

Основные методологии разработки ПО

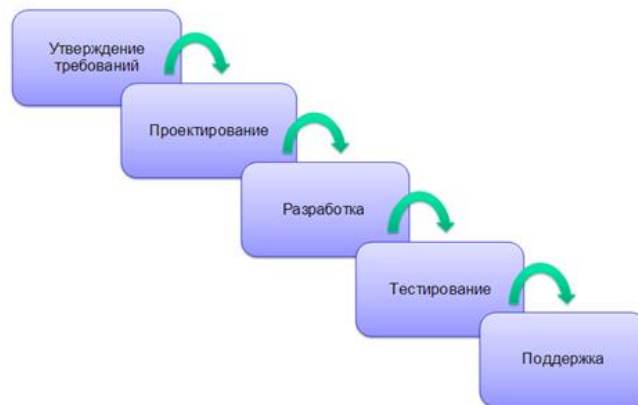
Методология разработки ПО – набор методов и критериев оценки, которые используются для постановки задачи, планирования, контроля и для достижения поставленной цели. Сам процесс разработки описывается моделью, которая определяет последовательность наиболее общих этапов и получаемых результатов.

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

Модель выбирают исходя из:

- Направления проекта
- Масштаба проекта
- Бюджета
- Сроков реализации конечного продукта

Каскадная модель (Waterfall model)



Самой старой и известной моделью построения многоуровневого процесса разработки является каскадная (или попросту водопадная) модель: в ней каждый этап разработки, соответствующий стадии жизненного цикла ПО, продолжает предыдущий. То есть для того, чтобы перейти на новый этап, мы полностью должны завершить текущий.

Примеры использования: строительство, медицина, работа с государственными контрактами, промышленностью и подобных фундаментальных целей разрабатывается при помощи той или иной модификации «водопада». Cisco (безопасность)

Особенности:

- Разработка проходит быстро.
- Стоимость и сроки заранее определены.
- Хороший результат только в проектах с четко и заранее определенными требованиями.
- Нет возможности сделать шаг назад.

Сильные стороны каскадной модели разработки ПО	Слабые стороны каскадной модели разработки ПО
<ul style="list-style-type: none">• предельная детализация каждого шага работы, сопровождающаяся документированием	<ul style="list-style-type: none">• затраты времени на ведение подробной документации, которая, к тому же, может быть не всегда понятной заказчику, и вызывать у него вопросы

- требования максимально внятно и четко изложены, не могут противоречить друг другу или меняться в середине работы
- необходимы квалифицированные бизнес-аналитики, способные сформулировать приемлемое для продуктивной работы ТЗ
- отсутствует возможность для маневра, если в процессе разработки выяснилось, что продукт не отвечает требованиям рынка
- возможность заранее знать, сколько времени и денег будет потрачено на проект
- затраты времени и денег достаточно высоки
- легкость понимания методики как таковой даже для не самых опытных разработчиков
- высокая вероятность выявления критических проблем уже на завершающем этапе разработки, причем их устранение на этапе готового продукта обходится чрезмерно дорого.
- **легкость контроля** и, при необходимости, передачи проекта другой команде, благодаря строгой системе отчетности.

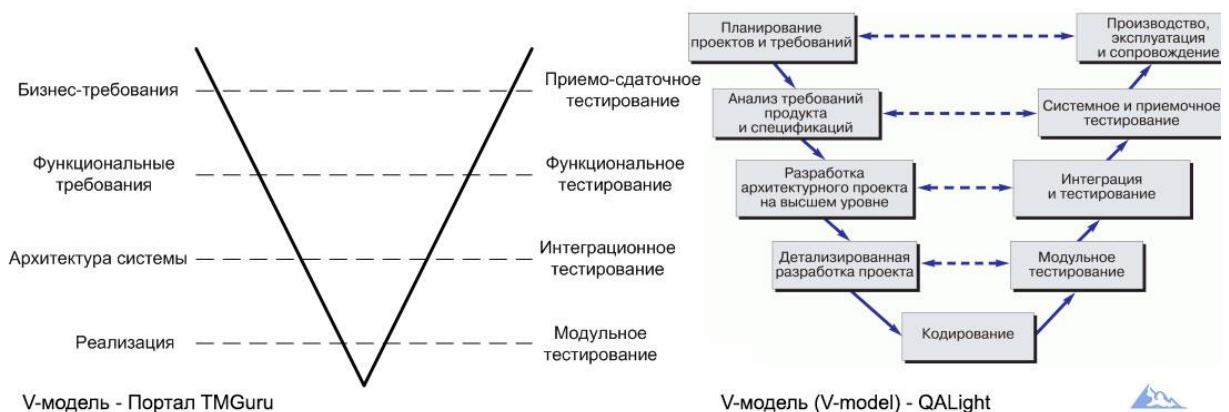
Применяется в следующих случаях:

1. заказчик участвует в проекте только на первом этапе и принимает готовый продукт;
2. изменять требования к продукту не планируется;
3. проект отличается высокой сложностью, длительностью и дороговизной;
4. основной приоритет — качество, даже в ущерб времени;
5. отсутствие команды разработчиков экстра-класса;
6. допускается возможность выполнения проекта на аутсорсе.

V-модель (V-model)

Концепция V-образной модели была разработана Германией и США в конце 1980-х годов независимо друг от друга.

V-модель используется для управления процессом разработки программного обеспечения для немецкой федеральной администрации. Сейчас она является стандартом для немецких правительственных и оборонных проектов, а также для производителей ПО в Германии.



V-модель – это улучшенная версия классической каскадной модели. Здесь на каждом

этапе происходит контроль текущего процесса, для того чтобы убедиться в возможности перехода на следующий уровень. В этой модели тестирование начинается еще со стадии написания требований, причем для каждого последующего этапа предусмотрен свой уровень тестового покрытия. V-образная модель применима к системам, которым особенно важно бесперебойное функционирование. Например, прикладные программы в клиниках для наблюдения за пациентами, интегрированное ПО для механизмов управления аварийными подушками безопасности в транспортных средствах и так далее.

Особенности:

- Направлена на тщательную проверку и тестирование продукта с ранних стадий проектирования.
- Стадия тестирования проводится одновременно с соответствующей стадией разработки.
- Количество ошибок в архитектуре ПО сводится к минимуму.
- Если при разработке архитектуры была допущена ошибка, то вернуться и исправить её будет стоить дорого, как и в «водопаде».

Итеративная модель (Iterative model)

Вместо одной продолжительной последовательности действий здесь весь жизненный цикл продукта разбит на ряд отдельных мини-циклов.

Планирование – Реализация – Проверка – Корректировка

В каждой из итераций происходит разработка отдельного компонента системы, после чего этот компонент добавляется к уже ранее разработанному функционалу.

