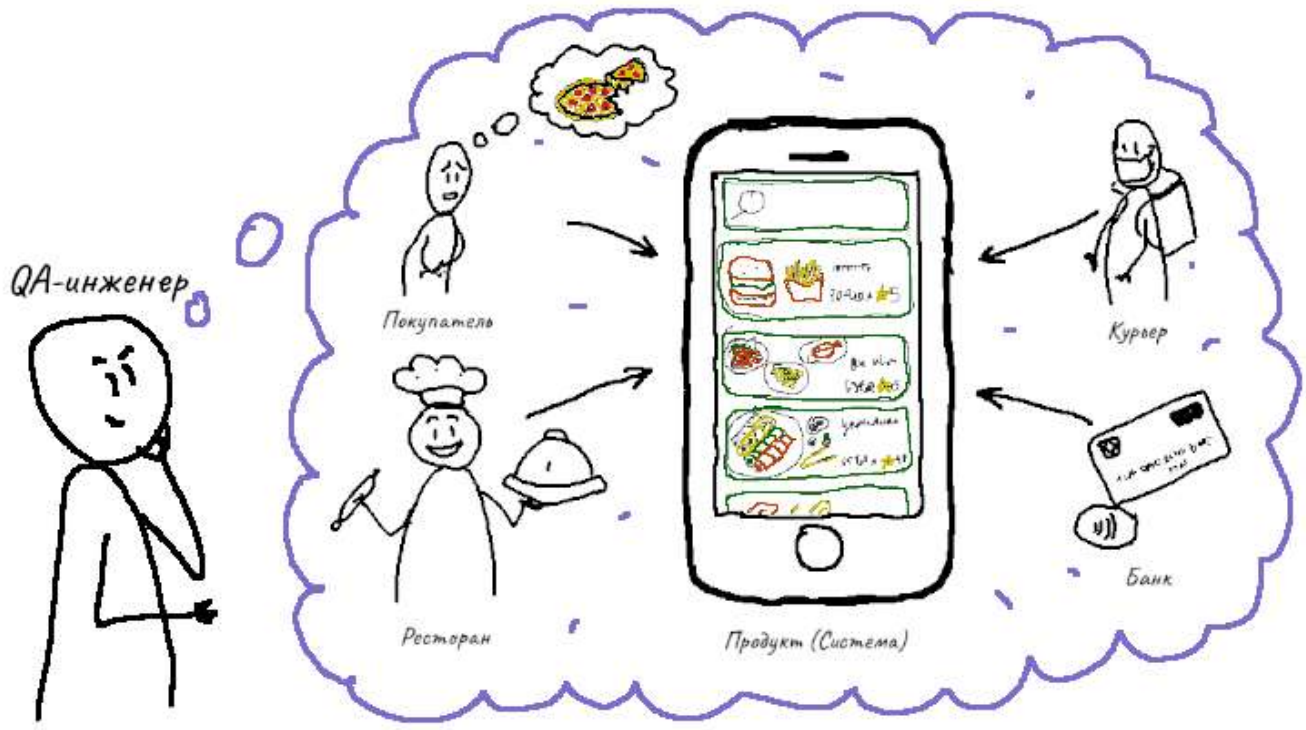
Olya_letsLED 5 апреля 2021 в 06:40

Тестирование требований: как я нахожу ошибки в бизнес-логике фичи прежде, чем их закодят

Блог компании Plesk , Тестирование IT-систем *, Анализ и проектирование систем *, Usability *

Технотекст 2021 Tutorial



Привет, Хабр. Меня зовут Ольга, я работаю в тестировании с 2013 года, специализируюсь на тест-анализе и тест-дизайне. Сегодня хочу рассказать, как при планировании тестирования

Часто тестировщики начинают планирование тестирования с составления карты приложения. Т.е. формируют список страниц и перечисляют все контролы на странице. Это приводит к тому, что каждая страница сама по себе работает, но это не значит, что пользователь может выполнить свою задачу целиком.

За время работы я обратила внимание, что аналитики используют в своей работе те же техники, что и тестировщики при тест-анализе, и их можно применять как для тестирования требований, так и для составления тестов по задаче.

Под катом покажу, как с помощью одной техники Use Cases найти пробелы в бизнес-логике, протестировать требования и дизайн прежде, чем написан код, и убедиться, что пользователь сможет использовать приложение, даже если что-то пойдет не так.

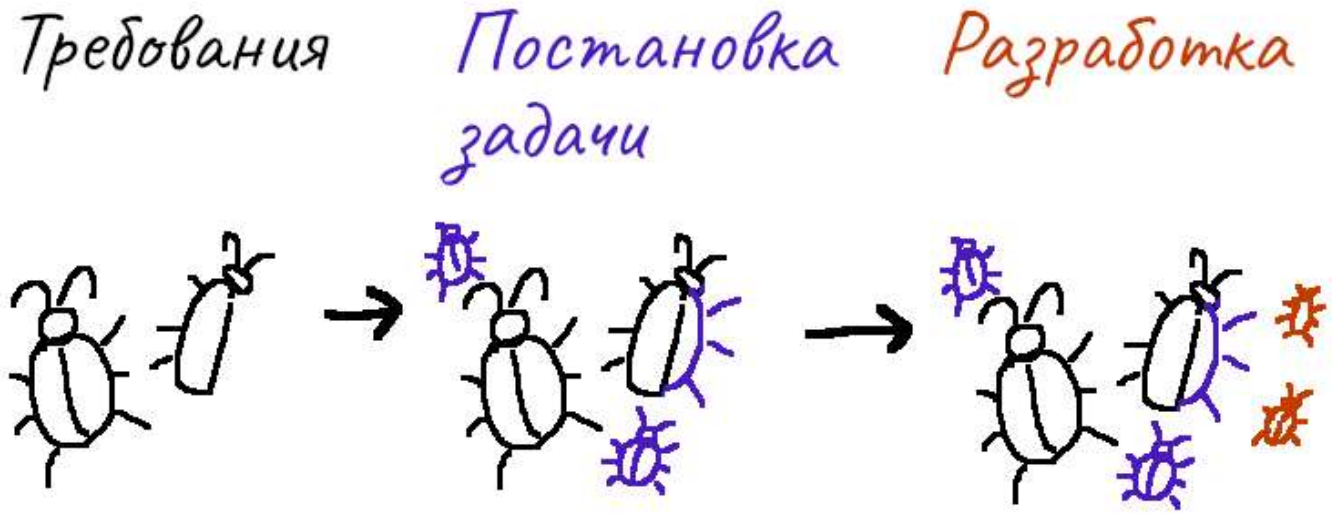
Статья получилась объемной, так что вот оглавление:

Введение

Есть разные команды: в одних командах куча аналитиков пишут требования к софту на сотню

Есть разные команды: в одних командах куча аналитиков пишут требования к софту на сотни страниц, в других есть 1 аналитик на несколько проектов и он пишет, как успеется, в третьих вообще нет аналитика и задачи ставит ПМ, а команда уже на ходу придумывает, что именно реализовать. Причем, может быть непонятно, что нужно сделать по задаче, независимо от того, есть ли аналитик на проекте. И чаще всего так и происходит. То задача идет вразрез с текущим функционалом, то никто не понимает, что там вообще надо сделать, то все всё понимают, но каждый по-своему.

Самые нажористые, сложные в исправлении баги обычно скрываются в требованиях и в постановке задачи.



Есть разные варианты, что может пойти не так:

- Ожидания пользователей (пользователи хотят одно, а мы думаем, что они хотят другое; нет важных фичей; фичи есть, но неудобные).
- Ожидания заказчика (ПМ хочет то, что невозможно/дорого реализовать; или то, что противоречит уже существующей логике).
- Ожидания бизнеса (бизнес хочет платную фичу, а мы не разобравшись даем ее пользователю бесплатно).
- Предполагаемые алгоритмы (не делают кое-что важное; не проверяют предусловия; делают вообще не то, что ожидается; от пользователя требуется слишком много лишних действий).

