## **Введение в конструирование программного обеспечения**.

Программное обеспечение  (ПО) — [программа](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или множество программ, используемых для управления [компьютером](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) (*ISO/IEC 26514:2008*).

Имеются и другие определения из международных и российских стандартов:

* совокупность программ системы [обработки информации](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8) и [программных документов](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F_%D0%BD%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), необходимых для эксплуатации этих программ ([*ГОСТ*](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D1%81%D1%83%D0%B4%D0%B0%D1%80%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82)*19781-90*);
* все или часть программ, процедур, правил и соответствующей документации системы обработки информации (*ISO/IEC 2382-1:1993*);
* компьютерные программы, процедуры и, возможно, соответствующая документация и данные, относящиеся к функционированию компьютерной системы (*IEEE Std 829—2008*).

Программное обеспечение является одним из видов обеспечения [автоматизированной системы](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), наряду с техническим (аппаратным), математическим, информационным, лингвистическим, организационным, методическим и правовым обеспечением.

Академические области, изучающие программное обеспечение, — это [информатика](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и [программная инженерия](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D0%B8%D1%8F).[[1]](#footnote-1)

В [компьютерном сленге](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D0%B9_%D1%81%D0%BB%D0%B5%D0%BD%D0%B3) часто используется слово «*софт*», произошедшее от английского слова «*software*», которое в этом смысле впервые применил в статье журнала *[American Mathematical Monthly](https://ru.wikipedia.org/wiki/American_Mathematical_Monthly" \o "American Mathematical Monthly)* математик из Принстонского университета [Джон Тьюки](https://ru.wikipedia.org/wiki/%D0%A2%D1%8C%D1%8E%D0%BA%D0%B8,_%D0%94%D0%B6%D0%BE%D0%BD) в [1958 году](https://ru.wikipedia.org/wiki/1958_%D0%B3%D0%BE%D0%B4).

Термин конструирование программного обеспечения описывает детальное создание рабочей программной системы посредством комбинации кодирования, верификации (проверки), модульного тестирования, интеграционного тестирования и отладки.

Данная область знаний связана с другими областями. Наиболее сильная связь существует с проектированием (Software Design) и тестированием (Software Testing). Причиной этого является то, что сам по себе процесс конструирования программного обеспечения затрагивает важные аспекты деятельности по проектированию и тестированию. Кроме того, конструирование отталкивается от результатов проектирования, а тестирование (в любой своей форме) предполагает работу с результатами конструирования. Достаточно сложно определить границы между проектированием, конструированием и тестированием, так как все они связаны в единый комплекс процессов жизненного цикла и, в зависимости от выбранной модели жизненного цикла и применяемых методов (методологии), такое разделение может выглядеть по- разному.

Хотя ряд операций по проектированию детального дизайна может происходить до стадии конструирования, большой объем такого рода проектных работ происходит параллельно с конструированием или как его часть. Это есть суть связи с областью знаний «Проектирование программного обеспечения».

В свою очередь, на протяжении всей деятельности по конструированию, инженеры используют модульное и интеграционное тестирование. Таким образом, данная область знаний связана с «Тестированием программного обеспечения».

В процессе конструирования обычно создается большая часть активов программного проекта — конфигурационных элементов. Поэтому в реальных проектах просто невозможно рассматривать деятельность по конструированию в отрыве от области знаний «Конфигурационного управления» (Software Configuration Management).

Так как конструирование невозможно без использования соответствующего инструментария и, вероятно, данная деятельность является наиболее инструментально-насыщенной, важную роль в конструировании играет область знаний «Инструменты и методы программной инженерии» (Software Engineering Tools and Methods).

Безусловно, вопросы обеспечения качества значимы для всех областей знаний и этапов жизненного цикла. В то же время, код является основным результирующим элементом программного проекта. Таким образом, явно напрашивается и присутствует связь обсуждаемых вопросов с областью знаний «Качество программного обеспечения» (Software Quality).[[2]](#footnote-2)

Из связанных дисциплин программной наиболее тесная и естественная связь данной области знаний существует с компьютерными науками. Именно в них, обычно, рассматриваются вопросы построения и использования алгоритмов и практик кодирования. Наконец, конструирование касается и управления проектами, причем, в той степени, насколько деятельность по управлению конструированием важна для достижения результатов конструирования.

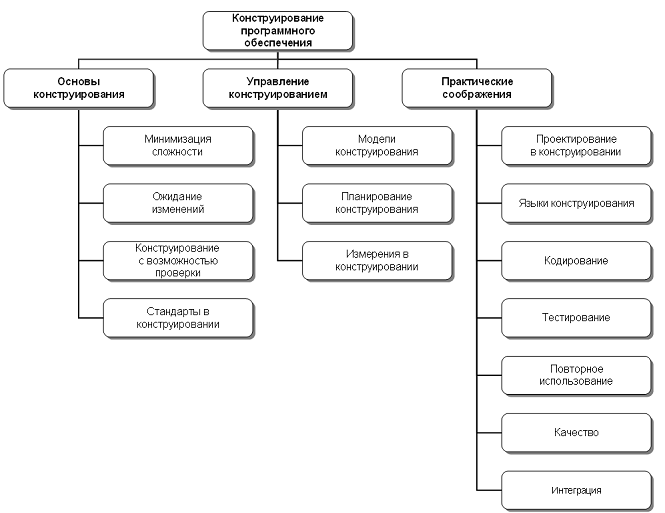
****

Рисунок 1.4.1 - КПО

## **1.5. Жизненный цикл программного обеспечения.**

Жизненный цикл[[3]](#footnote-3)– это модель создания и использования программной системы. Он отражает различные состояния программной системы, начиная с момента возникновения необходимости в этой программной системе и принятия решения о ее создании и заканчивая полным изъятием программной системы из эксплуатации.