Ключ к успешному использованию этой модели – строгая валидация требований и тщательная верификация разрабатываемой функциональности в каждой из итераций.

Преимущества:

- Гибкость в принятии новых требований или изменений.
- Возможность адаптации процесса на основе уроков, извлеченных из предыдущих итераций.
- Более короткие сроки вывода продукта на рынок.

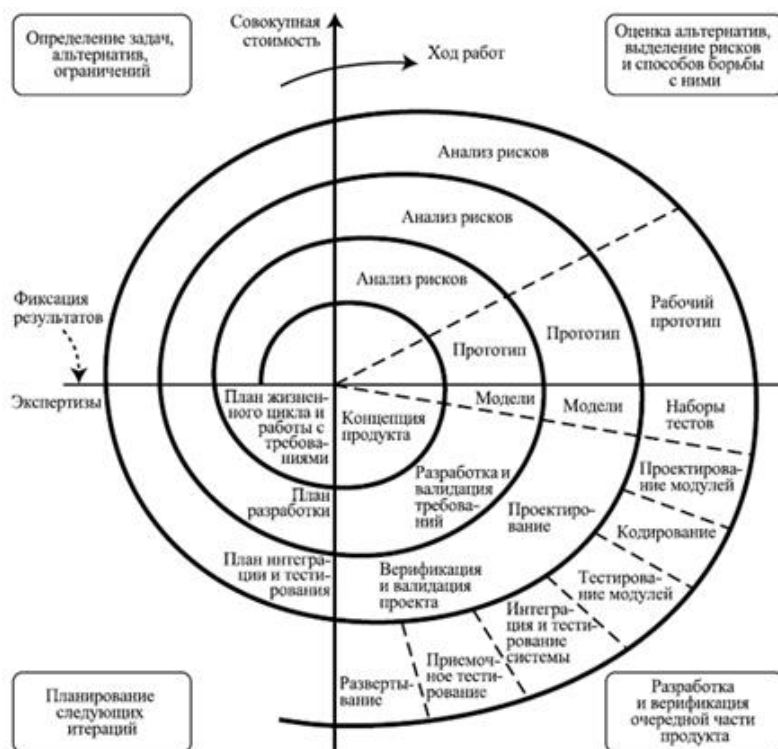
Недостатки:

- Стоимость продукта неизвестна
- Могут возникнуть проблемы с архитектурой системы, поскольку требования для всего жизненного цикла программы не собираются.

Итеративная модель является ключевым элементом так называемых «гибких» (Agile) подходов к разработке программного обеспечения

Спиральная модель (Spiral model)

Спиральная модель представляет шаблон процесса разработки ПО, который сочетает идеи итеративной и каскадной моделей. Суть ее в том, что весь процесс создания конечного продукта представлен в виде условной плоскости, разбитой на 4 сектора, каждый из которых представляет отдельные этапы его разработки: определение целей, оценка рисков, разработка и тестирование, планирование новой итерации.



Преимущества:

- Анализ рисков и управление рисками на каждом этапе.
- Подходит для больших проектов.
- Возможность изменения в требованиях на поздних этапах.
- Заказчик может наблюдать за развитием продукта на ранней стадии разработки.

Недостатки:

- Спиральная модель намного сложнее других моделей SDLC.
- Не подходит для небольших проектов, так как она дорогая.
- Успешное завершение проекта зависит от анализа рисков.
- Количество этапов неизвестно в начале проекта

[GanttPRO](#) — приложение для удобного управления проектами и задачами.

Когда использовать спиральную модель:

- когда важен анализ рисков и затрат;
- крупные долгосрочные проекты с отсутствием четких требований или вероятностью их динамического изменения;
- при разработке новой линейки продуктов.

Методология Agile – это набор практик, целью которых является оперативная реакция на изменения в ходе рабочего процесса. Такие подходы помогают командам быстро реагировать на обратную связь от клиентов и заказчиков, тем самым постоянно улучшая производимый продукт.

Ценности Agile

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

Суть AGILE МАНИФЕСТА:

- Вся работа над проектом разделяется на короткие циклы (итерации) и ведется поэтапно;
- В конце каждой итерации заказчик получает готовый минимально работающий продукт или его часть, которую уже можно использовать;
- В течение всего рабочего процесса команда сотрудничает с заказчиком;
- Любые изменения в проекте приветствуются и быстро интегрируются в работу.

Проще говоря, Agile — это своеобразная философия, но как ее придерживаться, зависит от конкретного случая. Есть 2 основных фреймворка, которые основываются на базовых принципах Agile: Scrum и Kanban.

Скрам



«Скрам — это фреймворк управления, согласно которому одна или несколько кроссфункциональных самоорганизованных команд создают продукт инкрементами, то есть поэтапно».

В Скраме есть система **ролей, встреч, правил и артефактов**. В этой модели за создание и адаптацию рабочих процессов отвечают команды.

Беклог продукта (product backlog) – это упорядоченный (приоритеты) набор задач, которые заинтересованные люди хотят получить от продукта + которые команда берется завершить до конца спринта. Этот список содержит краткие описания всех возможностей продукта.

Беклог спринта – набор элементов из беклога продукта для выполнения в текущем спринте.

Burndown chart является основным средством для отслеживания выполненных задач в спринте или во всем проекте.

В Скраме используются **итерации** фиксированной длины, называемые Спринтами. Они обычно занимают **1-2 недели (до 1 месяца)**. Скрам команды стремятся создавать готовый к поставке (качественно протестированный) Инкремент продукта в каждой итерации.

В Скраме три роли, которые вместе образуют Скрам команду:

- Владелец Продукта
- Команда разработки
- Скрам-Мастер

Суть данного фреймворка:

- Работа делится на спринты длительностью 1-3 недели.
- Перед началом спринта команда сама формирует список задач на итерацию.
- Во время каждого спринта создается продукт или услуга, которые можно продемонстрировать клиенту.
- Митинги, помогающие в процессе работы: Ежедневный мит, Планировка, Ретроспектива и Sprint Review.
- После выполнения спринта проводится ретроспектива. Это митинг, цель которого получить фидбэк от каждого участника команды, выявить текущие успехи и проблемы, то есть оценка и анализ проделанной работы.
- Каждый последующий этап будет наращивать функционал проекта, пока все функции не будут реализованы. Scrum подходит для объемных проектов.

ЧАСТЫЕ ПРОБЛЕМЫ ПЛАНИРОВАНИЯ:

- Владелец продукта сам определяет и решает, какая работа будет завершена.
- Беклог продукта не актуален, не приоритезирован или не готов к обсуждению.
- В конце планирования все слишком детализировано и вся работа уже распределена по исполнителям (эту проблему трудно преодолеть).
- Никто не понимает, что означает статус "Готово".
- Встреча слишком длинная.
- Встреча не включает участников в процесс.
- Некоторым людям сложно проявляться.
- неподходящая среда, команда не чувствует поддержки или безопасности.
- Нет доверия или уважения с обеих сторон.
- Команда не понимает, для чего нужна эта встреча.