Если ошибки из списка выше тестировщик найдет после того, как разработчик закончил кодить, то исправление будет стоить очень дорого, ведь нужно переделать все: и требования, и интерфейс, и реализацию, и всё это снова протестить.

и интерфеис, и реализацию, и все это заново перетестить.

> Чтобы предоставлять информацию о проблемах своевременно и сократить количество переделок, начинать тестировать нужно как можно раньше.

Другой вопрос: а как тестить до разработки, если по постановке задачи ничего непонятно?

Для этого нам нужно определить:

- Зачем и для кого фича.
- Как пользователь/заказчик будет использовать эту фичу.

Это позволит сформулировать задачу понятнее для команды, отловить ошибки в требованиях прежде, чем они будут закодированы и спланировать тестирование.

Зачем и для кого фича?

В первую очередь нужно определить, для кого мы делаем эту фичу, и зачем она заказчику.

Это можно спросить у автора задачи.

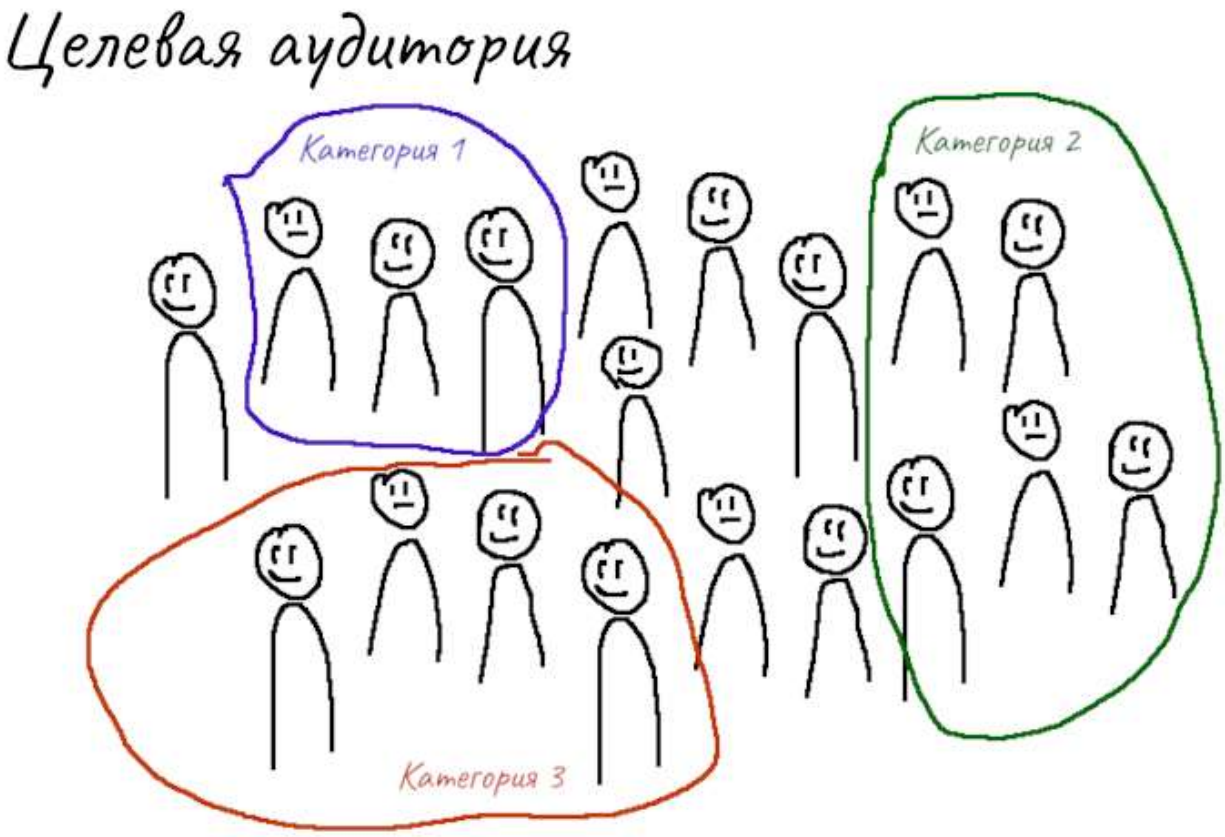
> Стоит различать пользователя продукта и заказчика.

Например, если ваш продукт — это софт для частной клиники, то заказчиком тут будет представитель бизнеса, допустим, владелец клиники. А конечные пользователи - это врачи, регистраторы, медсестры и другой персонал клиники, которые используют продукт, чтобы

вести записи о здоровье пациентов.

Пользователи продукта тоже могут быть разные. Важно выявить:

- Какие категории пользователей есть в вашей целевой аудитории (ЦА).
- Зачем люди из каждой категории используют эту фичу.
- Какие проблемы они решают с помощью нее.
- Какие есть варианты использования фичи.



Итак, задача может быть для нужд бизнеса, например, чтобы организовать PR-компанию, или уменьшить затраты на X, или увеличить доходы с У, или предотвратить/остановить отток пользователей.

Или для определенной категории пользователей нужно реализовать фичу, которую пользователи давно хотят и просят.

Задача может быть для ПМа или владельца продукта, например, чтобы собрать статистику и на основе нее принять решение. При этом важно понимать, что это за решение.

Для команды разработки — например, рефакторинг определенной области.

Для саппорта — чтобы лучше локализовать проблему или чтобы снизить нагрузку на саппорт.

Для другой команды — обычно это интеграции или еще какой-то обмен данными.

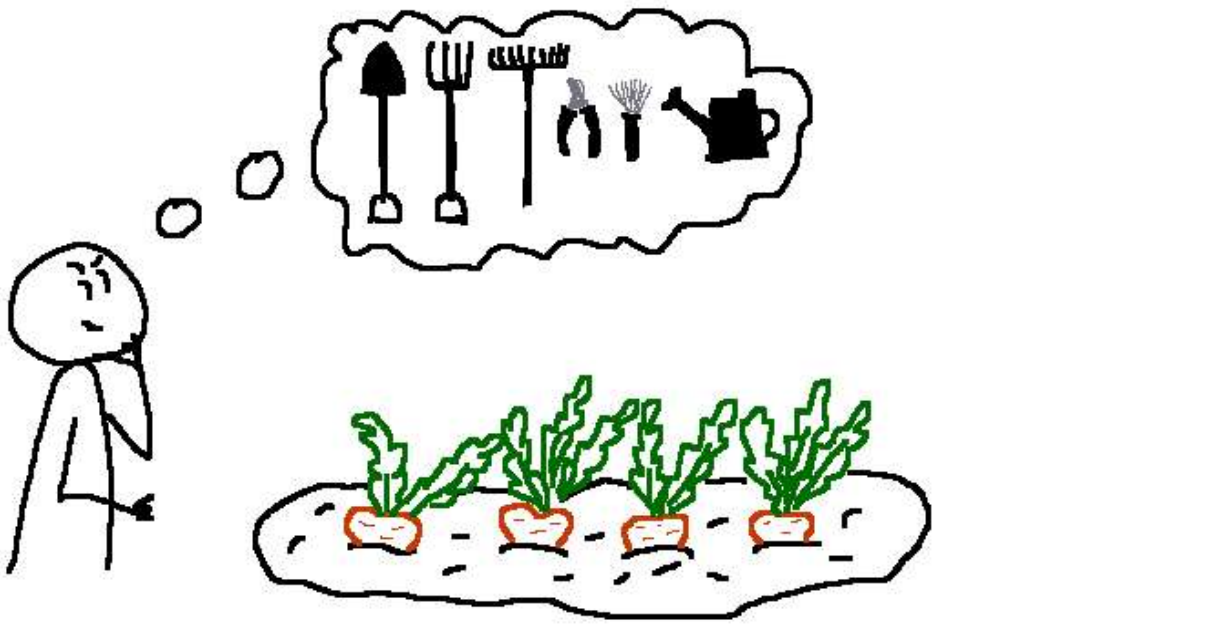
Какую проблему решает фича?