Международный стандарт ISOIES 12207 регламентирует структуру жизненного цикла, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания программного обеспечения. По этому стандарту жизненный цикл программного обеспечения базируется на трех группах процессов:

1. основные процессы жизненного цикла, то есть приобретение, поставка, разработка, эксплуатация и сопровождение;
2. вспомогательные процессы, обеспечивающие выполнение основных процессов, то есть документирование, верификация, аттестация, оценка качества и другие;
3. организационные процессы, то есть управление проектами, создание инфраструктуры проекта и обучение.

Разработка включает в себя все работы по созданию программного обеспечения в соответствии с заданными требованиями. Сюда включаются оформление проектной и эксплуатационной документации, подготовка материалов, необходимых для проверки работоспособности и качества программных продуктов.

Основные этапы процесса разработки:

1. анализ требований заказчика;
2. проектирование;
3. реализация (программирование).

Процесс эксплуатации включает в себя работы по внедрению программного обеспечения в эксплуатацию, в том числе конфигурирование рабочих мест, обучение персонала, локализация проблем эксплуатации и устранение причин их возникновения, модификация программного обеспечения в рамках установленного регламента и подготовка предложений по модернизации системы.

Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными и результатами.

Жизненный цикл программного обеспечения носит, как правило, итерационный характер, то есть реализуются этапы, начиная с самых ранних, которые циклически повторяются в соответствии с изменением требований внешних условий и введением ограничений.

## **1.6. Управление проектом программного обеспечения.**

*Модели жизненного цикла программного обеспечения*

Существует несколько моделей жизненного цикла, которые определяют порядок исполнения этапов разработки и критерии перехода от этапа к этапу. К настоящему времени наибольшее распространение получили две модели жизненного цикла: каскадная и спиральная.

В существующих ранее однородных информационных системах каждое приложение представляло собой единое целое. Для разработки таких приложений применялась каскадная модель жизненного цикла, которую также называют классической или водопадной.

При использовании каскадной модели разработка рассматривалась как последовательность этапов, причем переход на следующий более низкий этап происходит только после того, как полностью завершены все работы на текущем этапе. Подразумевается, что в каскадной модели разработка начинается на системном уровне и происходит через анализ, проектирование, кодирование, тестирование и сопровождение.

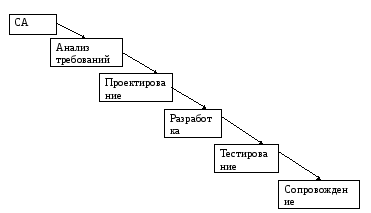


Рисунок 1.6.1 – Основные этапы разработки каскадной модели

1. Системный анализ задает роль каждого элемента в компьютерной системе и взаимодействие элементов друг с другом. Поскольку программное обеспечение рассматривается как часть большой системы, то анализ начинается с определения требований по всем системным элементам. Необходимость системного анализа явно проявляется, когда формируется интерфейс программного обеспечения с другими элементами, т.е. с аппаратурой или базами данных. На этом же этапе начинается решение задач планирования проекта. В ходе планирования проекта определяется объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к отдельному программному элементу. На этом этапе уточняются и детализируются функции каждого элемента, его характеристики и интерфейс. На этом же этапе завершается решение задачи планирования проекта.

2. Проектирование состоит в создании:

1. архитектуры программного обеспечения;
2. модульной структуры программного обеспечения;
3. алгоритмической структуры программного обеспечения;
4. структуры данных;
5. входного/выходного интерфейса (входных/выходных форм данных).

При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

3. Кодирование или разработка состоит в переводе результатов проектирования в код программы.

4. Тестирование – это выполнение программы на выявление дефектов в функциях, логике и форме реализации программного продукта.

5. Сопровождение – это внесение изменений в эксплуатируемое программное обеспечение с целью:

1. исправления ошибок;
2. адаптации к изменениям внешней для программного обеспечения среды;
3. усовершенствование программного обеспечения в соответствии с требованиями заказчика.

*Достоинства применения каскадной модели:*

1. дает план и временной график по всем этапам проекта, упорядочивая, таким образом, ход разработки;
2. на каждом этапе формируется законченный набор проектной документации, проверенный на полноту и согласованность;
3. выполняемые в логической последовательности этапы работы позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадная модель хорошо себя зарекомендовала при построении информационных систем, для которых в самом начале разработки можно достаточно точно сформулировать все требования в системе, например, сложные расчетные системы, различные системы реального времени и т.д.

*Недостатки каскадной модели:*

1. реальные проекты часто требуют отклонений от стандартной последовательности шагов;
2. каскадная модель основана на точной формулировке исходных требований к программному обеспечению, однако реально в ряде случаев в начале проекта требования заказчика определены только частично;
3. результаты реализации проекта доступны заказчику только после завершения всех работ.

Из-за необходимости в процессе создания программного обеспечения постоянного возврата к предыдущим этапам и уточнения или пересмотра ранее принятых решений реальный процесс разработки программного обеспечения на основе каскадной модели может быть представлен следующей схемой (рис. 1.6.2).

