Разработка ПО, 4 сем. ПИ, Конспекты

Собрано 2 июня 2023 г. в 07:27

Содержание

Раздел #1: Разработка ПО

1.1. Требования, виды требований, стейкхолдеры. Бизнес-анализ, системный анализ

Определение 1 (Требования). Требования — это спецификация того, что должно быть реализовано. Это описание того, как система должна вести, описание ее свойств или атрибутов. Они могут вносить ограничения на процесс разработки системы.

Замечание. Отражение взгляда на систему заказчиков, разработчиков и пользователей (стейкхолдеров).

Определение 2 (Виды требований). Фундаментально, требования разделяют на функциональные и нефункциональные требования.

- Функциональные требования — требования, которые отвечают на вопросы о том, что система должна сделать с точки зрения алгоритмов.
- Нефункциональные требования — все требования, что не относятся к алгоритмам системы.

Приведем следующий пример с приложением, где можно покупать билеты на электрички. Требование уметь покупать билеты на электрички является функциональным требованием. С другой стороны, требование к удобному интерфейсу, например, не является функциональным требованием.
Вообще разница может показаться довольно размытой. Но часто проводят следующую грань: все, что относится к алгоритмам, является функциональными требованиями, а все, что относится к тому, как система выглядит и какими свойствами обладает, является нефункциональными требованиями.
Тем не менее, данная классификация не является единственной.

- Бизнес-требования. Это такие высокоуровневые требования, которые позволяют достигать целей с точки зрения бизнеса.
- Бизнес-правила. Более мелкие требования по сравнению с бизнес-требованиями.
- Ограничения.
- Требования по внешним интерфейсам. Как система взаимодействует с конкретным пользователем и внешними по отношению к ней системами (источники и получатели данных, например).

- Возможности системы (фичи). То, что в широком смысле она умеет делать в рамках реализации бизнес-требований. Например, добавлять определенные маршруты в избранные.
- Требования по качеству. Например, покрытие всего кода тестами с определенным процессом, бесперебойная работа системы под определенной нагрузкой.
- Требования, выдвигаемые к системе в целом. То есть, атрибуты, которыми обладает не отдельный компонент системы, а она вся. Например, доступность системы в течение 99.99% времени.
- Пользовательские требования. Те требования, которые выдвинули пользователи, очевидно.

Определение 3 (Стейкхолдеры). Стейкхолдеры — это люди, являющиеся источниками требований. Как правило, они заинтересованы в разрабатываемой системе.

- Заказчики.
- Разработчики.
- Пользователи, операторы.
- Бенефициары (те, кто получит какую-то выгоду от продажи продукта).
- Ответственные за покупку, продажу, установку системы.
- Регулирующие органы.
- Конкуренты.
- Вертикально интегрированные организации. Например, если компания занимается нефтепереработкой, то компании по нефтедобыче и нефтепродаже являются таковыми.
- Партнеры по горизонтальной интеграции. Похожи на конкурентов, но являются дружественными компаниями.

Замечание. Существует несколько способов документирования требований.

- Software Requirements Specification.
- Use cases.
- User stories.
- ...

1.2. Software Requirements Specification: назначение, общая структура.

Перспектива продукта

1.3. Software Requirements Specification: функции продукта, характеристики пользователя, ограничения

1.4. Software Requirements Specification: допущения, внешние интерфейсы, функции

1.5. Software Requirements Specification: атрибуты программной системы, верификация

Определение 4 (SRS). SRS ~ Техническое задание.

Определение 5 (Структура SRS). Рассмотрим структуру SRS.

- **SRS overview** (обзор SRS). Краткое описание документа: этот документ содержит ..., разрабатывается ..., для ..., и т. д.
- Purpose (цель, ради которой система разрабатывается). Какие бизнес-задачи решает данная система. Здесь это все расписывается не очень подробно.
- Scope (границы проекта). Что система должна делать, а чего не должна и не будет (на высоком уровне).
- **Product perspective** (перспектива продукта). Что из себя представляет продукт.
  1. System interfaces. Как происходит взаимодействие с системным программным обеспечением.
  2. User interfaces. Как пользователи взаимодействуют с системой.
  3. Hardware interfaces. Как происходит взаимодействие с железом.
  4. Software interfaces. Как система взаимодействует с какими-то программами (библиотеками), например.
  5. Communications interfaces. Как происходит взаимодействие с внешним миром. Например, с сетью.
  6. Memory. В какие ограничения по памяти укладывается система.
  7. Operations. Что система позволяет сделать с технической точки зрения. Например, создавать документы. (Еще пример с требованием выгрузки данных раз в какое-то время тоже можно отнести сюда).
  8. Site adaptation requirements. Что необходимо донастроить или докупить на месте развертывания системы.
  9. Interfaces with services. С какими внешними сервисами система взаимодействует.