Capacity(Ёмкость):

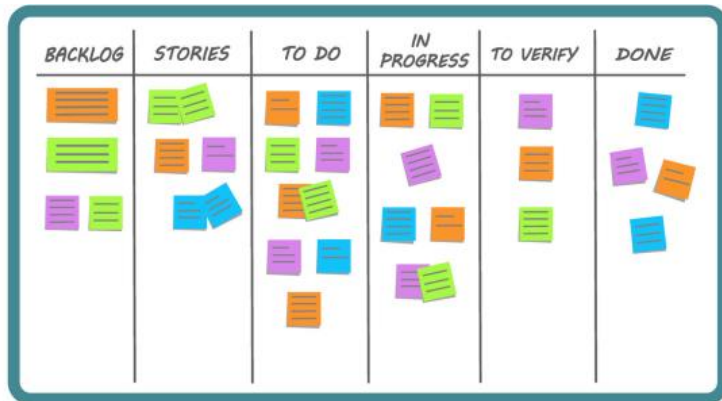
- Capacity прогноз - количество идеальных часов, доступное в следующем спринте.
- Понимание, сколько часов у нас есть на работу: на написание кода, тестирование, т.д.
- Как правило, участник проекта работает не более пяти часов в день.
- Эффективное распределение задач.
- Нет смысла планировать задачи на тех, кто будет в отпуске или занят другими активностями.
- Мало пользы принесет технический анализ задачи, выполненный участником проекта, который в следующем спринте будет отсутствовать.
- Аккуратное и точное планирование.
- Мы оцениваем задачи в часах и берем в спринт столько, сколько соответствует нашей capacity.

Канбан

Канбан — это метод улучшения процессов разработки и часть agile-философии. В его основе — «Манифест гибкой разработки программного обеспечения».

Цель Kanban: — получать готовый качественный продукт вовремя.

Канбан начинается с визуализации, чтобы процессы были на виду у команды. Для этого используют специальную доску и набор карточек или стикеров.



Доски Kanban:

Доска — это обязательный элемент для гибкой методологии. Она есть в Scrum, есть и в Kanban. Каждый член команды получает к ней доступ в любое время и видит, на каком этапе находится задача.

Система Канбан основана на принципах

- Визуализация. Основа Канбан – визуальная доска, на которой представлены этапы выполнения текущей задачи: «запланировано» / «выполняется» / «сделано».
- Разделение задач. Четкое понимание целей упрощает процесс.
- Фокусировка на работе. Невыполненные задачи требуют особого внимания. Если процесс затягивается, нужно подключать дополнительных сотрудников и перераспределять ресурсы.

Если в Scrum цель команды – закончить спринт, то в Kanban – задачу.

Основные отличия:

- В Канбане нет спринтов. Задачи и приоритеты можно менять во время работы, тогда как в Scrum – только к началу следующего спринта.
- В Канбане нет таких ролей, как у Scrum. Может быть только команда и Тех Лид.
- В Scrum митинги обязательны, а в Канбане - нет.

Принципы тестирования

1. Тестирование демонстрирует наличие дефектов, а не их отсутствие

Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программном обеспечении, но, даже если дефекты не были обнаружены, тестирование не доказывает его корректности.

2. Исчерпывающее тестирование недостижимо

Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев. Вместо попытки исчерпывающего тестирования должны использоваться анализ рисков, методы тестирования и расстановка приоритетов, чтобы сосредоточить усилия по тестированию.

3. Раннее тестирование сохраняет время и деньги

Для нахождения дефектов на ранних стадиях, как статические, так и динамические активности по тестированию должны быть начаты как можно раньше в жизненном цикле разработки программного обеспечения. Раннее тестирование иногда называют «сдвигом влево». Тестирование на ранних этапах жизненного цикла разработки программного обеспечения помогает сократить или исключить дорогостоящие изменения.

4. Кластеризация дефектов

Обычно небольшое количество модулей содержит большинство дефектов, обнаруженных во время тестирования перед выпуском, или отвечает за большинство эксплуатационных отказов. Это применение принципа Парето для тестирования ПО: около 80% проблем возникает в 20% модулей. Предсказанные кластеры дефектов и фактические наблюдаемые кластеры дефектов в ходе тестирования или эксплуатации являются важными входными данными для анализа риска, используемого для сосредоточения усилий по тестированию (как указано в принципе 2).

5. Парадокс пестицида

Если одни и те же тесты будут выполняться снова и снова, в конечном счете эти тесты больше не будут находить новых дефектов. Для обнаружения новых дефектов может потребоваться изменение существующих тестов и тестовых данных, а также написание новых тестов. (Тесты больше не эффективны при обнаружении дефектов, так же как пестициды через некоторое время больше не эффективны при борьбе с вредителями). В некоторых случаях, таких как автоматизированное регрессионное тестирование, парадокс пестицидов имеет положительный результат, который является относительно низким числом регрессионных дефектов.

Вот несколько рекомендаций на этот счет:

1. Следите за изменениями в продукте и предугадывайте их возможные последствия
2. Откажитесь от неэффективных и устаревших тестов
3. Постоянно меняйте тестовые данные (+при изменениях в функционале)
4. Используйте неформальные средства тестирования (прибегнуть к исследовательскому виду тестирования)

6. Тестирование зависит от контекста

Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение управления производством, в котором критически важна безопасность, тестируется иначе, чем мобильное приложение электронной коммерции.

7. Заблуждение об отсутствии ошибок

Некоторые организации ожидают, что тестировщики смогут выполнить все возможные тесты и найти все возможные дефекты, но принципы 2 и 1, соответственно, говорят нам, что это невозможно. Кроме того, ошибочно ожидать, что простое нахождение и исправление большого числа дефектов обеспечит успех системе. Например, тщательное тестирование всех указанных требований и исправление всех обнаруженных дефектов может привести к созданию системы, которая будет трудной в использовании, не будет соответствовать потребностям и ожиданиям пользователей или будет хуже по сравнению с другими конкурирующими системами.

- тестирование должно производиться независимыми специалистами;
- тестируйте как позитивные, так и негативные сценарии;
- не допускайте изменений в ПО в процессе тестирования;
- указывайте ожидаемый результат выполнения тестов.

Работа с тестовой документацией и ее виды

Тестовая документация бывает двух видов: **внешняя** и **внутренняя**. И та, и другая – инструмент, облегчающий жизнь проектной команде. Не более и не менее.

