ii. Серверные
   b. **Использование**
      i. Для привязки коммита к задаче
      ii. Для запрета коммита в ветку
      iii. Для запуска утилит статического анализа
      iv. Для чего угодно… (Надеюсь Макса такой ответ устроит)

9) **Непрерывная интеграция**

Практика разработки ПО, которая заключается выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем.

10) **Непрерывное развертывание**

Процесс развертывания проверенных фич из промежуточного окружения в промышленное, где они будут готовы к релизу.

11) **Непрерывная доставка**

Практика разработки программного обеспечения, когда при любых изменениях в программном коде выполняется автоматическая сборка, тестирование и подготовка к окончательному выпуску.

12) **Назначение тестирования**

Деятельность на протяжении всего ЖЦ ПО, направленная на: анализ и поддержку качества продукта, а также на определение соответствия реального ПО бизнес-требованиям.

13) Способы тестирования ПО

   a. **Функциональное тестирование**

   Тестирование в целях проверки реализации функциональных требований.
      i. Функциональные требования — ожидаемое поведение ПО; те функции, которое ПО должно выполнять.
      ii. Проводится на всех уровнях тестирования.
      iii. Тестами должны быть покрыты все функциональные требования!
      iv. Проверяются:
         1. корректные наборы данных — система должна правильно выполнять свои функции;
         2. некорректные наборы данных — система должна быть устойчива к ошибкам.

b. **Test-case**

Сценарий теста для проверки определенного функционала.

i. Могут объединяться в тестовый комплект (набор), один комплект закрывает одно требование к продукту. По итогу все требования заказчика должны быть покрыты тест-кейсами. Нежелательна зависимость одного тест-кейса от другого.

ii. Атрибуты тест-кейса (и, по сути, структура):

1. id
2. названия (раскрывающее суть)
3. предусловия — условия, которые должны быть выполнены, чтобы начать прохождение теста
4. шаги — четко сформулированная последовательность действий для получения результата
5. ожидаемый результат — кейс проверяет вполне конкретную ситуацию, потому результат должен быть четко определен в единственном экземпляре

iii. Результат теста может быть положительным (если фактический результат совпал с ожидаемым), отрицательным (не совпал, ошибка), неопределенным (тест блокирован, либо из-за того, что нельзя продолжить тестирование после некоторых шагов, либо потому что не выполнены предусловия).

c. Smoke-тесты

i. Исполняются с целью проверить что критически важные функциональные части AUT работают как положено.

ii. Цель — проверить «стабильность» системы в целом, чтобы дать зелёный свет проведению более тщательного тестирования

iii. Перепроверка дефектов не является целью Smoke

iv. Может выполняться автоматизированно или вручную.

d. Sanity-тесты

i. Нацелено на установление факта того, что определённые части AUT всё так же работают как положено после минорных изменений или исправлений багов.

ii. Целью является проверить общее состояние системы в деталях, чтобы приступить к более тщательному тестированию.

iii. Перепроверка дефектов не является целью Sanity.
iv. Чаще выполняется вручную.
v. Санитарное может выполняться без тест-кейсов, но знание тестируемой системы обязательно.

e. Уровни тестирования

i. Модульное тестирование или unit-тесты

1. Процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

2. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

ii. Интеграционное тестирование

1. Предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).
2. **Уровни интеграционного тестирования:**
    a. Компонентный интеграционный уровень (Component Integration testing)
        i. Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
    b. Системный интеграционный уровень (System Integration Testing)
        i. Проверяется взаимодействие между разными системами после проведения системного тестирования.
3. Подходы к интеграционному тестированию:
    a. Снизу вверх (Bottom Up Integration)
        i. Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей

для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения

b. Сверху вниз (Top Down Integration)
    i. Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз.
c. Большой взрыв ("Big Bang" Integration)
    i. Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

iii. Системное тестирование
    1. Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная

функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

2. Можно выделить два подхода к системному тестированию:
    a. **На базе требований** (requirements based)
        i. Для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.
    b. **На базе случаев использования** (use case based)
        i. На основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (test cases), которые должны быть протестированы.

iv. Приемочное тестирование
    1. Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:
        a. определения удовлетворяет ли система приемочным критериям;
        b. вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.
    2. Приемочное тестирование выполняется на основании набора типичных тестовых случаев и сценариев, разработанных на основании требований к данному приложению.
    3. Решение о проведении приемочного тестирования принимается, когда:
        a. продукт достиг необходимого уровня качества;

b. заказчик ознакомлен с Планом Приемочных Работ (Product Acceptance Plan) или иным документом, где описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д.

4. Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

f. **Автоматизация тестирования**
Для более-менее простых тестов (смок-тесты, некоторые санитарные тесты, часть модульных тестов) необязательно выполнять человеку, затрачивая человеко-часы на рутинную работу, можно написать программу-тестер, которая будет выполнять тесты автоматизированно.

i. Автоматизированное тестирование — шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи специализированных библиотек, фреймворков и инструментов (хотя для простых случаев можно обойтись и возможностями языка).
ii. Хорошо сочетается с системами CI/CD.
iii. Примеры библиотек и фреймворков:
1. Gtest, Catch, Catch2 (рекомендуем) — для C++;
2. JUnit — для Java, возможно и для Kotlin;
3. Unittest — для Python;
4. Selenium — для поведенческого тестирования веб-приложений (реализован на основе разбора DOM).

14) **Методология разработки**
Набор правил и приемов, использующихся при разработке, которые позволяют достичь определенных качеств процесса реализации или разрабатываемого продукта (скорость разработки, непрерывная поставка версий, качество продукта и т.д.).

a. выбирается в зависимости от специфики продукта и компетенций команды;
b. допускаются мотивированные отклонения или дополнения к принципам методологии в начале работы над продуктом.

15) TDD
методология разработки, основанная на повторении коротких итераций:

a. Пишется тест, закрывающий желаемое изменение
   Тест пишется на основе требований к системе. Также могут изменяться уже существующие тесты. Измененные тесты не должны проходить по условию.
b. Пишется такой код, который пройдет все тесты
   Подгоняем поведение под тест, возможно неидеальным способом.
c. Рефакторинг получившегося решения
   Если все тесты проходят, можно изменить неидеальное решение на более чистое и идиоматическое. Рефакторинг может производиться раз в несколько итераций.
d. **Достоинства:**
   i. фокусировка на требованиях;
   ii. тесты являются документацией;
   iii. продукт обладает только необходимой пользователю функциональностью;
   iv. интерфейс к модулю проектируется при создании теста;
   v. модули обладают меньшей связностью (так как сильно-связанные модули сложнее тестировать, а программисты ленивые);
   vi. увеличение временных затрат на создание тестов компенсируется уменьшением временных затрат на отладку;
   vii. уменьшается риск пропуска ошибок.
e. **Недостатки**
   i. тесты пишет программист: в случае недостаточно формализованных требований, неправильная трактовка требования приведет к неправильной реализации при полной уверенности, что все идет хорошо;
   ii. ложное ощущение безопасности приводит к снижению интенсивности на высоких уровнях тестирования;
   iii. обнаружение поздних ошибок приводит к необходимости изменения уже существующих тестов;
   iv. тесты могут содержать ошибки;
   v. сложно полноценно использовать для разработки GUI-приложений.

16) BDD

Расширение TDD, «мостик» между техническими специалистами (программисты, тестировщики, системные администраторы…) и управляющим персоналом (аналитики, менеджеры).

a. Поведение и результаты работы системы описываются на языке предметной области, близком к естественному, понятном специалистам разных лагерей.
b. Используется для сложных предметных областей.
c. Акценты BDD:
    i. бизнес-цели;
    ii. заинтересованные лица;
    iii. сценарии использования системы.
d. На основе сценариев создаются тесты (автоматически или вручную).
e. Достоинства:
    i. упрощение коммуникации между всеми участниками проекта, т.к. поведение описано понятным языком;
    ii. описания бизнес-целей и их сценарии являются документацией;
    iii. упрощение интерпретации результатов тестов;
    iv. сценарии легко портируются между языками и платформами;
    v. при наличии фреймворков для языка, из «человеческого» описания можно получить почти готовые тесты.
f. Недостатки:
    i. долго, дорого;
    ii. этим никто не будет заниматься, кроме программистов (потому что фу, сложно);
    iii. требует предварительного глубокого изучения предметной области всеми участниками проекта (а не только аналитиками);
    iv. даже при наличии фреймворка с парсером описаний поведения, адаптация этого описания к тесту может занимать больше времени, чем написание теста с нуля;
    v. усложнение изменения тестов (т.к. добавился новый уровень абстракции).