- **Product functions**. Описывается, что продукт должен уметь делать, какие функции выполнять (уже более подробно). Вообще все, что удовлетворять бизнес-цели.
- **User characteristics**. Каких пользователей мы ожидаем для нашей системы.
- **Limitations** (ограничения).
    1. Законодательные требования.
    2. Hardware limitations. Минимальные системные требования, рекомендуемые, ...
    3. Interfaces to other applications. API, который мы предоставляем наружу.
    4. Parallel operation. Может ли система работать параллельно.
    5. Audit functions (функции по аудиту). Некий организационный контроль за тем, что бизнес-требования выполняются.
    6. Control functions. Контроль за тем, что пользователю разрешено делать, а что запрещено, например. Контроль за какими-то подозрительными действиями внутри системы.
    7. Higher-order language requirements. Здесь указывается, например, парадигма программирования. Речь также про языки и про то, что стоит выражаться аккуратно, возможно.
    8. Signal handshake protocols. То, как мы передаем данные, например (TCP/IP).
    9. Quality requirements (требования по качеству). Например, бесперебойная работа одного пользователя на определенной конфигурации железа (165 кадров в секунду, например).
    10. Criticality of the application (критичность приложения). Качественная работа одного приложения 24/7 может быть намного критичнее другого.
    11. Safety and security considerations.
    12. Physical and mental considerations (ограничения по физическим и умственным соображениям). Возможно ли систему использовать людям с ограниченными возможностями.
- **Assumptions and dependencies** (допущения и зависимости). Здесь описываются некие аксиомы, которые помогают при разработке приложения. Например, можно считать, что за три года браузеры, которые используются, не устареют. Используемые библиотеки также могут быть описаны здесь. Какие-то внешние зависимости, над которыми контроля не имеется.
- Apportioning of requirements.
- Specified requirements.

- **External interfaces** (внешние интерфейсы). Здесь интерфейсы расписываются более подробно по сравнению с пунктами перспектив продукта и его ограничений. Описываются внешние интерфейсы. Например, какие входы и выходы ожидаем (в широком смысле). Например, форматы файлов, которые поступают на вход.
- **Functions** (функции). Здесь функции расписываются более подробно по сравнению с пунктами перспектив продукта и его ограничений. Происходит детализация алгоритмики.
  1. Validity checks on the inputs (проверка входных данных на корректность).
  2. Описание последовательности операций.
  3. Обработки исключений.
  4. Эффект входных параметров.
  5. Как вывод связан со входом.
- ...
- **Software system attributes** (атрибуты программной системы). Какими словами можно описать систему. Качества.
  1. Reliability (надежность и что под ней понимается).
  2. Availability (доступность: всем категориям лиц, например).
  3. Security (безопасность: что данные никуда не утекают, например).
  4. Maintainability (обслуживаемость).
  5. Portability (переносимость с одного железа на другой).
- **Verification** (верификация). Описание проверки всех требований выше. Как именно проверяется, что требования выполнены.
- Supporting information.

Замечание. В настоящее время документы с полным содержанием SRS пишутся редко.

1.6. (Use cases) Прецеденты: назначение, диаграмма прецедентов, форматы описания

Определение 6 (Система). Система — разрабатываемая программная или программно-апаратная система. То, что разрабатывается.

Определение 7 (Актор). Актор — внешняя по отношению к системе сущность, которая или с которой взаимодействует система.

Определение 8 (Прецедент). Прецедент — последовательность действий между актором (-ами) и системой, приводящая к достижению какой-либо цели.

Замечание. Изначально, для документирования в форме прецедентов использовалась UML диаграмма прецедентов.

Определение 9 (Форматы описания Use Cases). Существует пара подходов к описанию самого прецедента.

- Кобёрн.
  - Title (цель).
  - Primary Actor (первичный актор, который инициирует этот прецедент).
  - Scope (описание той части системы, с которой актор начинает взаимодействовать).
  - Level (уровень предецента: насколько важно его реализовывать).
  - Story (что происходит в системе, можно неформально).
- Фаулер.
  - Title.
  - Main Success Scenario (главный сценарий успеха). Нумерованный список шагов, по которому достигается цель.
  - Extensions (расширения). Список, в котором описывается, на каком шаге что-то может пойти не так (показываем, что будет происходить в другом случае).