Внешняя документация:

- Замечание (по поводу процессов)
- Баг-репорт
- Запрос на изменение (улучшение функционала продукта: оптимизация расхода заряда, памяти, убрать/добавить фичу, изменить баланс)
- Отчет о тестировании (тест репорт)

Внутренняя документация:

- Тест-план (план тестирования)
- Тестовый сценарий
- Тестовый комплект
- Чек-лист (лист проверок)
- Тестовый случай (тест-кейс)

• **Тест-план.** Описывает весь объем работ по тестированию: описания объекта тестирования, стратегии, критериев начала (готова документация, готов билд для теста, готово нужное оборудование/окружение/доп.софт, зашли деньги на счет компании, когда подписали контракт, готова команда, и окончания тестирования (закрытие проекта, завершение работы с заказчиком, его банкротство, выдержка определенного периода без открытия новых багов major+, закончилось время на тест/итерация), необходимое оборудование и знания, оценки рисков с вариантами их разрешения.

Тест-план призван ответить на следующие вопросы:

- Что НАДО тестировать?
 - Что БУДЕМ тестировать? (Тест-Аналитик).
 - КАК будем тестировать? (Тест-Дизайнер).
 - На каких уровнях будем проводить тестирование?
 - Какие виды тестирования применим?
 - Каким образом будем тестировать – руками или автотестами?
 - КОГДА будем тестировать? Оценка трудозатрат и сроков.
 - Какие РИСКИ возможны? Какие затраты времени, средств и труда они могут повлечь. Степень их влияния на исход проекта, прописать мероприятия по нейтрализации последствий срабатывания рисков.
- платформы, на которых будет проводиться тестирование;
 - какой функционал тестим и что не тестим;
 - дату начала и завершения задачи;
 - пожелания Заказчика к тестированию (какому функционалу уделить больше внимания, где чаще возникают проблемы, какие устройства использовать);
 - наличие и возможность предоставления проектной и тестовой документации;
 - наличие читов и/или тестовых аккаунтов для ускорения выполнения проверок (например, тестирования покупок или прохождения игры);
 - место заведения баг-репортов (например, JIRA, Trello, Google Sheets, uTrack и т.п.);

- формат и частоту предоставления отчетов (ежедневно, еженедельно, по завершению задачи);
- **Тестовая стратегия.** Определяет то, как тестируем ПО. Это набор идей, которые направляют процесс тестирования.
- **Чек-лист.** Список, содержащий ряд необходимых проверок. Статусы: Passed, Failed, Blocked, In progress, Skipped, Not run.

Преимущества использования чек-листов:

- улучшается представление о системе в целом, виден статус её готовности;
- виден объём проделанной и предстоящей работы по тестированию;
- легче не повторяться в проверках и не упустить ничего важного в процессе тестирования.

Чтобы составить работающий чек-лист, обратите внимание на эти рекомендации:

- Один пункт = одна проверка. Минимальная полная операция проводимая тестировщиком при проверке — это один пункт чек-листа.
- При составлении чек-листа нужно опираться на требования, чтобы не тестировать то, что не существенно.
- Давайте пунктам чек-листа названия по форме, общей для всех членов команды, чтобы работа с чек-листом не вызывала неоднозначных толкований. Можно договориться использовать во всех пунктах только глаголы в инфинитиве или существительные: «проверить»/ «добавить»/ «отправить» либо «проверка»/«отправка»/«добавление».
- Детализируйте чек-лист в зависимости от задачи.
- Объединяйте чек-листы в матрицы, где можно отразить не только сами проверки, но и условия проверки (платформа, версия продукта, сотрудник и т.п.) и статус проверки. Матрицы — это компромисс между чек-листами и тест-кейсами. Их легче поддерживать, чем тест-кейсы, так как в такой таблице отсутствуют шаги (steps). В них одна строка = одна проверка.

Преимущества:

- чек-лист легко читается;
- по чек-листу быстро тестировать: в тест-кейсе нужно отмечать статус каждого шага, в то время как в чек-листе достаточно одной строчки;
- чек-лист — источник результатов для отчёта: можно быстро посчитать сколько проверок выполнено, в каком они статусе, узнать количество открытых репортов;
- в любой момент можно узнать статус — всегда есть то, что нужно проверить в первую очередь, можно упорядочить пункты чек-листа или изменить порядок, когда это требуется.

Недостатки:

- неопределенность тестового набора: каждый тестировщик выполняет пункт чек-листа по-своему;
- неопределенность тестовых данных;
- недостаточность детализации;

- сложнее обучить начинающих сотрудников: пункты чек-листа чаще абстрагируются от конкретных элементов интерфейса и описывают то, что нужно сделать;
- чек-лист менее эффективен для начинающих тестировщиков, лучше использовать тест-кейсы.

• **Тест-кейс.** Описывает наши тесты. Говорит, как их выполнить, при каких условиях и что должно получиться после выполнения шагов, которые заложены в тест-кейсе.

Позитивные/негативные

Высокоуровневые/низкоуровневые

1. Залежностей від інших тест-кейсів. Пов'язаний тест-кейс завжди може бути видалений через непотрібність або він може бути змінений, в цьому випадку стане незрозуміло як виконати тест-кейс, в якому є посилання. Також, через залежність тест-кейсів, може виникнути відчуття, що продукт, що тестується, вже призведе до потрібного стану завдяки виконанню пов'язаних тест-кейсів.
2. Нечітких формулювань. Якщо опис теми буде нечітким, то це, як мінімум, ускладнить проходження тест-кейсу та сприйняття тест-сюту в цілому. Під час створення теми тест-кейсу потрібно пам'ятати:
 - a. те, що очевидно для вас зараз, може стати абсолютно незрозумілим через пару місяців. Так, скорочення з нерозшифрованими аббревіатурами, зрозумілими вам зараз, можуть згодом стати китайською грамотою для вас самих, так що простіше написати тест-кейс заново, ніж пробиратися через нетрі необачно зроблених скорочень, як у темі, і усередині кейса.
 - b. тест-кейс, тема якого не може бути зрозуміла і згодом виконана ніким, крім його автора, повинен бути публічно знищений. Обґрунтування просте: автор кейсу може захворіти, піти у відпустку, піти з компанії тощо. Будь-який тест-кейс має створюватися з думкою про колегу, який одного разу візьме його до рук.
 - c. Зайвої деталізації. Надмірна інформація лише ускладнює розуміння вкладеного в тему сенсу. Тестувальник при виконанні тест-кейсу спочатку прочитає тему, усвідомлює її, вже приблизно уявить, що йому зараз потрібно буде виконати, і тільки після цього перейде до кроків.

• **Тест-сют.** Комплект тест-кейсов для исследуемого компонента или системы.

• **Баг-репорт.** Содержит полное описание дефекта.

• **Отчёт о результатах тестирования.** Здесь обобщаются все результаты работ по тестированию.

Отчёт о результатах тестирования включает следующие разделы:

- Краткое описание.
- Команда тестировщиков.
- Описание процесса тестирования.
- Расписание.
- Статистика по новым дефектам.
- Список новых дефектов.
- Статистика по всем дефектам.
- Рекомендации.