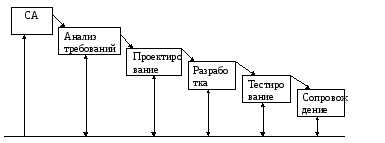


Рисунок 1.6.2 – Процесс разработки программного обеспечения на основе каскадной модели

*Спиральная модель жизненного цикла*

Спиральная модель является классическим примером эволюционной стратегии конструирования программного обеспечения.

Спиральная модель базируется на лучших свойствах каскадной модели жизненного цикла и макетирования, к которым добавляется анализ риска.

Спиральная модель включает четыре основных этапа, которые периодически повторяются:

1. планирование – это определение целей, вариантов и ограничений;
2. анализ риска – это анализ вариантов и распознавание риска;
3. конструирование – это разработка программного продукта следующего уровня;
4. оценивание – это оценка заказчика текущих результатов конструирования.

При движении по спирали строятся все более полные версии программного обеспечения при продвижении от центра к периферии. В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, то на помощь заказчику и разработчику приходит макетирование. Заказчик оценивает инженерную или конструкторскую работу и вносит предложения по модификации.

Следующая фаза планирования и анализа риска базируется на предложении заказчика. Если риск слишком велик, проект может быть остановлен. В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. Количество действий по разработке возрастает по мере продвижения от центра спирали.

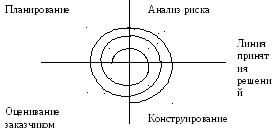


Рисунок 1.6.3 – Этапы спиральной модели

*Достоинства спиральной модели:*

1. наиболее реально отображает процесс разработки программного обеспечения;
2. позволяет явно учитывать риск на каждом витке эволюции разработки;
3. использует моделирование для уменьшения риска и совершенствования программного продукта.

*Недостатки спиральной модели:*

1. повышенное требование к заказчику;
2. трудности контроля и управления временем разработки.

*Все, что нужно знать о работе в скрам-команде*

**Scrum** – это способ организации командной работы, особенно хорошо подходящий для создания чего-то принципиально нового. Scrum возник как методология разработки программного обеспечения, а сейчас применяется в любой деятельности, где требуется коллективная работа: в производстве, в образовании, в госуправлении. Scrum принадлежит к Agile – семейству «гибких» подходов к управлению проектами. На каких принципах основана методология Scrum, как распределены роли и как организована работа команды, мы коротко расскажем в этой статье.

Традиционно работа над проектами строится «методом каскада». Разрабатывается подробный поэтапный план работы от начала до конца: определяются сроки выполнения каждого этапа, исполнители и бюджет, план согласовывается с заказчиком и начинается работа в соответствии с утвержденным планом. Чаще всего, а точнее, всегда, планы не выполняются: сроки не соблюдаются, бюджет проекта растет, заказчик получает продукт с ошибками и недоделками, или совсем не такой, на который он рассчитывал. Каждый, кто хоть раз делал ремонт, прекрасно понимает, о чем идет речь.

Главная идея Scrum – ориентация на клиента. Заказчик должен получить то, что хочет, вовремя и с минимальными затратами. Этого можно достичь, соблюдая принципы Scrum и организовав работу специальным образом.

#### На каких принципах основана методология Scrum?

**Работа происходит короткими циклами – спринтами.**Спринт – это период времени, в течение которого команда работает интенсивно и максимально быстро над полностью законченной частью продукта. Это может быть период от недели до четырех, но всегда один и тот же, чтобы команда уловила ритм работы и поняла, сколько она может сделать за определенное время. Команда планирует свою работу именно на спринт, а не на весь проект сразу.

**Команда гибко подходит к процессу создания продукта.**Продукт тестируется после каждого спринта. Если что-то идёт не так, команда всегда готова сменить стратегию. Команда анализирует, есть ли способы выполнять задачи еще быстрее и качественнее, и сразу внедряет найденные решения в свою работу.

**Заказчик и пользователи участвуют в создании продукта.** По окончании спринта заказчик получает полностью работающую функцию, которая может быть передана пользователям. Полученная от пользователей обратная связь позволяет своевременно обнаруживать просчеты и ошибки и корректировать план разработки.

**Команда работает как единое целое, стремясь к общей цели.** Сам термин scrum – схватка – заимствован из регби, где он обозначает метод командной игры, позволяющий завладеть мячом и вести его дальше по полю, для чего требуются слаженность, единство намерений и четкое понимание цели. Схватка представляет собой идеальную модель взаимодействия игроков.

#### Как устроена команда и как распределены роли?

В скрам-команде обычно работает от 5 до 9 человек – это оптимальное количество для эффективного взаимодействия. Команда ведет весь проект от начала до конца. Одна команда — один готовый продукт. Роли в команде распределены следующим образом.

**Владелец продукта.**Человек, который является посредником между заказчиком, пользователями и командой разработчиков. На встречах с заказчиками владелец продукта рассказывает, как продвигается разработка, узнает, отвечают ли их ожиданиям последние внесенные изменения и новые возможности. На встречах с командой он объясняет членам группы, что клиент оценил в продукте, что воспринял как недоделку и каковы его ожидания. Владелец продукта определяет концепцию проекта, собирает и формулирует пользовательские требования, несет ответственность за составление и ведение бэклога – списка всех требований к продукту, отсортированного по приоритетности.