С заказчиком разобрались. Дальше нужно понять, какую проблему заказчика решает эта фича и действительно ли проблема будет решена релизом этой фичи.

Можно обдумать это еще и в ключе: как пользователь/заказчик решает эту проблему сейчас?

Например:

- Делает руками, долго и болезненно. Или наоборот, руками получается быстро и ему ок.
- Запускает какой-то скрипт.
- Решает проблему через другое приложение.
- Никак не делает и страдает.
- Никак не делает и ему норм.



Ответ на этот вопрос поможет понять: а мы этой фичей точно улучшим пользователю работу или еще хуже сделаем?

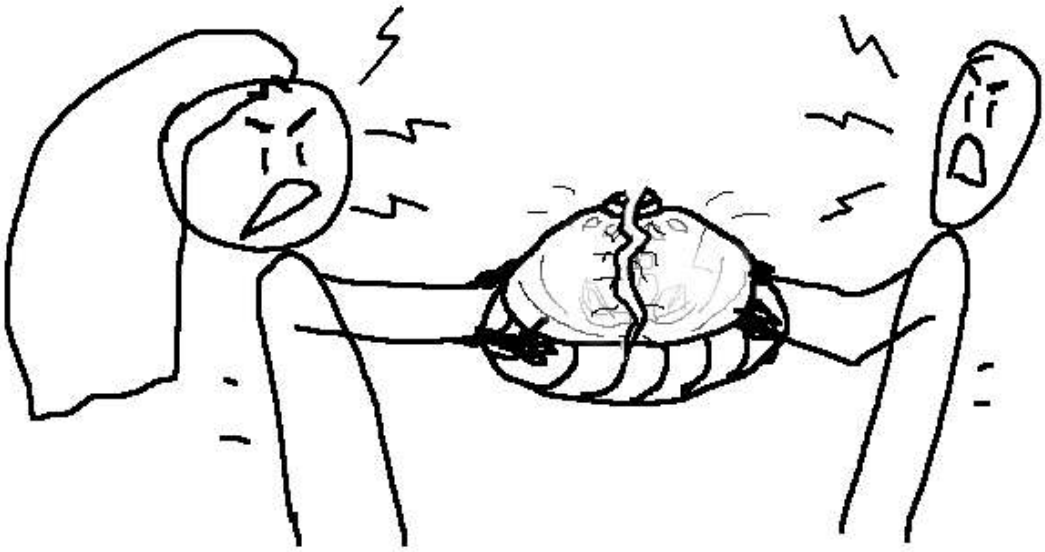
Если заказчик — команда разработки, то вполне вероятно, что это какое-то исследование, рефакторинг, или что-то сугубо внутреннее, то возможно, тут и не нужно пытаться анализировать каждый шаг. Лучше сфокусироваться на поиске последствий для пользователей.

Если заказчик — другая команда и фича про интеграции или обмен данными, то есть риск подумать, что задача простая, т.к. мы просто передаем данные от одного приложения к другому. А приложения бесчувственные, терпимо относятся к огрехам и шероховатостям передачи, так что можно и не стараться.

Но если посмотреть на это глубже, то всегда при передаче данных от одного приложения к другому можно по цепочке дойти до человека, который заинтересован в результатах этого обмена данными. Вот на этого человека и нужно ориентироваться и трактовать его как пользователя.

Конфликт интересов

При выявлении требований бывает, что у фичи есть сразу несколько заинтересованных сторон и их интересы могут конфликтовать. Нужно это выявить на ранних этапах, предупредить команду и разрешить эти конфликты до начала разработки, даже если потребуется отложить фичу на следующий релиз.



Конфликт интересов

В книге «Разработка требований к программному обеспечению» (Карл Вигерс, Джой Битти) я нашла замечательную таблицу о том, как разрешать конфликты между сторонами при определении требований. Привожу ее здесь:

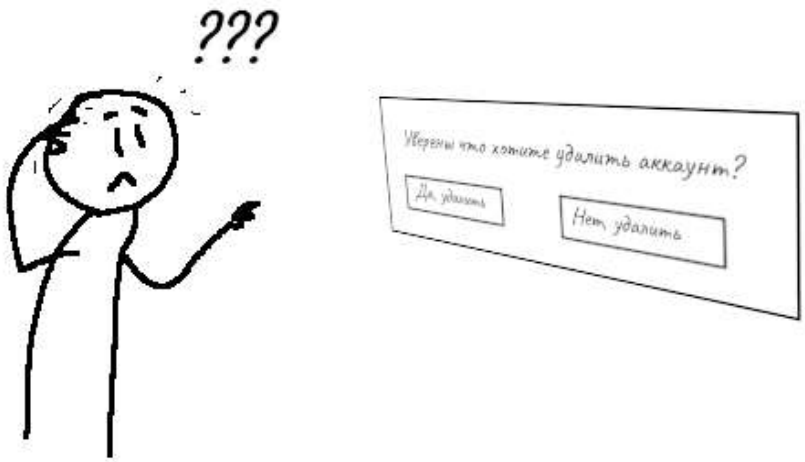
Разногласия между...	**Как разрешать**
...отдельными пользователями	Решение принимает сторонник или владелец продукта
...классами пользователей	Предпочтение отдается наиболее важному классу пользователей
...сегментами рынка	Предпочтение отдается сегменту, оказывающему наибольшее влияние на успех бизнеса
...корпоративными клиентами	Решение определяется на основе бизнес-целей
...пользователями и менеджерами	Решение принимает сторонник класса пользователей или владелец продукта
...разработчиками и клиентами	Предпочтение отдается клиентам, но с учетом бизнес-целей

...разработчиками и маркетингом	Предпочтение отдается специалистам по маркетингу

Как пользователь будет использовать эту фичу?

Есть команды, которые используют подход, ориентированный на продукт. При разработке они предполагают, что функции, которые они придумали, понравятся пользователям.

Часто это приводит к тому, что одни функции не используются, т.к. не нужны пользователю; другие функции пользователю очень нужны, но он не может их найти; а третьи — делают не то, что ожидается.



Если при разработке ориентироваться на пользователя и его работу с продуктом, то этих проблем можно избежать. Для этого удобно использовать технику Варианты Использования (Use Cases).

Варианты использования (Use cases)

Варианты использования — это техника, которая применима как к продукту целиком, так и к конкретной фиче. Она позволяет смоделировать, как пользователи используют продукт, и как продукт реагирует на действия пользователя в той или иной ситуации.

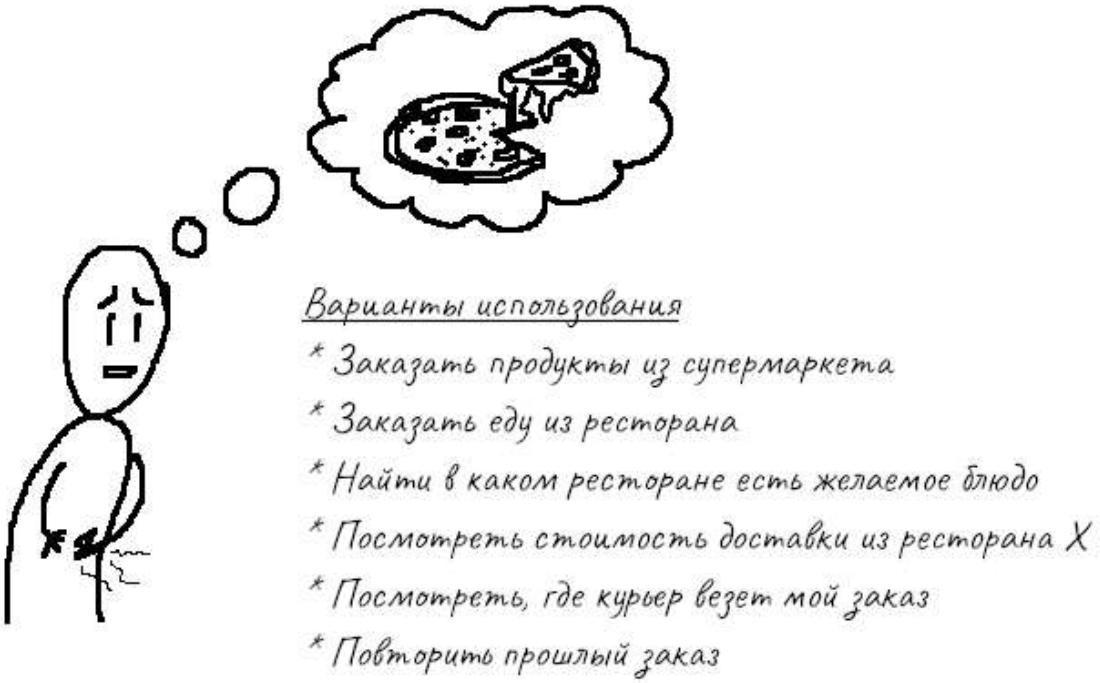
В результате такого моделирования получается набор сценариев, по сути, тест-кейсов, с помощью которых еще на этапе требований можно выявить нестыковки в логике фичи, ошибки в дизайне, непроработанные сценарии, или ситуации, когда пользователь оказывается в тупике и не достигает своей цели.