Замечание (Уровни прецедентов). Уровней прецедентов существует несколько.

- Very high summary.
- Summary.
- User Goal.
- Subfunction.
- Too Low.

Замечание. В отличие от SRS, подход с прецедентами фокусируется на том, как система взаимодействует с пользователями, как она внутри себя работает и что делает. Основной фокус делается на функциональных требованиях. UI, например, тут не описать.

1.7. (User stories) Пользовательские истории: назначение, общий шаб-

лон, виды

Замечание. Подход с SRS хорош тем, что он показывает все множество требований к системе в едином документе. Подход с прецедентами хорош тем, что очень хорошо показывает алгоритмику. Подход с пользовательскими историями показывают, что требования могут со временем эволюционировать, становиться более подробными, изменяться.

Определение 10 (Пользовательская история). Пользовательская история — это описание функциональности ПО простыми, общими словами, составленное с точки зрения конечного пользователя. Она пишется с целью разъяснить, как именно функциональность принесет пользу клиенту.

Замечание. Это не совсем требование, а скорее его обсуждение.

Замечание. Выяснение требований в данном случае — это процесс, который не прекращается. С течением времени мы выявляем все новые и новые требования к системе.

Замечание (Шаблон). Как <роль пользователя>, я хочу получить <фича> с целью <цель>.

Замечание. Специальные собрания по проработке US — груминги (planning poker).

Определение 11 (Виды пользовательских историй). Выделяют разные виды пользовательских историй.

- Эпики (очень большие пользовательские истории, разработка которых может занимать недели и месяцы, поэтому они разбиваются на подзадачи).
- Задачи.
- Ошибки.
- Исследовательские задачи (они не всегда внедряются в систему).

Замечание. Пользовательские истории помечаются сложностью выполнения и важностью внедрения.

1.8. Планирование разработки. Треугольник(-и) управления проектами

Замечание. Предположим, что требования собраны. Далее их нужно реализовывать. Багтрекер здесь может помочь (например, Trello), но на деле все более сложно. Планирование разработки — отдельный вид деятельности.

- Ответственность лежит на менеджере проекта и руководителях команд (тим-лидах).

  Менеджер проекта — человек, который управляет всем процессом разработки на высоком уровне, отвечая за то, чтобы проект на этом же самом высоком уровне был завершен (проект реализован в срок, сделано то, что нужно, уложились в бюджет).

  При этом на тим-лида сваливаются задачи не столько стратегического хода проекта, сколько тактического, повседневного: кто получает какие задачи, что с ними делать и т.д.

- Поиск и предоставление необходимых ресурсов (менеджер проекта).
- Определение методологии разработки (менеджер проекта).
- Определение политики выпуска версий (менеджер проекта).
- Определение границ версий (менеджер проекта).
- Разбивка на задачи (и оценка трудоемкости), установка приоритетов, назначение задач командам и людям (менеджер проекта или тим-лид).
- Контроль за выполнением (менеджер проекта).

Замечание. Аккаунт менеджер — менеджер, который работает с несколькими проектами (менеджер менеджеров).

Определение 12 (Треугольник управления проектами). В управлении проектами существует классический треугольник, с помощью которого подчеркивается следующий тезис: можно варьировать цену, время и объем задачи, но не все три пункта одновременно.

Определение 13 (Закон Брукса). Добавление рабочей силы к позднему программному проекту делает его более поздним.

1.9. Методологии разработки

Замечание. Вспомним несколько известных методологий разработки.

- Водопадная модель.
- Спиральная модель.
- Scrum.

- Kanban (Scrum на минималках: убраны люди, которые относятся к методологии Scrum, например, Scrum-мастера).

1.10. Политика выпуска версий, определение границ версий, разбивка на задачи

Замечание (Политика выпуска версий). Политика выпуска версий — уровень рассуждения менеджера проекта. Есть несколько подходов.

- Крупные релизы редко.
- Крупные релизы редко, более мелкие в рамках большого релиза — чаще.
- Обновления последней версии время от времени (регулярные, нерегулярные).
- Периодическая фиксация версия, обновления последней версии время от времени (регулярные, нерегулярные).

Также сюда входит ответ на вопрос о том, на какие версии распространяется поддержка.