**Scrum-мастер.** Чаще всего — специально нанятый сотрудник, который ведет команду к результату. Он не управляет командой, но наблюдает за исполнением принципов Scrum. Его задача — не давить, не делать всю работу самому и не распределять обязанности, а помогать, направлять и решать вопросы, которые тормозят процесс разработки. Scrum-мастер помогает команде регулярно искать ответ на вопрос «Как нам делать еще лучше то, что мы уже делаем хорошо?» Он поддерживает атмосферу доверия и открытости.

**Разработчики.**В составе команды разработчиков должны быть люди с разным набором навыков, но главное – с навыком взаимопомощи. Команда отвечает за результат как единое целое.

#### Как организована работа команды?

**Создание бэклога.** Владелец продукта составляет максимально полный список требований к функциональности продукта. Команда оценивает сложность каждой задачи по относительной шкале (например, присваивая задаче количество баллов из последовательности Фибоначчи: 1, 2, 3, 5, 8, 13, …). Важно отметить, что сложность задач нельзя оценивать количеством часов, которые планируется потратить на ее выполнение. Во-первых, люди плохо умеют это делать, а во-вторых, скрам-команды очень быстро наращивают свою производительность, и на третьем-четвертом спринте работают уже в несколько раз эффективнее, чем в начале работы.

Затем владелец продукта упорядочивает список, определяя, какие из задач

* имеют самое большое значение для хода работ над проектом;
* важнее всего для заказчика или будущего потребителя;
* принесут максимальный доход;
* проще всего выполнить.

Таким образом формируется бэклог проекта – список требований к функциональности продукта, упорядоченный по степени важности задач, подлежащих реализации.

**Планирование спринта.**Это совещание, на котором команда смотрит в верхнюю часть бэклога и прогнозирует количество заданий, которое возможно выполнить за этот спринт. Когда команда договорилась о цели предстоящего спринта и об определенном количестве заданий, которые нужно выполнить за этот спринт, добавлять новые уже нельзя. Планирование спринта – это еще одна возможность для команды удостовериться, что все точно понимают, как реализация каждого задания служит достижению конечной цели проекта.

**Скрам-доска и диаграмма выгорания задач.** Слаженной командной работе способствует прозрачность всех действий и процессов. Для этого команда заводит скрам-доску с колонками: «Нужно сделать», «В работе», «Сделано». Все задачи, которые команда выбрала для реализации в этом спринте, записываются на стикеры и приклеиваются в колонку «Нужно сделать». По мере выполнения задач команда перемещает стикеры из одной колонки в другую. Еще один способ сделать работу видимой – создать диаграмму выгорания задач. Каждый день скрам-мастер подсчитывает количество баллов за выполненные задачи и отражает оставшееся количество на графике. В идеале к концу спринта кривая должна прийти в ноль.

**Совещания на ходу.** Каждый день в одно и то же время не более чем на пятнадцать минут разработчики и скрам-мастер встречаются и дают ответы на три вопроса.

1. Что ты делал вчера, чтобы помочь команде завершить спринт?
2. Что ты будешь делать сегодня, чтобы помочь команде завершить спринт?
3. Какие препятствия встали на пути команды?

Суть таких встреч, которые действительно чаще всего проводятся стоя, в том, чтобы вся команда точно знала, на каком этапе находится каждое задание в текущем спринте. Все ли задачи будут выполнены в срок? Есть ли возможность помочь другим членам команды преодолеть препятствия? Никто не распределяет заданий сверху и не пишет отчетов руководству – команда самостоятельна и все решает сама. Скрам-мастер отвечает за устранение помех, мешающих команде продвигаться вперед.

**Обзор спринта.** Это открытая для всех заинтересованных лиц встреча, на которой команда рассказывает и показывает, что сделано за спринт. Демонстрировать команда должна только то, что полностью и окончательно готово. Это может быть и весь продукт, и его отдельная готовая функция.

**Ретроспективное собрание.** После того как команда передала клиенту результаты работы завершившегося спринта для получения обратной связи, все садятся за общий стол и обсуждают ряд вопросов. Что прошло хорошо? Что можно было сделать лучше? Что могло бы ускорить ход работ? Какое улучшение команда может внедрить в процесс немедленно? Ответы ищут всей командой, сохраняя фокус на решении проблемы, а не на поиске виноватых. К концу встречи разработчики и скрам-мастер договариваются о совершенствовании процесса, которое будет введено в действие в следующем спринте.

Далее цикл повторяется: команда приступает к планированию спринта.

На практике – для разных проектов и разных команд – процессы могут иметь свои особенности. Но знание основ в том виде, как их изложили авторы методологии, поможет вам быстрее освоиться, когда вы начнете работать в скрам-команде.

## **Анализ и сбор требований.**

Сбор требований — это один из самых важных этапов процесса создания любой информационной системы, будь то десктопное, веб или мобильное приложение или же просто доработка уже существующего решения. Прежде, чем начать собирать требования, необходимо выявить всех заинтересованных лиц (стейкхолдеров), которые будут пользоваться системой. Чем точнее будет этот список, тем полнее будут требования. Итак, для начала, рассмотрим кто же такие стейкхолдеры.

Стейкхолдерами могут быть любые физические лица и/или организации, которые активно участвуют в нашем проекте, и чьи интересы могут быть затронуты не только в процессе создания системы, но и непосредственно по завершению самого проекта. Ими могут быть менеджеры, начальники отделов, директора, любые сотрудники организации, которые будут хоть как-то взаимодействовать с готовым решением, и чьи требования (пожелания, идеи, потребности, проблемы) будем собирать.