оказывается в тупике и не достигает своей цели.

Полную версию использования техники вы можете посмотреть в той же книге «Разработка требований к программному обеспечению». В этой статье я покажу упрощенную версию, которую использую в работе.

Возьмем для примера агрегатор доставки заказов из ресторанов и магазинов. В нем можно найти вот такие варианты использования:

- Заказать еду из ресторана с доставкой.
- Заказать еду с самовывозом.
- Заказать доставку продуктов из супермаркета.
- Найти, в каком ресторане есть желаемое блюдо.
- Посмотреть стоимость доставки из ресторана X.
- Повторить прошлый заказ.
- Посмотреть, где находится курьер, который доставляет заказ.



Имя варианта использования должно быть понятное (глагол + объект), описывающее, какую именно ценность пользователю дает этот вариант использования, например, информацию о чем-либо (внутри приложения) или еду на обед (в реальном мире, за пределами приложения).

приложения).

К вариантам использования требуется пошаговое описание того, как пользователь использует продукт и как продукт реагирует на действия пользователя. Так у нас есть как минимум 2 действующих лица: пользователь и продукт (система). Таких участников может быть несколько.

Примеры вопросов, чтобы найти действующих лиц:

- Кому приходит уведомление о событиях в системе?
- Кто предоставляет системе информацию или услугу?
- Кто помогает системе среагировать и выполнить задачу?

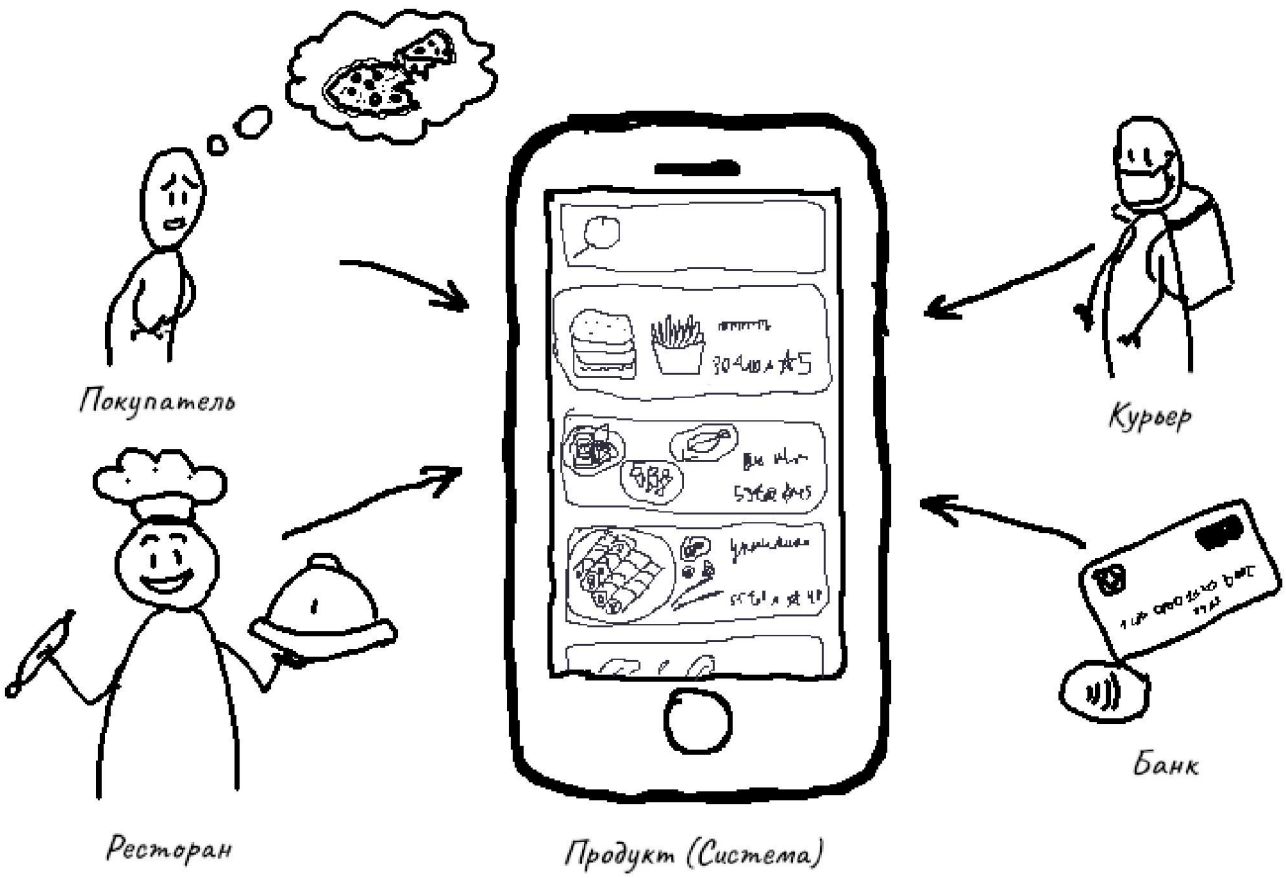
Экосистема агрегатора доставки еды предполагает, что есть приложение для того, кто заказывает еду, есть приложение для курьера, и приложение для ресторана.

Т.е. у нас будет как минимум 4 действующих лица:

- Покупатель
- Ресторан
- Курьер
- Система (т.е. само приложение)