- Приложения. Фактические данные (как правило, значения метрик и графическое представление их изменения во времени).
- **Use Case.** Сценарий взаимодействия пользователя с программным продуктом для достижения конкретной цели.

Тестирование документации — это начальная стадия процесса тестирования, которая выступает как система раннего оповещения об ошибках. Процесс тестирования так или иначе начинается с документации и требований. Тестирование документации предполагает начало тестирования еще до разработки продукта. Тестировщик может указать на логические ошибки в постановке задачи, несоответствия в требованиях, а также составить чек-лист, список проверок по предоставленному требованию.

В процесс тестирования документации важно вовлекать различных специалистов: тестировщики, проджект-менеджеры, бизнес-аналитики, разработчики.

- На основании чего создается тестовая документация (спецификации, ТЗ)
- Этапы написания тестовой документации;
- Критерии переходов между этапами написания;
- Какие критерии начала и окончания написания документации.

Критерии написания документации:

- **Полнота и соответствие действительности.**
- **Навигация.** И не просто навигация, а удобная навигация. У пользователя никогда не должно возникать проблем с поиском необходимой ему информации.
- От пункта выше, вытекает **структурированность документации**. Все документы должны находиться в полном порядке, по разделам.
- **Инструкции должны присутствовать везде.** Даже при выполнении абсолютно одинаковых манипуляций с программой – необходимо пошаговое описание действий во всех случаях. Это может быть, как и прямое повторение инструкций, так и ссылка на уже существующие.
- **Термины и их значение.** В любой документации может использоваться масса терминов, аббревиатур и сокращений.
- **Доступность пользователю.** Документация должна быть максимально понятной для любой целевой аудитории.
- Если документация создана и для иностранных пользователей – необходимо **привлечение специалистов** данного **лингвистического сектора**, вплоть до носителей языка.

Существует еще много требований к составлению и тестированию документации. Но главное правило, которое поможет нам – это умение ставить себя на место пользователя, попавшего в определенную проблемную ситуацию.

Интернационализация и локализация

Как можно понять из терминов «**интернационализация и локализация**» – это процесс придания продукту свойств определенной народности, местности, расположения.

Локализация – процесс адаптации программного продукта к языку и культуре клиента. Данный процесс адаптации включает в себя:

- Перевод пользовательского интерфейса.

- Перевод документации.
- Контроль формата даты и времени.
- Внимание к денежным единицам/валютам.
- Внимание к правовым требованиям.
- Раскладка клавиатуры пользователя и горячие клавиши.
- Контроль символики и цветов.
- Толкование текста, символов, знаков.
- И прочие подобные аспекты.

Тестирование локализации – это проверка содержимого приложения или сайта на соответствие лингвистическим, культурным требованиям, а также специфике конкретной страны или региона.

Тестирование интернационализации – это процесс проверки тестируемого приложения на работоспособность в разных регионах и культурах. Основная цель интернационализации — проверить, может ли код обрабатывать всю международную поддержку, не нарушая функциональность, которая может привести к потере данных или проблемам целостности данных.

Интернационализация – более обобщенное понятие, подразумевающее проектирование и реализацию программного продукта или документации таким образом, который максимально упростит локализацию приложения.

Интернационализация включает в себя:

- Создание продукта с учетом возможности кодировки Unicode (стандарт кодирования, поддерживающий практически все языки мира).
- Создание в приложении возможности поддержки элементов, которые невозможно локализовать обычным образом (вертикальный текст азиатских стран, чтение с права на лево арабских стран и т.д.). НАПРАВЛЕНИЕ ТЕКСТА
- Возможность загрузки/выделения локализованных элементов кода в будущем при желании пользователя

Техники тест-дизайна

Тест-дизайн – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.

- | | |
|-------------------------------------------------------|------------|
| • Эквивалентное Разделение (Equivalence Partitioning) | EP |
| • Анализ Граничных Значений (Boundary Value Analysis) | BVA |
| • Метод попарного тестирования (Pairwise Testing) | PT |
| • Диаграмма переходных состояний | |
| • Таблица принятия решений (Decision Table) | DT |
| • Предугадывание ошибки (Error Guessing) | EG |
| • Причина / Следствие (Cause/Effect) | CE |
| • Матрица покрытия требований (Traceability matrix) | |

Матрица соответствия требований (Requirements Traceability Matrix) – это двумерная таблица, содержащая соответствие функциональных требований (**functional requirements**) продукта и подготовленных тестовых сценариев (**test cases**).

В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки. Матрица обычно хранится в виде электронной таблицы.

Матрица соответствия требований используется **QA**-инженерами для валидации покрытия требований по продукту тестами. Цель «Traceability Matrix» состоит в том, чтобы выяснить:

- какие требования «покрыты» тестами, а какие нет.
- избыточность тестов (одно функциональное требование покрыто большим количеством тестов).

Данный тестовый артефакт является неотъемлемой частью тестирования.

План разработки тест-кейсов:

Анализ требований;

Определение набора тестовых данных на основании **EP, BVA, EG, PT, DT**;

Разработка шаблона теста на основании **CE**

Написание тест кейсов на основании первоначальных требований, тестовых данных и шагов теста.

Примеры на техники Тест-дизайна:

Класс эквивалентности (Equivalence class) – это набор входных (или выходных) данных ПО, которые обрабатываются программой по одному алгоритму или приводят к одному результату.

Далее, каждый класс эквивалентности можем разделить на дополнительные классы и т.д. до того момента, пока проверки не будут приводить к точечным и конкретным результатам тестирования. Области эквивалентности могут быть как для правильных, или позитивных, так и неправильных, или негативных, значений.

Эквивалентное разделение, алгоритм использования техники:

1. Необходимо определить класс эквивалентности. Это главный шаг техники. От него во многом зависит эффективность её применения.
2. Затем нужно выбрать одного представителя от каждого класса. На этом шаге из каждого эквивалентного набора тестов мы выбираем один тест.
3. Нужно выполнить тесты. На этом шаге мы выполняем тесты от каждого класса эквивалентности.

Плюсы и минусы техники анализа классов эквивалентности:

- К плюсам можно отнести заметное сокращение времени и улучшение структурированности тестирования.
- К минусам можно отнести то, что, при неправильном использовании техники, мы рискуем потерять баги.



Рассмотрим пример:

Система скоринга рассчитывает процентную ставку по кредиту для клиента исходя из его возраста, который вводится в форму:

- От 18 до 25 лет – 18 %
- От 25 до 45 лет – 16 %
- Свыше 45 лет – 20 %

Мы определяем 2 основных класса – это позитивные и негативные сценарии.

Позитивными сценариями будут все значения, которые приводят к получению результата, негативными сценариями – значения, результаты которых не описаны, как ожидаемый результат.