Существует множество различных техник сбора требований, которые помогут лучше понять, что же хочет заказчик.  
Рассмотрим основные из них чуть подробнее:

*Анкетирование*

Данный способ подразумевает под собой составление листа-опросника (анкеты, брифа), который может содержать открытые (требуют от опрашиваемого сформулировать его ответ) и закрытые (требуют от опрашиваемого выбрать ответ из предложенных вариантов) вопросы.

Анкетирование используется для того, чтобы подтвердить или детализировать ранее известные требования, выбрать параметры для решений.

Самым известным примером анкетирования может быть “Бриф на разработку сайта” — анкета содержащая список основных требований и информацию о будущем сайте.  
Преимущества:

* Высокая скорость получения результатов.
* Сравнительно небольшие материальные затраты.

Недостатки:

* Данный метод не подходит для выявления неявных требований.
* При составлении опросника физически невозможно учесть все необходимые вопросы.

*Интервью*

Этот метод известен многим, своего рода беседа “по душам” с заинтересованным лицом, тет-а-тет.

Необходимо задавать открытые вопросы для получения информации и закрытые для того, чтобы подтвердить или опровергнуть конкретные варианты требований.

Данный способ применяется, в основном, для получения информации по какой-либо конкретной теме и/или для уточнения требований.

Многим может показаться этот способ достаточно легким, но это не так. Провести хорошее интервью достаточно сложно. Вы должны гибко реагировать на реакцию интервьюируемого и в случае необходимости изменять порядок заготовленных вопросов или их формулировку. Не забудьте включить диктофон во время интервью или вести заметки.  
Из плюсов:

* Возможность задавать вопросы в произвольной последовательности.
* Возможность использовать вспомогательный материал.
* Анализ невербальной реакции опрашиваемого человека, позволит сделать дополнительный вывод о достоверности его ответов.

Из минусов:

* Интервью отнимает достаточно много времени и сил.
* Дополнительной сложностью является получение одинаковых ответов от интервьюируемых.

*Автозапись*

Данный метод подразумевает под собой работу с записями, письмами (электронными письмами), а также с любым другим документом, автором которого является Заказчик или конечный пользователь (т.е. стейкхолдер).

В действительности это может быть и документ и наговоренная на диктофон последовательность действий или самая обычная салфетка, на которой заказчик набросал свои идеи / проблемы /хотелки, которые необходимо превратить в полноценные требования, согласовать и передать в разработку.

Примером такого метода может быть работа с концепцией или видением проекта (сам Заказчик любит называть это — «ТЗ»), которую он прислал вам на момент начала работ по аналитике.

Преимущество:

* Помогает лучше понять сложные процедуры или процессы.

Недостаток:

* Данный метод сильно зависит от опыта Заказчика, а также от его умения формулировать и выражать свои мысли.

*Изучение существующей документации*

Данная методика может быть использована при наличии в организации документации, которая может помочь в определении потребностей Заказчика. Примеры документации включают в себя: регламенты, описания процессов, структура организации, спецификации продукта, различные процедуры, стандарты и инструкции, шаблоны документов и т.д.

Выявленные требования являются основой для дальнейшего анализа и должны быть детализированы.  
Данная методика применима, например, при автоматизации устоявшихся в организации регламентированных бизнес процессов.

Плюс:

* Быстрое получение информации.

Минус:

* Данный способ не применим при наличии в компании только базовых документов (или при их полном отсутствии) или, если в компании Заказчика не поддерживается актуальность документации.

*Повторное использование спецификации*

Повторно использовать спецификации можно в том случае, если есть уже завершенные один или несколько подобных проектов.

Техническое задание, подготовленное на предыдущем проекте, может быть использовано для другого проекта с целью сократить продолжительность сбора, анализа и разработки требований, что позволит быстрее начать разработку.  
Например, ТЗ для интернет магазинов похожи друг на друга и содержат одинаковые требования.

В большинстве случаев только часть документации актуальна для нового проекта, поэтому потребуется тщательная проверка требований на соответствие текущим целям и задачам Заказчика.

Преимущество:

* Сокращение времени на разработку документации.

Недостатки:

* Высокая стоимость первого проекта.
* Излишняя детализация требований, может привести к их дорогостоящим изменениям в будущем.

*Представитель заказчика в компании разработчика*

Один из наиболее эффективных методов сбора требований, поскольку позволяет получать от представителя Заказчика своевременную оценку прогресса, корректности реализации, в короткие сроки получать обратную связь (фидбек) и дополнительную информацию для корректировки и разработки требований.

Метод часто применяется для сбора и управления требованиями при итерационной разработке, позволяет оперативно собирать, согласовывать и дорабатывать требования.

В дополнение можно сказать, что наличие представителя заказчика в компании разработчика является одним из главных правил Agile.

Преимущество:

* Быстрое получение обратной связи и информации от Заказчика.

Недостатки:

* Достаточно высокая цена для Заказчика.
* Затраты по времени на адаптацию сотрудника.

*Работа «в поле»*

Работа «в поле» состоит из наблюдения за деятельностью и процессами будущих пользователей системы, и в определении требований, основанных на этом наблюдении. Если говорить проще, то это наблюдение за тем, как работают пользователи, и документирование процесса, задач и результатов их деятельности.