Замечание (Определение границ версий). Границы устанавливаются исходя из следующих пунктов.

- К какому сроку необходимо сделать работу.
- Сколько денег выделено на работу.
- Какая функциональность требуется заказчику.
- Какой уровень качества нужен заказчику.

Замечание (Разбивка на задачи). Чтобы из требований получить конкретный план действий, обсуждают следующие пункты.

- Что конкретно нужно сделать.
- Какие нужны ресурсы (материальные и нематериальные).
- Оценка трудоемкости.
- Назначение приоритета.
- Назначение ответственных.

Для разбивки на задачи есть Work Breakdown Structure, например.

1.11. Work Breakdown Structure

Определение 14 (Work Breakdown Structure). Иерархическая структура работ — способ декомпозиции проекта и диаграмма, соответствующая этой декомпозиции.
Диаграмма выглядит следующим образом. На самом верхнем уровне (в корне) находится результат, которого мы хотим достичь. Из корня появляются части, необходимые для выполнения этой задачи. Из каждой из частей могут появляться другие части.

- Самый верхний уровень — общий объем работы, весь проект.
- Далее — высокоуровневые составные части проекта.
- Ниже — части частей.

Замечание. Стоит понимать, что результат проект формулируется с точки зрения отчуждаемых артефактов (то, что сделано и может быть передано, например, программа).
Строго говоря, WBS не является to-do list'ом, поскольку со временем понимание о проекте растет, могут отвалиться какие-либо части, которые оказываются не очень-то и нужными. Может и наоборот, добавиться что-то новое.
Также это не mind map.
На всех уровнях работает правило 100%: сумма объема дочерних узлов в точности равна объему родительского узла.
В WBS входит все, на что приходится тратить время, даже, например, на общение с заказчиком.

Замечание. Дробить задачу требуется настолько сильно, насколько логично.
Также существует мнение, что если на задачу уходит более 80 часов, то, скорее всего, задача содержит в себе какие-то составные части.
Еще один вариант — это дробление на такие задачи, оценка работы каждой из которых не будет превышать отчетного периода.

Замечание. Определяется, что надо сделать, но не как.

Замечание (Как делать). После того, как определились, что делать, необходимо договориться о том, как делать.

- Собрать все важные документы.
- Определить ключевых участников команды (аналитики, программисты, тестировщики, менеджер, ...).
- Определить элементы первого уровня (фазовая разработка или артефактная).
- Рекурсивно повторить предыдущий шаг для каждого элемента первого уровня.
- Создать словарь WBS (этапы работ, риски, стоимость, границы, и т. д.).

- Составить диаграмму Ганта.

1.12. Подходы к оценке трудоёмкости

Замечание. Существует несколько подходов к оценке трудоемкости.

- PERT. Использует математические методы. Основывается на нормальном распределении.

  Берем три оценки: оптимистическую, нормальную (наиболее вероятный исход), пессимистическую.

  Оценка трудоемкости строится как оценка нормального распределения. Математическое ожидание вычисляется следующим образом (грубоватая оценка математического ожидания нормальной случайной величины):

$$\mu = \frac{optimistic + 4 * normal + pessimistic}{6}.$$

  Стандартное отклонение:

$$\sigma = \frac{pessimistic - optimistic}{6}.$$

  Оно позволяет оценить наиболее вероятный разброс по времени работ.

  Для последовательности задач:

$$\mu_{seq} = \sum \mu_{task};$$

$$\sigma_{seq} = \sqrt{\sum \sigma^2_{task}}.$$

- Широкополосный дельфийский метод.

  Группа экспертов обсуждает и оценивает задачу до тех пор, пока не придет к согласию (абсолютность и формальная подкрепленность не совсем обязательна).

- Planning poker.

  Игра.

  Каждый участник одновременно выкладывает свою карточку.

  Предложившие максимальное и минимальное значение объясняют свой выбор.

  Очки могут быть, например, не в человеко-днях, а, например, в спринтах.

1.13. Расстановка приоритета задач

Замечание. Приоритеты задач, несомненно, зависят от стейкхолдеров.
Если говорить о новой функциональности, то приоритет тем выше, чем сильнее она нужна ключевым стейкхолдерам.

Если речь о багах или техническом долге, то нужно ответить на следующие вопросы: насколько сильно проблема ломает работу системы, а также сколько подсистем, пользователей, иных стейкхолдеров задето.

Замечание. В современных багтрекерах используют следующие приоритеты.

- Critical.
- High.
- Normal.
- Low.
- Nice to have.