Далее мы делим класс позитивных сценариев 3 класса вводимых значений **18-24, 25-44 и 45 +**

В классе негативных сценариев мы формируем значения, исходя из необходимости проверки отказов программы, поэтому мы имеем **0, 1-17, отрицательные значения, ввод символов и т.д.**

Результатом данного разбиения будет значение или диапазон значений, в котором нам необходимо выполнить всего одну проверку с любым значением из диапазона данных. Могут возникнуть такие ситуации, как одно значение в диапазоне. Это тоже отдельный класс эквивалентности и тоже требует проверки.

Итого мы имеем: Позитивные проверки: Ввод значений: 19, 30, 48 (значения могут быть любыми из данного диапазона класса)

Граничные значения – техника тест-дизайна, которая дополняет классы эквивалентности дополнительными проверками на границе изменения условий.

Может быть применима, только если классы состоят из упорядоченных числовых значений. Максимальное и минимальное значение класса являются его границами.

Примерный алгоритм использования техники анализа граничных значений:

- Во-первых, нужно выделить классы эквивалентности. Опять же, это очень важный шаг и от правильности разбиения на классы эквивалентности зависит эффективность тестов граничных значений.
- Далее нужно определить граничные значения этих классов.
- Нам нужно понять, к какому классу будет относиться каждая граница.

- Для каждой границы нам нужно провести тесты по проверке значения до границы, на границе, и сразу после границы.

Плюсы и минусы техники анализа граничных значений:

- Эта техника добавляет в технику анализа классов эквивалентности ориентированность на конкретный тип ошибок.
- То есть, техника анализа классов эквивалентности просто говорит нам о том, что нужно разбить все тесты на классы и провести тестирование всех классов. А техника граничных значений ориентирована на обнаружение конкретной проблемы – возникновения ошибок на границах классов эквивалентности.
- Но, как и для техники анализа классов эквивалентности, эффективность техники анализа граничных значений зависит от правильности ее использования. Мы должны приложить усилия, чтобы правильно определить классы эквивалентности и их границы. Если мы отнесемся к этому поверхностно, то рискуем пропустить ошибки.

Вернемся к нашему примеру ранее.

Система скорринга рассчитывает процентную ставку по кредиту для клиента исходя из его возраста, который вводится в форму:

- От 18 до 25 лет – 18%
- От 25 до 45 лет – 16 %
- Свыше 45 лет – 20%

Что же здесь будет границей?

Что определить граничные значения нужно нечто иное. А именно, определить, какие значения являются начальным и конечным для нашего класса. И самое важное!!! БОльшая часть дефектов находится тестировщиками именно на стыке значений, которые меняют условия работы программы.

Поэтому, помимо граничного значения мы используем для тестирования дополнительно 2 значения, значение перед границей и значение после границы.

В итоге мы имеем:

Границы наших классов: **17, 18, 19, 24, 25, 26, 44, 45, 46** и **max**.

Также, у нас есть негативный класс, это от **0** до **18**. Поэтому тут мы тоже должны использовать для тестирования граничные значения: **-1, 0, 1, 17, 18**

Далее исключаем повторяющиеся значения, и получаем значения для проверки элемента ввода данных.

-1, 0, 1, 17, 18, 19, 24, 25, 26, 44, 45, 46, max.

Значение **max** обычно уточняется у Заказчика или аналитика. Если не могут предоставить, то следует бросить его и не проверять необходимо подобрать значение, соответствующее здравому смыслу (вряд ли кто-то придет за кредитом в возрасте 100 лет).

Следующий шаг, это наложить граничные значения на значения классов эквивалентности, исключить лишние проверки, пользуясь правилом «достаточно одного значения для проверки одного класса» и финализировать список.

Если ранее у нас были 3 значения для 3-х классов, 19, 30 и 48, то после определения граничных значений, мы можем исключить из списка значения 30 и 48 и заменить их предграницными значениями, такими как 26 (вместо 30) и 46 (вместо 48).

Граничные значения определяются не только для числовых значений, но и для буквенных (например, границы алфавита и кодировки), даты и времени, смысловых

значений. Граница числовых значений зависит от формата ввода, если у вас целые числа, например, 2, то граничные значения будут 1 и 3. Если дробные значения, то границы для числа 2 уже будут 1,9 (1,99) или 2,1 (2,01) и т.д.

Эквивалентное разбиение и Анализ граничных значений (Пример 2):

Рассмотрим пример. Допустим, мы тестируем Интернет-магазин, продающий карандаши. В заказе необходимо указать количество карандашей (максимум для заказа – 1000 штук). В зависимости от заказанного количества карандашей различается стоимость:

- 1 – 100 – 10 руб. за карандаш;
- 101 – 200 – 9 руб. за карандаш;
- 201 – 300 – 8 руб. за карандаш и т.д. С каждой новой сотней, цена уменьшается на рубль.

Если тестировать «в лоб», то, чтобы проверить все возможные варианты обработки заказанного количества карандашей, нужно написать очень много тестов (вспоминаем, что можно заказать аж 1000 штук), а потом еще все это и протестировать. Попробуем применить разбиение на классы эквивалентности. Очевидно, что наши входные данные мы можем разделить на следующие классы эквивалентности:

1. Невалидное значение: >1000 штук;
2. Невалидное значение: <=0;
3. Валидное значение: от 1 до 100;
4. Валидное значение: от 101 до 200;
5. Валидное значение: от 201 до 300;
6. Валидное значение: от 301 до 400;
7. Валидное значение: от 401 до 500;
8. Валидное значение: от 501 до 600;
9. Валидное значение: от 601 до 700;
10. Валидное значение: от 701 до 800;
11. Валидное значение: от 801 до 900;
12. Валидное значение: от 901 до 1000.

Диаграмма переходных состояний

Тестирование переходов состояний (state transition testing):

Диаграмма состояний и переходов показывает начальное и конечное состояния системы, а также описывает переходы между состояниями. Каждый переход вызывается событием (например, вводом данных пользователем). Если одно и то же событие может привести к разным переходам, выбор перехода может задаваться контрольным условием. Смена состояния может завершаться выполнением какого-либо действия (вывод результатов или сообщения об ошибке и т.д.).

Таблица переходов представляет собой все возможные комбинации начальных и конечных состояний. Она включает в себя действительные и недействительные переходы, инициирующие события, защитные условия и результирующие действия.

Диаграммы состояний и переходов показывают только действительные переходы и исключают недействительные переходы.

ПРИМЕР

Состояния воды:

1. Лёд – твердое состояние.
2. Вода – жидкое состояние.
3. Пар – газообразное состояние.