Метод позволяет избежать проблем, связанных с трудностями стейкхолдеров в описании и выражении своих потребностей. В некоторых случаях процесс наблюдения может сопровождаться «интервьюированием» пользователей для уточнения особенностей и деталей их работы и задач. В процессе наблюдения можно так же выявить пути оптимизации бизнес-процессов Заказчика.

Преимущества:

* Позволяет наглядно увидеть проблему и разработать наиболее оптимальный вариант ее решения.
* Помогает наиболее точно собрать требования, наблюдая за работой сотрудников.

Недостатки:

* В процессе наблюдения могут быть упущены некоторые альтернативные сценарии бизнес-процесса.
* Трудно применим на секретных предприятиях или опасных (вредных) производствах.

*Обучение*

Обучение — это процесс, в котором Заказчик или любой другой человек из организации Заказчика, знающий процесс, обучает аналитика по принципу учитель — ученик.

Метод полезен, когда процессы и деятельность сотрудников Заказчика трудно описать с помощью других методов или Заказчик не может предоставить адекватное описание требований.

Обучение позволяет лучше понять сложные бизнес-процессы, а также преодолеть трудности, связанные с нехваткой абстрактного мышления и самовыражения у будущих пользователей системы.

Преимущество:

* Позволяет понять сложный бизнес-процесс, что позволяет предложить наилучшее решение.

Недостатки:

* Высокая стоимость и длительность.
* Метод неприменим на опасных (вредных) производствах.

*Мозговой штурм[[4]](#footnote-4)*

Мозговой штурм — наиболее часто используемый метод получения требований, которые связанны с новыми или плохо изученными направлениями деятельности организации Заказчика или функциями системы.

Он позволяет собрать множество идей от различных заинтересованных лиц (стейкхолдеров) в кратчайшие сроки и практически бесплатно.

Во время мозгового штурма участники «накидывают» любые идеи, касающиеся решения данной проблемы.С помощью этой методики можно проработать несколько различных вариантов решения заданной проблемы, а также разрешить конфликты требований.  
Плюсы:

* Позволяет генерировать множество (в том числе и нестандартных) вариантов решений, а также позволяет участникам развивать идеи друг друга.

Минусы:

* Участники мозгового штурма должны быть мотивированы на идеи.
* Трудно применим в распределенных командах.

*Совещание*

Совещание — встреча, ориентированная на обсуждение конкретных вопросов, которые были определены и озвучены участникам заранее.

На такие встречи привлекаются люди, которые придерживаются различных точек зрения по текущей проблеме и могут помочь описать требования, основываясь на взглядах с разных сторон. В процессе совещания уточняется общий список требований, выявляются скрытые требования и решаются конфликты требований.

Совещания являются одной из ключевых практик в Agile, т.к. в них участвуют все стороны, заинтересованные в развитии проекта и решении проблемы.  
Плюсы:

* Позволяет развить и детализировать требования, определить приоритеты.

Недостатки:

* Сложности в организации встречи, если команда географически разделена, могут возникнуть трудности с присутствием всех необходимых людей на совещании.
* Консенсус необязательно будет достигнут.

*Use case*

Use cases или варианты использования позволяют собрать и сформировать функциональные требования от лица участников [Диаграммы вариантов использования](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BF%D1%80%D0%B5%D1%86%D0%B5%D0%B4%D0%B5%D0%BD%D1%82%D0%BE%D0%B2) определяют границы решения и показывают связи с внешними системами и участниками.

Метод позволяет детализировать требования с точки зрения пользователей, а также помогает уточнить и систематизировать функционал, который требуется реализовать.  
Плюсы:

* Позволяет проработать все варианты развития сценария (основной и альтернативные сценарии)

Минусы:

* Метод не применим для сбора нефункциональных требований.

## **2.3. Основные концепции проектирования ПО**.

Информационная система является системой людей, информационных технологий и бизнес-процессов. Существуют следующие подходы к классификации информационных систем:

По степени автоматизации:

• ручные (неэлектронные);

• электронные;

− автоматические;

− автоматизированные.

По процессам обработки данных:

• поисковые (информационно-поисковые) – ввод и хранение данных;

• решающие – преобразование (переработка данных).

По направлению деятельности различают:

• производственные системы;

– автоматизированные системы управления производством;

– автоматизированные системы управления технологическими процессами;

– автоматизированные системы управления техническими средствами • административные системы (человеческих ресурсов);

• финансовые и учетные системы;

• системы маркетинга;

• информационные системы здравоохранения.

Технология – это набор правил, методик и инструментов, позволяющих наладить производственный процесс выпуска какого-либо продукта.

Информационная технология – процесс различных операций и действий над данными. Все процессы преобразования информации в информационной системе осуществляются с помощью информационных технологий. Электронные информационные системы и технологии реализуются в виде программного обеспечения. Ввиду этого важной частью процесса проектирования информационных систем являются процессы проектирования и реализации программного обеспечения.

Проектирование программного обеспечения — процесс создания проекта программного обеспечения, а также дисциплина, изучающая методы проектирования. Целью проектирования является определение 6 внутренних свойств программного обеспечения и детализации его внешних свойств, на основе выданных заказчиком и впоследствии проанализированных требований. В процессе создания программного обеспечения проектированию подлежит следующее:

1. высокоуровневая архитектура программного обеспечения;

2. низкоуровневая (внутренняя) архитектура программных компонентов;

3. пользовательский интерфейс;

4. сценарии взаимодействия пользователя с программным продуктом;

5. структуры данных; 6. модель предметной области;

7. алгоритмы.