Замечание. Выбор задач может быть следующим.

- Какую скажет начальник.
- В порядке уменьшение приоритета задачи.
- В порядке уменьшения приоритета версии, затем в порядке уменьшения приоритета задачи.
- Любую, которая больше нравится.

1.14. Этические аспекты разработки ПО: общие принципы

1.15. Этические аспекты разработки ПО: профессиональная ответственность

Определение 15 (Этика). Этика — раздел философии, учение о том, что хорошо, а что плохо.

Замечание (ACM Code of Ethics and Professional Conduct). Кодекс этики и профессионального поведения ACM содержит следующие общие принципы.

- Вносите посильный вклад в общество и в человеческое благополучие, а также имейте в виду, что все люди каким-либо образом связаны с вычислительной техникой.
- Избегайте вреда.

  Этот и предыдущий пункт несут следующие идеи: старайтесь делать людям хорошо, не делать плохо. Например, если вас заставляют взламывать сайты, либо вскрывать чужой код, который не должен быть доступен, то вы этого делать не должны. Также стоит избегать насилия, как физического, так и психологического.

- Будьте честными и заслуживающими доверия.

  Это основа вашей профессиональной репутации как в коллективе, так и вне. Если вас спрашивают о чем-то, то давайте максимально честный ответ. А также держите свои слова.

- Будьте справедливы и не подвергайте других дискриминации.

- Уважайте труд, относящийся к производству новых идей, изобретений, артефактов программирования.

  Любой труд является тратой времени. Сколь бы бесполезной вам какая-либо идея не казалось, не нужно высказываться агрессивно. Лучше обсудить с автором ее достоинства и недостатки.

- Уважайте приватность и чтите конфиденциальность.

  Как разработчик, вам часто может приходиться работать с конфиденциальными данными. Нельзя распространять секреты, которые вам доверяют. Более того, стоит заботиться об анонимизации.

Следующая часть этического кодекса посвещена профессиональной ответственности.

- Старайтесь обеспечивать высокое качество как процессов профессиональной деятельности, так и продуктов.

  Например, программист, который регулярно коммитит в проект низкокачественный код, который не использует языковые фишки, который реализовывает функциональность неправильно, является плохим. Нужно стремиться к тому, чтобы повышать свой профессионализм.

- Поддерживайте высокие стандарты профессиональной компетенции и этической разработки.

  Принцип похож на один из предыдущих: уважать труд коллег.

- Знайте и уважайте существующие правила, касающиеся профессиональной работы.

  Если в коллективе приняты определенные правила разработки, то придерживайтесь их, не выражайтесь о них плохо.

- Принимайте и предоставляйте профессиональное ревью.

  Конструктивное код ревью — очень важный навык.

- Давайте полную оценку относительно системы, включая анализ и возможные риски.

  Если проблему выявить раньше, то стоимость решения проблемы может быть в разы дешевле.

- Работайте только в сферах своей компетенции.

- Способствуйте распространению честной информации о том, что из себя представляют определенные технологии и системы, а также каковы последствия от их применения.

  Лучше, чтобы широкая публика знала, что ваша система не несет вреда. Если же есть какой-то вред, то об этом нужно говорить открыто и прямо.

- Не заходите в те или иные системы не под своими учетными записями.

- Разрабатывайте системы, которые надежны и безопасны в применении.

1.16. Ветвление в репозитории: тривиальное, Gitflow

Замечание. Репозиторий — место, в которые складируется исходный код. Стоит, конечно, использовать систему контроля версий, а не какой-нибудь файлообменник.
Стандарт в наше время — git.

Замечание (Подходы к организации репозиториев). Существует несколько подходов к организации репозиториев.

- Монорепозиторий.
  - Весь код в одном месте.
  - Но не всегда он весь нужен.
  - Хранилище артефактов необязательно.
- Несколько репозиториев, каждый на один отчуждаемый артефакт разработки.

Замечание (Модели ветвления). Существует несколько моделей ветвления.

- Тривиальная — одна ветка main.
- Gitflow.
- GitHub flow.
- GitLab flow.

Замечание (Тривиальная модель). Отсутствие ветвлений вообще (просто одна ветка).

- Подходит для очень маленьких команд и очень маленьких приложений.
- Поддерживает CI/CD.
- Основную ветку элементарно сломать.
- Скорее всего, тестирование происходит руками на компьютерах разработчиков.