Состояние **ВОДА**

→ Действие **ОХЛАДИТЬ**

→ Состояние **ЛЕД**

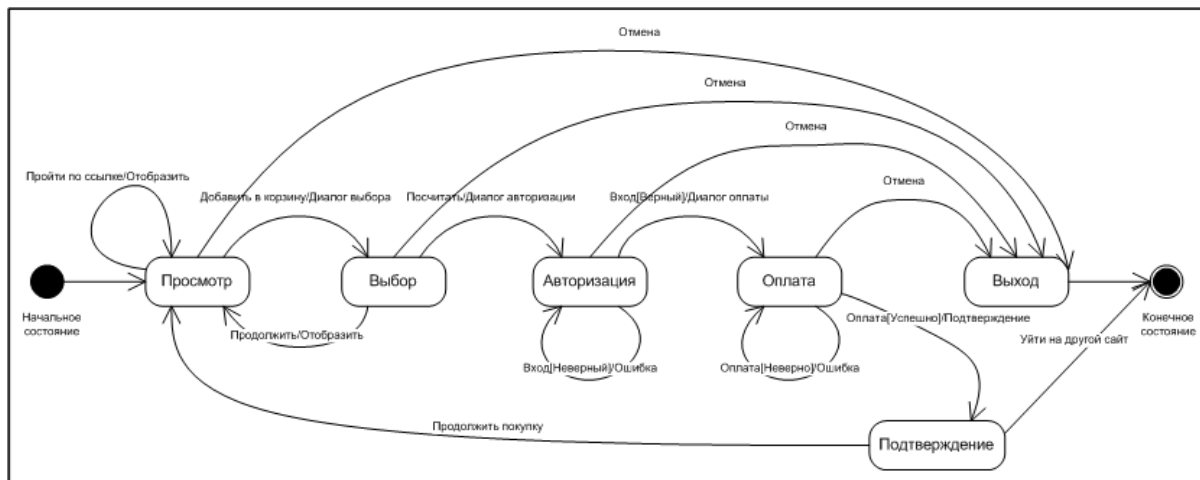


Рис. 1. Пример диаграммы переходов состояний.

Часто при разборе тестирования на основе состояний, люди спрашивают: «Как различить состояние, событие и действие?» Основные отличия следующие:

- В состоянии объект находится до тех пор, пока что-то не произойдет – что-то внешнее по отношению к самому объекту, обычно – вызывающее переход. В состоянии объект может существовать бесконечно долго.
- Событие происходит либо сразу, либо в ограниченный, конечный период времени. Это что-то, что наступает, происходит вне, – и вызывает переход.
- Действие – это ответ системы во время перехода. Действие, как и событие, либо мгновенно, либо требует ограниченного, конечного времени.

Завершающая проверка покрытия.

На рисунке изображено покрытие, которое было достигнуто путем дополнительных тестовых процедур, которые покрывают диаграмму переходов состояний:

1. («просмотр», «пройти по ссылке», «отобразить», «добавить в корзину», «диалог выбора», «продолжить», «отобразить», «добавить в корзину», «диалог выбора», «посчитать», «диалог авторизации», «вход [неверный]», «ошибка», «вход [верный]», «диалог оплаты», «оплата [неверно]», «ошибка», «оплата [успешно]», «подтверждение», «продолжить покупку», «отобразить», «отмена», «выход»);
2. («просмотр», «добавить в корзину», «диалог выбора», «отмена», «выход»);
3. («просмотр», «добавить в корзину», «диалог выбора», «посчитать», «диалог авторизации», «отмена», «выход»);
4. («просмотр», «добавить в корзину», «диалог выбора», «посчитать», «диалог авторизации», «вход [верный]», «диалог оплаты», «отмена», «выход»);
5. («просмотр», «добавить в корзину», «диалог выбора», «продолжить», «отобразить», «добавить в корзину», «диалог выбора», «посчитать», «диалог авторизации», «вход [верный]», «диалог оплаты», «оплата [успешно]», «подтверждение», «уйти на другой сайт»).

Рисунок отражает проверку покрытия состояний и переходов. Процесс нельзя считать завершенным до тех пор, пока каждое состояние и каждый переход не помечены, как это сделано на этом рисунке.

Метод попарного тестирования

Pairwise testing — техника тест-дизайна, а именно метод обнаружения дефектов с использованием комбинационного метода из двух тестовых случаев.

Тестирование с помощью алгоритма All-Pairs

All-pairs testing — комбинаторный метод тестирования программного обеспечения, который проверяет все возможные дискретные комбинации параметров для каждой пары входных параметров системы. Исходя из этого, мы получим меньшее число комбинаций, чем при использовании ортогональных матриц.

При попарном тестировании достаточно проверить лишь пары значений. При успешном выполнении тестов на 97% мы можем быть уверены, что проверяемая функциональность работает корректно.

Рассмотрим пример. Предположим, нам необходимо протестировать приложение для покупки/продажи б/у ноутбуков, мы имеем следующие переменные:

- категория заказа: покупка, продажа;
- местоположение: Киев, Харьков;
- марка ноутбука: HP, Lenovo, Asus;
- ОС: доступна, недоступна;
- тип расчета: наличный, безналичный;
- тип доставки: почтой, встреча.

Если мы захотим протестировать все возможные комбинации, то мы должны составить $2 \times 2 \times 3 \times 2 \times 2 \times 2 = 96$ тест-кейсов. Не многовато ли работы для тестирования формы?

Значком тильды “~” мы маркируем переменные, которые выступают произвольными. Таким образом мы получаем следующую таблицу.

Марка	Категория заказа	Местоположение	ОС	Расчет	Доставка
HP	покупка	Киев	+	наличный	почтой
HP	продажа	Харьков	-	безналичный	встреча
~HP~	покупка	~Харьков~	~-~	~безналичный~	встреча
Lenovo	покупка	Харьков	+	безналичный	почтой
Lenovo	продажа	Киев	-	наличный	встреча
~Lenovo~	продажа	~Киев~	~+~	~наличный~	почтой
Asus	покупка	Киев	-	безналичный	почтой
Asus	продажа	Харьков	+	наличный	встреча

Таким образом, мы получили готовые 8 тест-кейсов вместо 96.

Рассмотрим работу программы на примере из приведенной выше статьи из блога. Имеем следующие параметры и их значения: **пол** – мужской или женский; **возраст** – до 25, от 25 до 60, более 60; **наличие детей** – да или нет. Если перебирать все возможные значения, то количество сценариев будет 12. Составим модель и посмотрим какой результат нам выдаст программа.