На начальном этапе развития программной инженерии основными задачами проектирования были проектирование алгоритмов и структур данных. Помимо этого, проектировалась модульная структура программных приложений. С появлением объектно-ориентированной парадигмы программирования проектируются взаимодействия между объектами классов, появляются объектно-ориентированные шаблоны микроархитектуры (шаблоны проектирования) такие как обозреватель, синглентон, фасад, прокси и другие. Более сложной организацией программного обеспечения является программная система, состоящая из компонентов. В настоящее время первоочередной задачей проектирования является проектирования архитектуры программных систем – макроархитектуры.

Проектирование интерфейса пользователя включает не только графический дизайн, но проектирование взаимодействия пользователя и программного приложения (User Experience design, UX design), где учитывается количество переходов и схемы между окнами, среднее время на выполнение определенного действия и другие параметры. Проектирование интерфейса пользователя существенно отличается для различных типов клиентских устройств и вида программного приложения (веб-приложение, полноэкранное игровое программное приложение, корпоративное приложение). Подход к проектированию программного приложения, при котором наибольшее значение имеет качество человеко-машинного взаимодействие, именуется Look and Feed Driven Design [Chung C. Pro Objective-C Design Patternsfor iOS. - New York, USA: Appress, 2011.]. Данный подход имеет особую популярность для мобильных программных приложений, где эстетические предпочтения пользователя оказывают значительное влияние на решение о покупке и установке программного приложения. [1,5-6c].

## **2.4. Разработка программного обеспечения. Использование инструментов контроля версий.**

Важно различать автоматическое тестирование и тестирование, выполняемое вручную. Тестирование в ручном режиме проводит человек, который проверяет работу всех функций приложения вручную либо путем взаимодействия с программным обеспечением и API посредством соответствующего инструментария. Это очень затратный способ, поскольку кто-то должен настраивать среду и проводить тесты. Кроме того, необходимо учитывать человеческий фактор, так как тестировщик может допустить опечатку или пропустить какой-либо этап тестового скрипта.

Автоматические тесты, напротив, выполняются машиной, которая использует заранее написанный тестовый скрипт. Такие тесты могут значительно различаться по сложности — от проверки одного метода в классе до обеспечения условий, в которых выполнение последовательности сложных действий в пользовательском интерфейсе приводит к одинаковым результатам. Такой подход гораздо стабильнее и надежнее по сравнению с тестами, выполняемыми вручную, однако качество автоматического тестирования зависит от качества тестовых скриптов. Если вы только начинаете внедрять тестирование, рекомендуем прочитать наше учебное руководство по непрерывной интеграции, которое поможет создать первый комплект тестов. Ищете дополнительные инструменты для тестирования?

1. Модульные тесты

Модульные тесты работают на очень низком уровне, близко к исходному коду приложения. Они заключаются в тестировании отдельных методов и функций классов, компонентов или модулей, используемых в ПО. Модульные тесты, как правило, не требуют больших расходов на автоматизацию и могут выполняться сервером непрерывной интеграции очень быстро.

2. Интеграционные тесты

В ходе интеграционного тестирования проверяется, хорошо ли работают вместе различные модули и сервисы, используемые приложением. Например, можно протестировать взаимодействие с базой данных или убедиться, что микросервисы работают вместе так, как задумано. Этот вид тестирования является более затратным, поскольку для проведения тестов требуется запуск различных компонентов приложения.

3. Функциональные тесты

В функциональных тестах основное внимание уделяется бизнес-требованиям к приложению. Они проверяют только результат некоторого действия и не проверяют промежуточные состояния системы при выполнении этого действия.

Иногда возникает путаница между понятиями интеграционных и функциональных тестов, так как и те и другие требуют взаимодействия нескольких компонентов друг с другом. Разница в том, что интеграционный тест нужен просто чтобы убедиться, что вы можете отправлять запросы к базе данных, тогда как функциональный тест будет ожидать получения из базы данных определенного значения в соответствии с требованиями продукта.

4. Сквозные тесты

Сквозное тестирование копирует поведение пользователя при работе с ПО в контексте всего приложения. Оно обеспечивает контроль того, что различные схемы действий пользователя работают должным образом. Сценарии могут быть как очень простыми (загрузка веб-страницы или вход в систему), так и гораздо более сложными (проверка почтовых уведомлений, онлайн-платежей и т. д.).

Сквозные тесты очень полезны, но их выполнение обходится довольно дорого, к тому же, когда они автоматизированы, такие тесты тяжело обслуживать. Рекомендуется иметь в наличии несколько основных сквозных тестов и активнее полагаться на более низкие уровни тестирования (модульные и интеграционные тесты), чтобы получать возможность быстро выявлять критические изменения.

5. Приемочное тестирование

Приемочные тесты — это формальные тесты, которые проверяют, отвечает ли система требованиям бизнеса. При этом во время тестирования должно быть запущено само приложение, и основное внимание уделяется воспроизведению поведения пользователей. В ходе этого тестирования возможен даже замер производительности системы, и в случае несоответствия установленным требованиям внесенные изменения могут быть отклонены.

6. Тестирование производительности[[5]](#footnote-5)

В тестах производительности оценивается работа системы при определенной рабочей нагрузке. С помощью таких тестов можно оценить надежность, скорость, масштабируемость и отзывчивость приложения. Например, это может быть наблюдение за временем отклика при выполнении большого количества запросов или определение поведения системы при работе со значительными объемами данных. Этот вид тестирования позволяет определить, соответствует ли приложение требованиям к производительности, найти узкие места, оценить стабильность при пиковом трафике и многое другое.