Замечание (Gitflow). Ветки, очевидно, нужны, чтобы у разработчика была возможность разрабатывать систему не в основной ветке. Это позволяет дорабатывать функциональность до момента, когда уже можно мерджить в основную ветку.
Помимо прочего, они полезны для отслеживания истории разработки.
В рамках данного подхода, в первую очередь, выделяются ветки master и develop. В мастер попадает протестированный код, который официально является релизом.
Также, для разработки делаются отдельные ветки (фичи или багфиксы), которые создаются от develop ветки.
При этом, от develop также могут создавать релизные ветки, которые используются для того, чтобы отполировать релиз до публикуемого состояния и замерджить в мастер (а также и в develop).
При этом, от мастера могут создаваться другие ветки для хотфиксов.

Замечание. Рассмотрим преимущества и недостатки.

- Хорошо подходит для поддержки нескольких версий, но не очень для одной.
- Каждая ветка содержит актуальное состояние кода.
- История коммитов очень трудно читаема.
- Затрудняет CI/CD (для master и develop ветки сделать можно, но как быть с релизными ветками — отдельный вопрос).

1.17. Ветвление в репозитории: GitHub flow, GitLab flow

Замечание (GitHub flow). Используются следующие идеи.

- Главная master-ветка.
- Множества feature-веток.
- Пулл реквесты для merge.

Замечание. Преимущества и недостатки.

- Master легко сломать.
- Хорошо подходит, когда нужна только самая актуальная версия продукта.
- Хорошо ложится на CI/CD.

Замечание (GitLab flow). Идеи.

- Главная ветка разработки — master.

- Коммиты в master автоматически развертываются в тестовом окружении.
- Развертывание в пре-проде делается вручную или автоматически.
- Развертывание в проде — вручную.

Зачем нужен pre-production? Поскольку в master может накапливаться мусор, в pre-production можно содержать более чистый код, более комфортный для разработчика. Это еще одна возможность удостовериться, что в production все будет хорошо.

Замечание. Плюсы и минусы.

- Удобно, если необходимо поддерживать только один релиз, чище история коммитов.
- Если несколько — тоже удобно, но история коммитов не такая чистая.
- Из организации веток логично вытекает архитектура CI/CD.

1.18. Тестирование ПО: основные термины и концепции

Определение 16 (Баг ПО). Баг ПО — это отклонение фактического результата от ожидаемого.
Баг существует, если известны следующие пункты.

- Ожидаемый результат.
- Фактический результат.
- Факт их отличия.

Любое тестирование — поиск багов.

Определение 17 (Источники ожидаемого результата). Перечислим источники ожидаемого результата.

- Спецификация.
- Жизненный опыт и здравый смысл.
- Общение.
- Устоявшиеся стандарты, статистические данные, авторитетное мнение.

Замечание. Тестирование и QA не тождественны друг другу.

- Тестирование — нахождение багов до того, как их найдут пользователи.

- QA — работа над качеством путем предупреждения появления багов (совершенствование процесса разработки).

Замечание. Цикл тестирования ПО.

- Изучение и анализ предмета тестирования.
  - Какие функциональности предстоит протестировать.
  - Как эти функциональности работают.
- Планирование тестирования.
  - Тест-кейсы.
  - Тест-документация.
- Исполнение тестирования.
  - Регрессионное тестирование.
  - Тестирование новых функциональностей.

1.19. Тестирование ПО: классификации тестирования

Замечание. Классификация тестирования ПО.

- По знанию внутренностей системы.
  - Черный ящик — тестирование основывается на паттернах взаимодействия с системой.
  - Белый ящик — тестирование основывается на знании внутренностей системы.
  - Серый ящик — сочетание обоих подходов.
- По объекту тестирования.
  - Функциональность.
  - UI.
  - Локализация.
  - Скорость и надежность (стресс-тестирование, нагрузочное тестирование).
  - Безопасность.
  - UX.
  - Совместимость.
- По времени тестирования.

  - Альфа-тестирование — до передачи пользователю.
  - Бета-тестирование — после передачи пользователю.
- По субъекту тестирования.
  - Альфа-тестировщик.
  - Бета-тестировщик.
- По критерию «позитивности сценариев»
  - Позитивное тестирование.
  - Негативное тестирование.
- По степени изолированности тестируемых компонентов.
  - Юнит-тестирование.
  - Интеграционное тестирование.
  - Системное тестирование.
- По степени автоматизации тестирования.
  - Ручное.
  - Автоматизированное.
  - Смешанное.