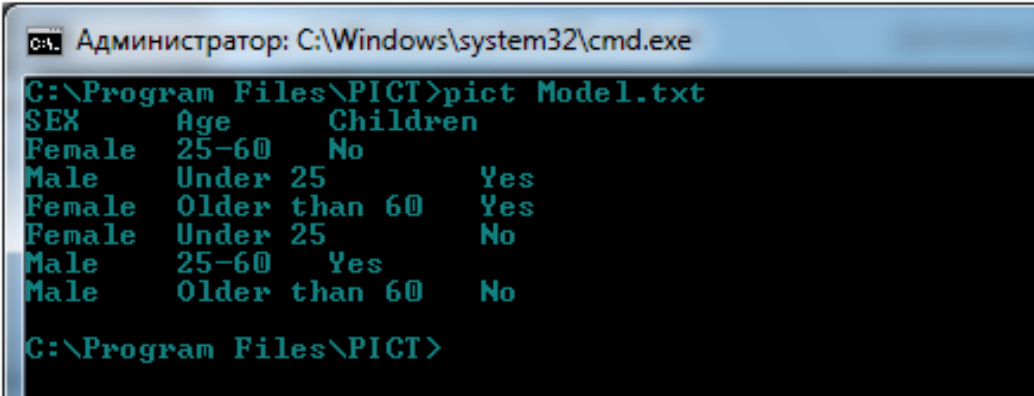
Модель:

SEX: Male, Female

Age: Under 25, 25-60, Older than 60

Children: Yes, No

Используем модель и получим 7 тестовых сценариев (вместо 12):



```
Администратор: C:\Windows\system32\cmd.exe
C:\Program Files\PICT>pict Model.txt
SEX      Age      Children
Female   25-60     No
Male     Under 25   Yes
Female   Older than 60  Yes
Female   Under 25     No
Male     25-60      Yes
Male     Older than 60  No
C:\Program Files\PICT>
```

Разница не такая ощутимая, но она будет становится все более и более заметной при увеличении количества параметров или их значений.

Можно использовать прямой вывод и сохранение тест кейсов в Excel.

```
C:\Program Files\PICT>pict Model.txt > example1.xls
```

В результате будет создан Excel файл со следующим содержанием:

	A	B	C	
1	SEX	Age	Children	
2	Female	25-60	No	
3	Male	Under 25	Yes	
4	Female	Older than 60	Yes	
5	Female	Under 25	No	
6	Male	25-60	Yes	
7	Male	Older than 60	No	

Таблица принятия решений

Таблица принятия решений — это способ компактного представления модели со сложной логикой. Простыми словами, это варианты действий при различных входных условиях.

Таблица принятия решений содержит следующие элементы:

- Условия — список возможных условий
- Варианты — комбинация из выполнения и/или невыполнения условий этого списка
- Действия — список возможных действий (вариантов исхода)

ПРИМЕР. Применение скидки в корзине покупателя										
	Тест 1	Тест 2	Тест 3	Тест 4	Тест 5	Тест 6	Тест 7	Тест 8	Тест 9	Тест 10
Покупатель авторизован?	+	+	+	+	+	-	-	-	-	-
Количество товара от 1 шт. до 2 шт.?	+	-	-	-	-	+	-	-	-	-
Количество товара от 3 шт. до 4 шт.?	-	+	-	-	-	-	+	-	-	-
Количество товара от 5 шт. до 9 шт.?	-	-	+	-	-	-	-	+	-	-
Скидка за авторизацию 7%	+	+	+	+	+	-	-	-	-	-
Скидка за количество (1-2 шт) - 0%	+					+				
Скидка за количество (3-4 шт) - 3%		+					+			
Скидка за количество (5-9 шт) - 5%			+					+		

Давайте заменим случай из жизни на случай из тестирования, например, известная всем **форма авторизации в системе**. Такая таблица позволит нам рассмотреть все возможные варианты развития событий и результат каждого такого события.

Мы знаем, что форма содержит поля логина, пароля и кнопки **“Войти”** и **“Отмена”**.

При вводе неверных данных система выдает соответствующую ошибку о том, что логин или пароль введены неверно. Если мы не ввели значение для логина или пароля — система выдает ошибку о необходимости заполнить поля.

Выберем “Условия” для данных сущностей, т.е. возможные входные значения.

- Логин: пустое значение / верное значение / неверное значение
- Пароль: пустое значение / верное значение / неверное значение
- Кнопки: Войти / Отмена

Пустое значение выбрано как отдельное условие из-за того, что ошибка в этом случае отличается от ошибки ввода неверного значения. Кнопки объединили в одно условие, т.к. мы можем нажать или одну или другую кнопку, одновременно нажать две — проблематично

Выделим возможные действия:

- Успешная авторизация
- Ошибка “Неверно введены логин или пароль”
- Ошибка “Заполните поля логин или пароль”
- Авторизация отменена

Считаем, что в случае неверного значения для логина и пустого значения пароля — система выдает нам обе ошибки. Как должна вести себя ваша программа — смотрите в требованиях и спецификациях

Теперь для всех 18 вариантов определим необходимость действий.

Условие	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Логин	Пустое	Пустое	Верное	Верное	Верное	Верное	Верное	Верное	Пустое	Пустое	Пустое	Пустое	Неверное	Неверное	Неверное	Неверное	Неверное	Неверное
Пароль	Пустое	Пустое	Верное	Верное	Пустое	Пустое	Неверное	Неверное	Верное	Верное	Неверное	Неверное	Верное	Верное	Неверное	Неверное	Пустое	Пустое
Кнопка	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена	Войти	Отмена
Действие																		
Успешная авторизация			X															
Отмена авторизации		X		X		X		X		X		X		X		X		X
Ошибка "Неверно введены логин или пароль"							X				X		X		X		X	
Ошибка "Заполните поля логин или пароль"	X				X				X		X						X	

В итоге для проверки всех возможных вариантов действий с формой авторизации нам потребуется 18 тест-кейсов. Они по факту уже готовы и записаны в таблице.

(Пример 2)

Рассмотрим на примере функции "Регистрация нового пользователя на сайте". Предположим, что веб-форма содержит два обязательных к заполнению поля: имя (не логин) и емейл. Для упрощения, не будем углубляться в требования к каждому полю. Ограничимся лишь номинальными понятиями "корректных" и "некорректных" значений. Итак, чтобы регистрация прошла успешно, необходимо заполнить корректными данными оба поля. Если поля заполняются некорректными данными и/или одно из полей пустое, то система должна выдать сообщение, мол, исправьте введенные данные.

	Правило 1	Правило 2	Правило 3	Правило 4
Условия				
Ввод корректных данных в поле "E-mail"	+	-	+	-
Ввод корректных данных в поле "Имя"	+	-	-	+
Ввод некорректных данных в поле "E-mail"	-	+	-	+
Ввод некорректных данных в поле "Имя"	-	+	+	-
Действия				
Регистрация успешна	+	-	-	-
Выдается ошибка, мол, исправьте данные	-	+	+	+

Правила 2, 3 и 4 приводят к одному и тому же результату, только с разными входными значениями. Очень важно при разработке тест-кейсов не пропустить различные *пути прохождения* продукта для получения одинакового результата.