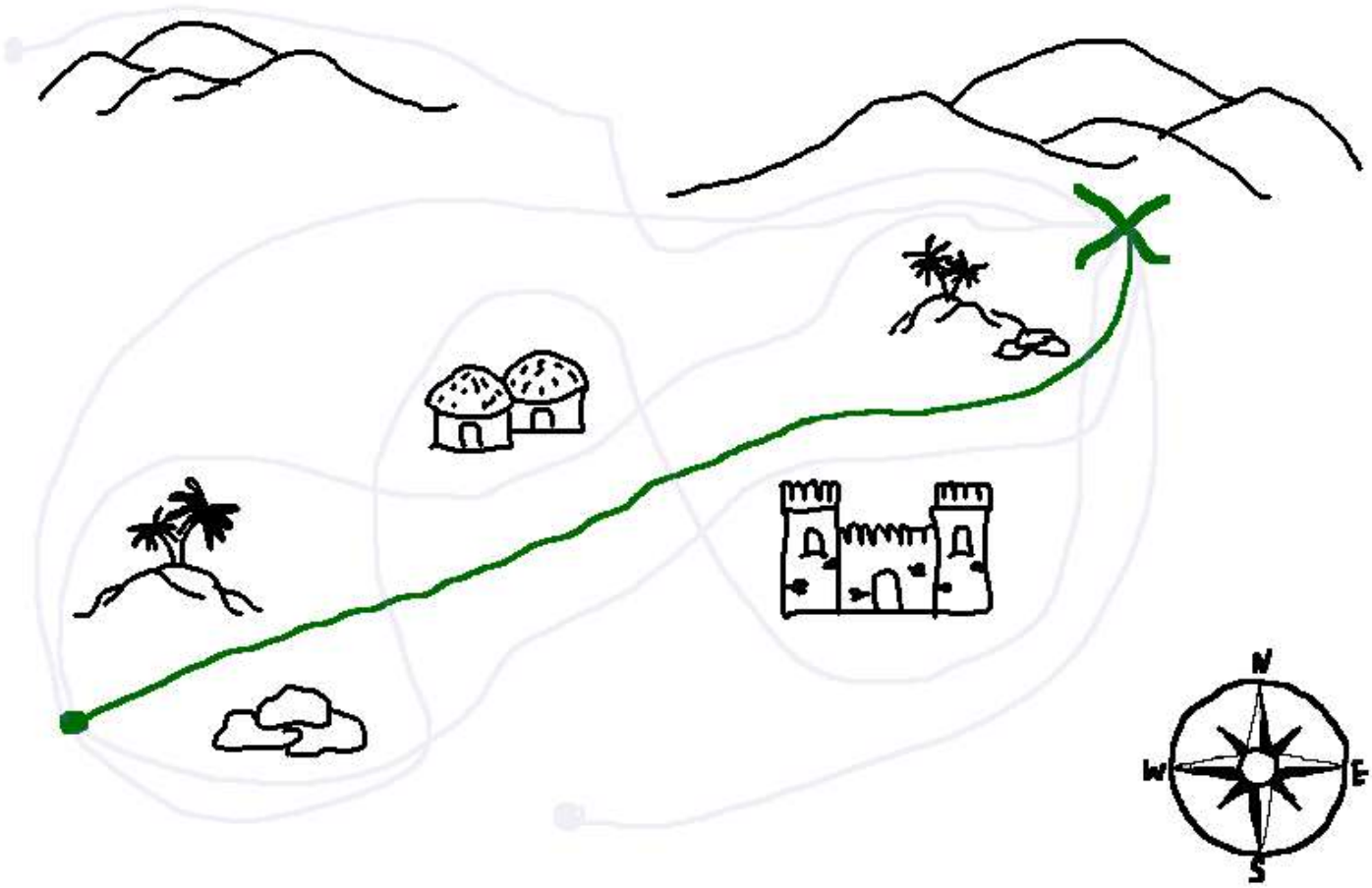
Для разных вариантов использования список действующих лиц может меняться.

Если покупатель выберет в заказе оплату картой, то добавится действующее лицо «Банк». Если закажет еду не себе, а бабушке, то появится еще «Получатель». А если будет писать в техподдержку, то добавятся «Бот» и «Оператор техподдержки».



Для каждого варианта использования нужно описать еще предусловия, входные и выходные данные, альтернативные сценарии развития и исключения, т.е. когда что-то пошло не так в сценарии.

Основное направление варианта использования



Определим варианты использования основной фичи агрегатора доставки — заказа еды из ресторана.

Вот так может выглядеть основной сценарий использования целиком: от того, как покупатель начал накидывать товары в корзину, и до того, как получил готовую еду в руки.

Действующие лица:

- П — Покупатель
- Р — Ресторан
- К — Курьер
- С — Система (т.е. само приложение)
- Б — Банк

Предусловия:

- Пользователь авторизован в приложении

- Доступен как минимум 1 ресторан
- Заказ делается в часы работы ресторана
- У ресторана есть блюда в наличии

Сценарий (основное направление):

П: Добавляет блюда в корзину

С: Проверяет доступность блюд

С: Показывает пользователю сумму заказа в корзине

С: Подсчитывает сумму и прогноз доставки

П: Переходит в Корзину

С: Показывает текущее содержимое заказа

П: Инициирует оформление заказа

С: Предлагает выбрать параметры заказа (адрес, время доставки, вид оплаты)

П: Выбирает оплату картой онлайн и инициирует оплату

Б: Открывает страницу подтверждения оплаты

П: Подтверждает оплату (вводит код из СМС)

Б: Проверяет подтверждение

Б: Передает в Систему данные, что заказ оплачен

С: Присылает покупателю чек об оплате

С: Показывает покупателю, что заказ принят

С: Запрашивает подтверждение заказа у Ресторана

Р: Подтверждает заказ

Р: *Начинает готовить заказ (в реальном мире)*

С: Находит Курьера и запрашивает у него подтверждение принятия заказа

К: Подтверждает, что принял заказ

К: *Идет в ресторан (в реальном мире)*

Р: Подтверждает, что заказ готов

К: Подтверждает, что принял заказ в доставку

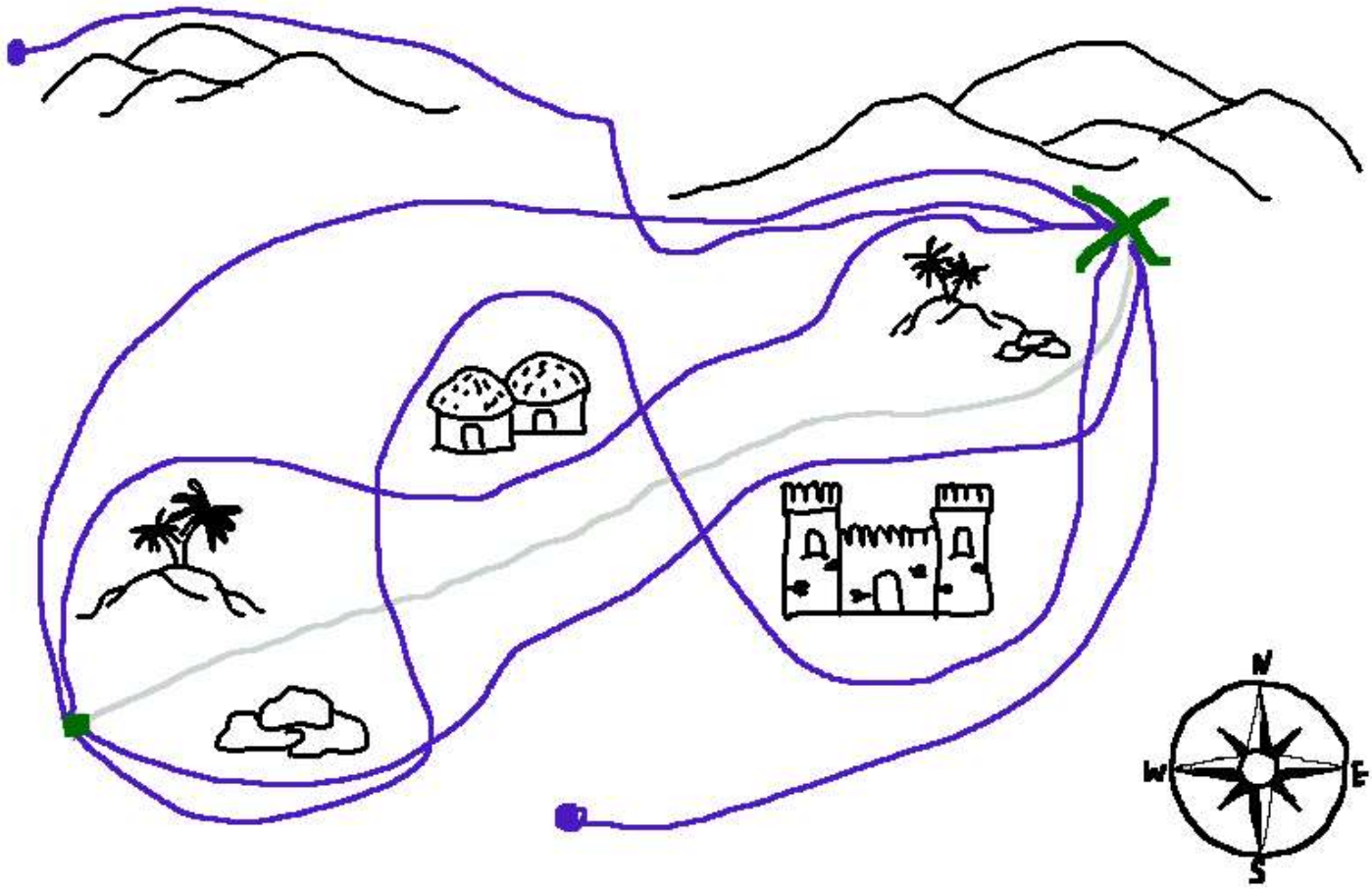
К: *Транспортирует заказ (в приложении видна геометка курьера на карте) (в реальном мире)*

П: Получает заказ

К: Подтверждает, что передал заказ покупателю

С: Показывает, что заказ завершен

Альтернативные направления



Дальше нужно составить альтернативные направления развития, т.е. описать, как можно выполнить все то же самое, но немного другим путем.