7. Smoke-тестирование

Smoke-тесты — это базовые тесты, которые проверяют основные функциональные возможности приложения. Они должны выполняться быстро, поскольку цель таких тестов — убедиться, что основные возможности системы работают как запланировано.

Smoke-тесты полезно запускать сразу после создания новой сборки (для определения, можно ли запускать более ресурсоемкие тесты) или сразу после развертывания (чтобы убедиться, что приложение работает правильно в новой, только что развернутой среде).

*Тестирование* – это процесс выполнения программы с целью обнаружения ошибок.

Шаги процесса задаются тестами. Каждый тест определяет:

1. свой набор исходных данных и условий для запуска программы;
2. набор ожидаемых результатов и работы программы.

Другое название теста – тестовый вариант. Полную проверку программы гарантирует только исчерпывающее тестирование. Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. На практике исчерпывающее тестирование практически никогда не выполняется из-за ресурсных ограничений, прежде всего ограничений по времени.

Хорошим считается тестовый вариант с высокой вероятностью обнаружения еще нераскрытой ошибки. Успешным называется тест, который обнаруживает до сих пор не найденную ошибку. Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Тестирование обеспечивает:

1. обнаружение ошибок;
2. демонстрацию соответствия функций программы ее назначению;
3. демонстрацию реализации требований характеристикам программы;
4. отображение надежности, как индикатора качества программы.

Тестирование не может показать отсутствие дефектов, оно может показать только присутствие дефектов.

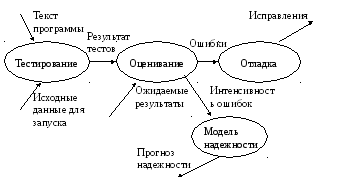


Рисунок 2.4.1 – Информационные потоки процесса тестирования

На входе процесса тестирования 3 потока: текст программы, исходные данные, ожидаемые результаты.

После выполнения тестов полученные результаты оцениваются. Это значит, что результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение и фиксируется ошибка, начинается отладка. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Процесс отладки непредсказуем по времени. На поиск места возникновения ошибки и исправление может потребоваться час, день или месяц. Неопределенность в отладке приводит к большим трудностям в планировании действий.

После сброса и оценивания результата тестирования начинается оценка качества и надежности программного обеспечения. Если регулярно встречаются серьезные ошибки, требующие значительных проектных изменений, то качество и надежность программного обеспечения становятся подозрительными и констатируется необходимость усиления тестирования.

С другой стороны, если функции программного обеспечения реализованы правильно, а обнаруживаемые ошибки легко исправляются, может быть сделан один из двух выводов: либо качество и надежность программного обеспечения полностью удовлетворительны, либо используемые тесты неспособны обнаружить серьезных ошибок. В конечном счете, если тесты не обнаруживают ошибок, появляются сомнения в том, что тестовые варианты достаточно продуманы и что в программном обеспечении нет скрытых ошибок.

Такие ошибки будут, в конечном счете, обнаруживаться пользователями и корректироваться разработчиками на этапе сопровождения, когда стоимость исправления возрастет в 100 раз по сравнению с этапом разработки.

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используются модели надежности программного обеспечения, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

*Существуют 2 основных принципа тестирования программ:*

1. функциональное тестирование (тестирование черного ящика);
2. структурное тестирование (тестирование белого ящика).

При тестировании черного ящика известны функции программы и исследуется работа каждой функции на всей области определения.

Местом приложения тестов черного ящика является интерфейс программного обеспечения.

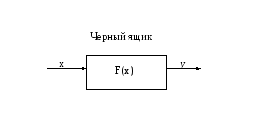


Рисунок 2.4.2 – Схема тестирования черного ящика.

*Тесты черного ящика демонстрируют:*

1. как выполняются функции программ;
2. как принимаются исходные данные;
3. как вырабатываются результаты;
4. как сохраняется целостность внешней информации.

При тестировании черного ящика рассматриваются системные характеристики программ, но игнорируется их внутренняя логическая структура.

Исчерпывающее тестирование черного ящика также, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется тестовых вариантов.

Необходимо также отметить, что тестирование черного ящика не реагирует на многие особенности программных ошибок.

При тестировании белого ящика известна внутренняя структура программы, а исследуются внутренние элементы программы и связи между ними.

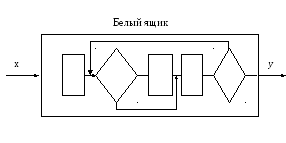


Рисунок 2.4.3 - Схема тестирования белого ящика

При тестировании белого ящика объектом тестирования является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. При этом обычно анализируются управляющие связи элементов, реже информационные.

Тестирование по принципу белого ящика характеризуется степенью, в которой тесты выполняют логику, т.е. исходный текст программы.

1. https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5\_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5 [↑](#footnote-ref-1)
2. http://www.4stud.info/software-construction-and-testing/lecture1. [↑](#footnote-ref-2)
3. https://studfile.net/preview/5615745/page:2/ [↑](#footnote-ref-3)
4. https://habr.com/ru/companies/simbirsoft/articles/307844/ [↑](#footnote-ref-4)
5. https://www.atlassian.com/ru/continuous-delivery/software-testing/types-of-software-testing [↑](#footnote-ref-5)