Можно посмотреть на сценарий целиком и подумать, как можно получить заказ другим способом:

Например:

- A-1 Зайти в раздел с завершенными заказами и повторить прошлый заказ

А можно смотреть на каждую строку и думать: «может ли действующее лицо сделать другой выбор на этом шаге?»

Например:

- A-2 Покупатель оформляет заказ на другого получателя (а не на себя)
- A-3 Покупатель выбирает оплату наличными (и сценарий минует шаги с онлайн-оплатой)

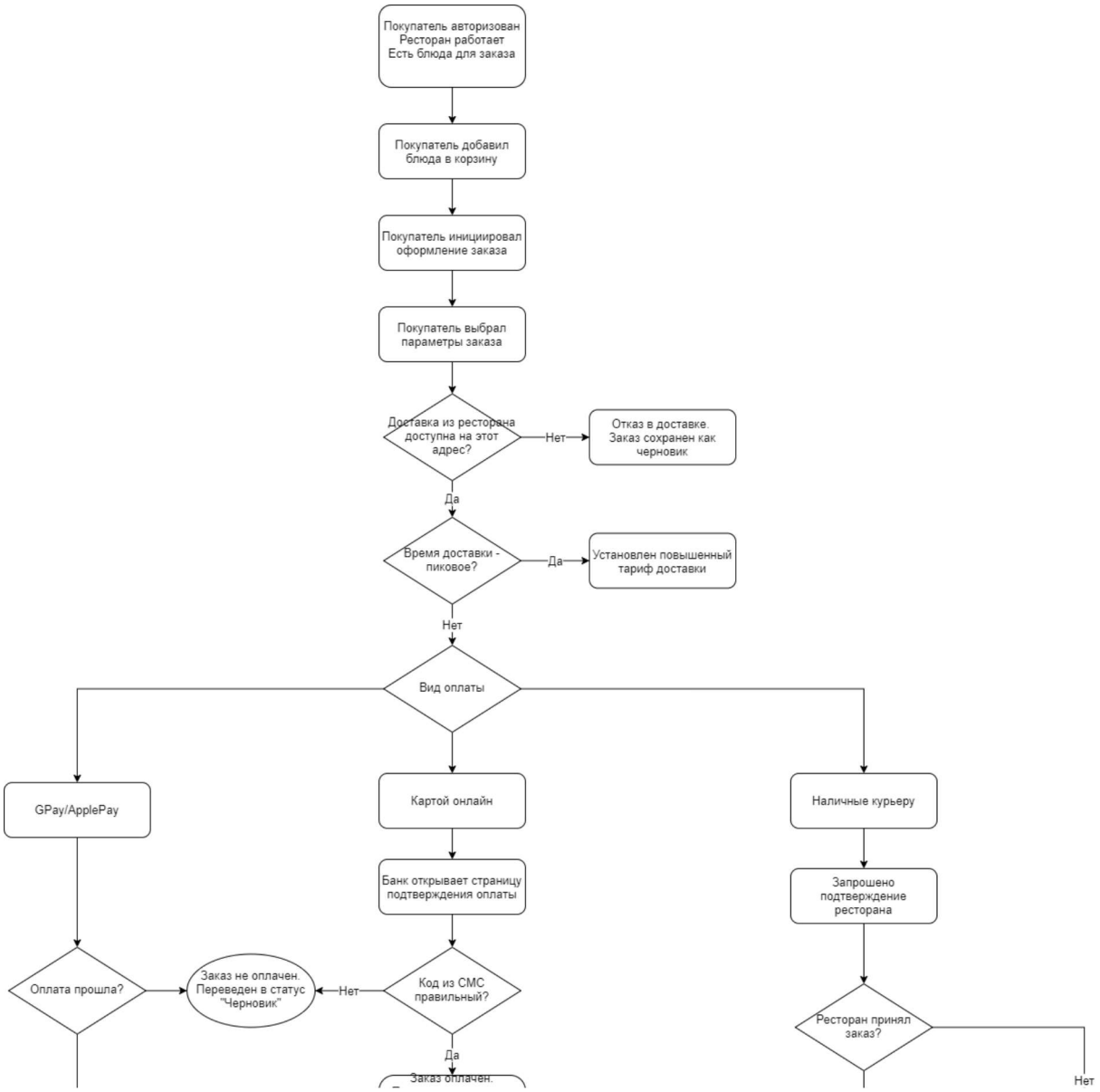
- А-4 Покупатель применяет промокод и сумма к оплате пересчитывается

- А-5 Покупатель выбирает доставку к определенному времени
- А-6 Курьер сначала принимает заказ, а потом отказывается и заказ берет новый курьер

Нумерация в альтернативах не относится к порядковому номеру шага. А часть альтернатив могут вообще начинаться в другой точке входа, так что цифры можно расставить просто по порядку.

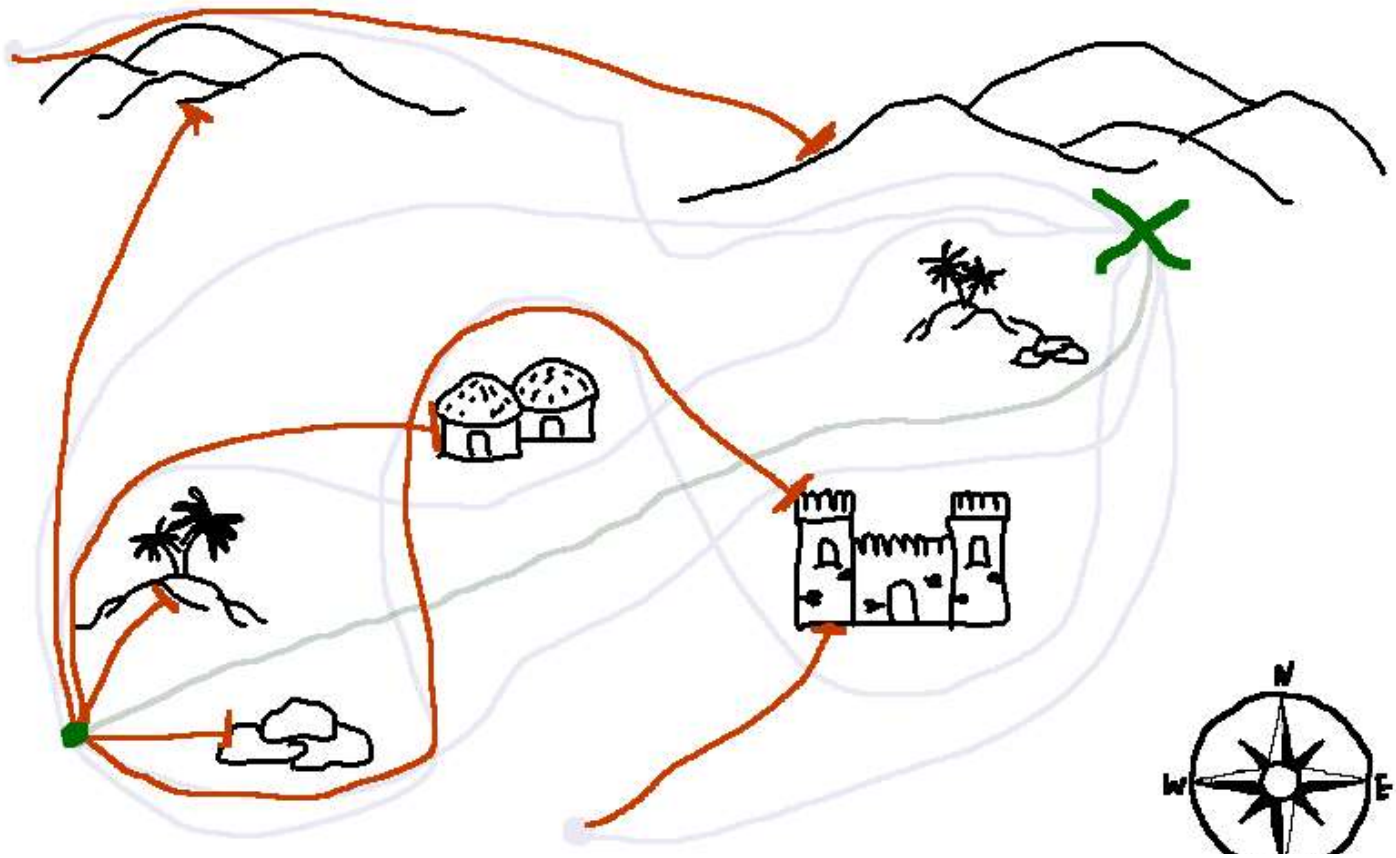
Каждую такую альтернативу можно пошагово расписать, как и основной сценарий. А можно нарисовать блок-схему.

▾ Пример блок-схемы




Да
Покупателю показано подтверждение. Отправлен чек по операции. Запрошено подтверждение ресторана
Ресторан принял заказ?
Нет
Да
Ресторан начал готовить заказ
Курьер найден?
Нет
Заказ отменен. Деньги возвращены на карту. Покупателю показано уведомление об отмене заказа.
Да
Ресторан передал заказ курьеру
Да
Курьер передал заказ покупателю?
Нет
Да
Покупатель получил еду. Заказ завершен.
Да
Ресторан начал готовить заказ
Да
Курьер найден?
Нет
Заказ отменен. Покупателю показано уведомление об отмене заказа.
Да
Ресторан передал заказ курьеру
Покупатель заплатил курьеру за заказ?
Нет
Да
Курьер передал заказ покупателю?
Нет
Да

Исключения


N
W
E

Если на каком-то шаге возникло условие, которое препятствует успешному завершению сценария, то это называется Исключение. Исключения описывают, как поведет себя система в случае ошибки. Для тестировщиков это те самые негативные сценарии. Вообще, техника Use Cases удобнее всего показывает все возможные ошибки, которые могут случиться во время сценария.

Чтобы найти все возможные исключения, нужно так же, как и при поиске альтернатив, пройтись по сценарию и подумать «что может пойти не так?» на каждом шаге сценария.

Например, когда пользователь добавляет блюда в корзину, а система проверяет доступны ли они для заказа, могут произойти такие исключения:

- И-1.1 Добавляемое блюда закончилось в ресторане и больше недоступно для заказа.
- И-1.2 У ресторана закончилось время работы.
- И-1.3 Ресторан прекратил прием заказов и вообще исчез из списка.
- И-1.4. В корзине лежат блюда другого ресторана (обычно при этом система предлагает убрать блюда другого ресторана из корзины).

На шаге подсчета суммы и прогноза времени доставки может оказаться, что:

- И2.1 Сумма заказа недостаточна для доставки (возможен только самовывоз).

При оплате онлайн бывает, что страница банка зависает, и невозможно ввести код оплаты.

Вот этот блок в сценарии:

П: Выбирает оплату картой онлайн и инициирует оплату.

Б: Открывает страницу подтверждения оплаты (но страница зависает).

П: Не может ввести код оплаты.

П: Обновляет страницу.

П: ~~Психует и~~ Переходит на предыдущий экран (т.е. обратно в приложение).

Обычно в этот момент система просто отменяет заказ и пользователю нужно пройти весь

путь оформления заказа заново.

Так и назовем исключение:

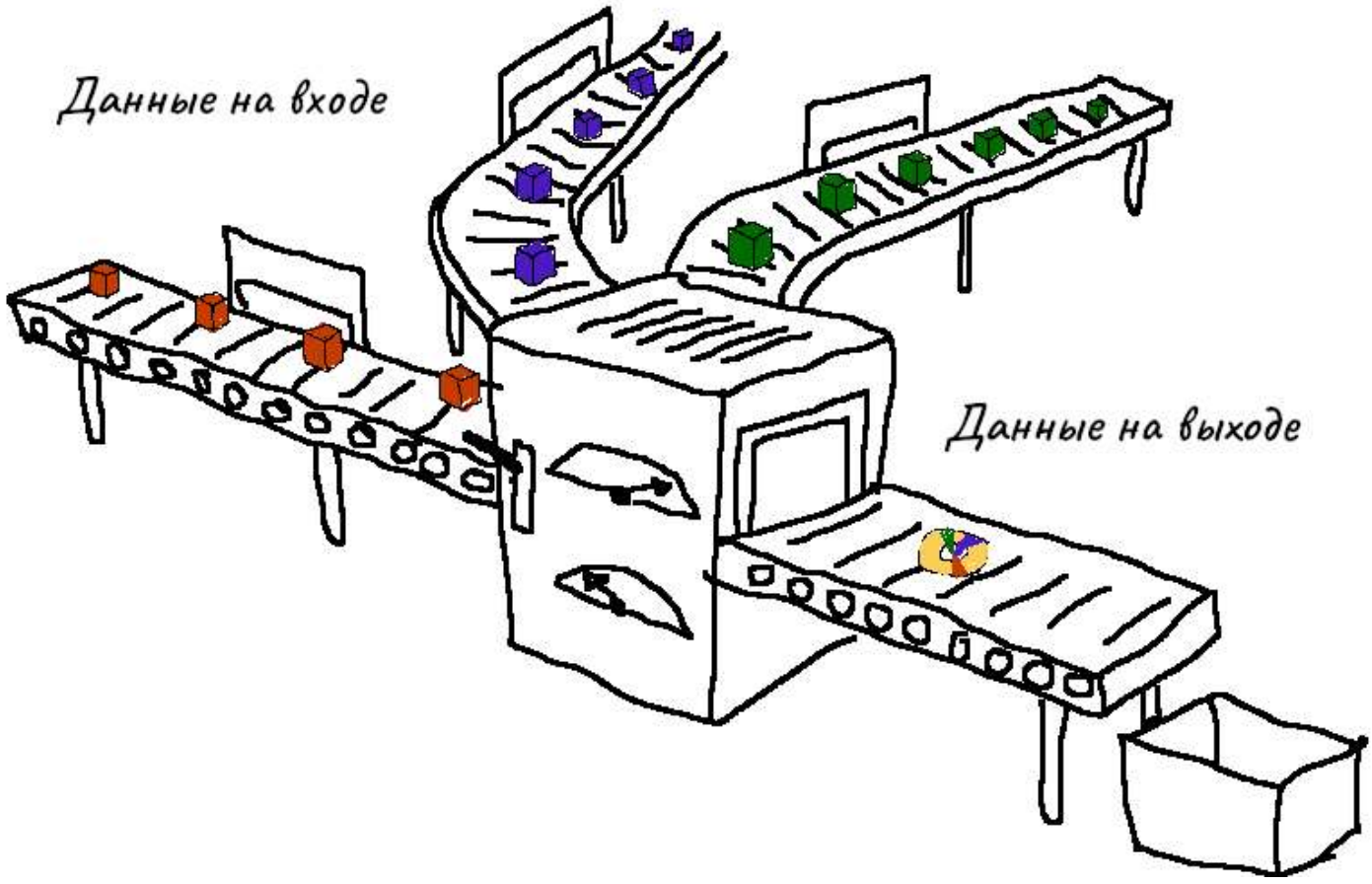
- И-3.1 Страница подтверждения оплаты зависла.

Еще несколько примеров исключений из этого сценария:

- И-4.1 Курьер не вышел на связь с рестораном (Система отменила заказ автоматически).
- И-4.2 Нет доступных курьеров в зоне доставки (Система отменила заказ автоматически).
- И-5.1 Курьер случайно отменил заказ когда еда уже была у него ~~и отдал заказ бомжу~~.

Так, анализируя исключения, можно предусмотреть большинство возможных ошибок, показать их разработчику и дизайнеру, спланировать, как показать в интерфейсе все эти ситуации так, чтобы пользователю было понятно: что произошло, и что нужно сделать, чтобы все-таки достичь своей цели.

От входных/выходных данных к тест-кейсам



На основе всех этих сценариев мы можем составить тест-кейсы, чтобы протестить работу фичи. Тест-кейс предусматривает, что мы подаем какие-то данные на вход, делаем нужные действия и ожидаем какой-то результат на выходе.

При этом одни и те же параметры на входе в разные сценарии могут иметь разные значения.

Например, при заказе еды в агрегаторе доставки на вход поступают вот такие параметры:

Покупатель:

- Имя
- Телефон
- Адрес
- Способ оплаты

Заказ:

- Блюда и их стоимость
- Количество приборов
- Прогнозируемое время доставки
- Время оформления заказа
- Промокод

Ресторан:

- Адрес
- Часы работы
- Доступные блюда

Курьер:

- Имя
- Телефон

- Местоположение
- Доступность для заказа

На выходе могут быть такие данные:

- Доставлен ли заказ
- Количество доставленных блюд
- Качество (вкус, температура) доставленных блюд
- Доставленные приборы (количество, комплектность)
- Стоимость доставки
- Стоимость заказа
- Фактическое время доставки

В случае, если значения в заказе обычные, то на выходе мы получим вовремя доставленную еду. Но некоторые значения параметров могут изменить результат на выходе.

Если Адрес покупателя будет расположен слишком далеко от Адреса ресторана, то доставка будет недоступна вообще, или сумма доставки будет высокой.

И наоборот, если Адрес покупателя находится в соседнем доме от ресторана, то доставка будет по минимальной стоимости.

Но если при этом время оформления и доставки заказа — в час-пик, то стоимость будет выше.

Если способ оплаты — «Наличные курьеру», но по факту покупатель не заплатил курьеру, то заказ не будет доставлен (отменен).

То же самое, если на банковской карте не хватило средств для оплаты заказа, то заказ может быть отменен. Или отложен как не оплаченный и пользователь сможет поменять способ оплаты. Реальный исход зависит от реализации.

Из менее очевидного:

Что если покупатель закажет очень много блюд? В 2-3 раза больше, чем влезет в курьерскую термосумку. Сработает ли ограничение в, допустим, 15кг на 1го курьера? Или системе придется искать 3х курьеров?

Как будет в таком случае спрогнозировано время доставки? Ведь все эти блюда нужно еще успеть приготовить.

Если покупатель закажет доставку из ресторана за минуту до закрытия и ресторан не успеет подтвердить заказ, заказ доставят? Или перенесут на другой день? Или отменят?

Такие вопросы следует уточнять у постановщика задачи, ПМа или аналитиков на вашем проекте и не пытаться угадать/придумать «правильную» логику самостоятельно.

Получается, что у каждого варианта использования может быть несколько вариантов завершения. При составлении тест-кейсов нужно убедиться, что мы учли все возможные варианты как входных, так и выходных данных.

Пишите в сценариях смысл действия без опоры на внешний вид

При написании сценариев важно опираться на смысл пользовательского действия, а не на интерфейс. Это позволит не переписывать их при каждом редизайне. Т.е. мы можем написать:

- Покупатель выбирает вид оплаты Картой.

И не стоит писать:

- Покупатель кликнул на серую кнопку «Visa/Mastercard» в правом нижнем углу формы.

Кнопка может поменять цвет и перестать быть серой.

В текущей версии приложения это кнопка, а потом ее поменяют на плитку/выпадашку/любой-другой контрол.

В десктопной версии термин «кликнуть» еще приемлем, в мобильной версии приложения это может быть тап, или свайп, или другое действие.

Кнопка может поменять название и расположение.

Более того, в мобильной версии приложения кнопка может быть не в правом нижнем углу, а просто внизу или вообще посередине экрана.

Т.е. при описании сценариев желательно не использовать:

- Цвет
- Форму
- Расположение на экране
- Точное название
- Вид элемента интерфейса

Общий алгоритм применения подхода

Чтобы составить тест-кейсы по фиче с помощью техники Use Cases нужно сделать следующее:

1. Определить, зачем и для кого фича:
    - Какие категории пользователей есть в вашей целевой аудитории (ЦА).
    - Зачем люди из каждой категории используют эту фичу.
    - Какие проблемы они решают с помощью нее.
2. Выписать, какие есть варианты использования фичи.
3. Для каждого варианта использования нужно определить:
    - Какие действующие лица участвуют в сценарии (включая само приложение, ботов и сторонние системы).
    - Основное направление варианта использования.
    - Альтернативные сценарии.
    - Возможные исключения, когда что-то пошло не так.

В результате мы получаем кучу пользовательских сценариев с предусловиями, шагами и ожидаемым результатом.

Профит от подхода для тестировщика

С помощью всех этих сценариев можно протестировать требования и прототипы интерфейса от дизайнера раньше, чем задача придет к разработчику. А значит можно предотвратить

большое количество переделок и недоделок. Мне в работе эта техника позволяет:

- Найти пробелы в бизнес-логике.
- Помочь дизайнеру и разработчику разобраться в том, как будет работать фича.
- Убедиться, что есть обработка всех возможных ошибок.
- Проверить, что пользователь может пройти сценарий от начала до конца.
- Выявить все неочевидные предусловия, варианты развития событий, возможные входные/выходные данные.

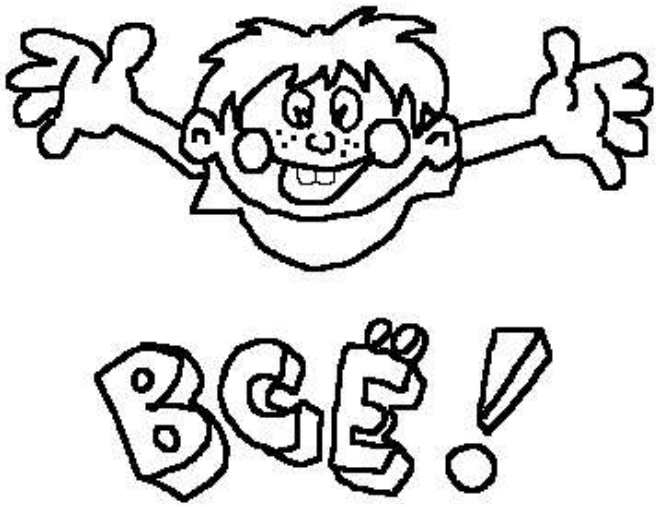
Заключение

Есть много техник, чтобы тестировать требования, и Use Cases, показанные в этой статье — это тоже не панацея от всех бед.

Техника легко ложится на фичи, где есть взаимодействие пользователя с продуктом. И если фича не предусматривает участие человека, но это интеграция двух или более систем, техника все еще применима. Просто действующие лица будут не «Пользователь» и «Система», а «Система 1» и «Система 2».

В то же время, подход не применим в задачах, где вообще нет никакого взаимодействия между сторонами.

Но не стоит забывать, что требования можно протестировать с помощью других инструментов, например, используя диаграмму «Сущность-Связь» (Entity-relationship), анализируя состояния объектов в продукте (State machine diagram) и т.д. Как при работе с требованиями, так и в тестировании все эти инструменты только усиливают друг друга, но это уже тема для отдельной статьи.



Хабравчане, о каких еще техниках по работе с требованиями вам было бы интересно узнать? Пишите в комментариях, поделюсь в следующих статьях.

Теги: тестирование, use cases, тестирование требований

Хабы: Блог компании Plesk, Тестирование IT-систем, Анализ и проектирование систем, Usability

plesk
Plesk
Plesk – панель управления хостингом


11 Карма 0 Рейтинг

Ольга Перевозкина @Olya_letsLED