PATTERNS

Наша задача — протестировать:

- операцию перевода денег с карты одного клиента на карту другого клиента через Интернет Банк
- операцию перевода денег с одной карты на другую одного и того же клиента через Интернет Банк

Начнём с уточняющих вопросов, обязательно стоило бы спросить:

- с какой карты и на какую производится перевод (дебетовая, кредитная или другая)?
- кому (в первом случае) - клиенту того же банка, другого?
- каким образом (в первом случае) - по номеру карты, ФИО или телефону (по ФИО и телефону можно, если это клиент того же банка*)?
- ряд других вопросов, специфичных для конкретного банка, например, в Альфа Банке можно запланировать время перевода (завтра).

Q: Но откуда я всё это могу знать, если даже мне ещё не показали интерфейс и я не произвёл тестовый

платёж (про описание логики вообще молчу)?

A: К сожалению, часто такое бывает, что вам нужно строить гипотезы и производить предварительную оценку в условиях неопределённости, когда доступа у вас ещё нет, а описание могут вообще не предоставить (например, вы работаете в сторонней компании и полного процесса вам не предоставят по соображениям безопасности).

Q: Но тогда я просто не возьмусь за работу!

A: Хорошая позиция, но она равносильна тому, что вы ждёте идеальных

условий для начала работы. А они могут никогда так и не наступить.

В подобных случаях используются два ключевых подхода:

- исследования;
- специализация.

Вы самостоятельно становитесь «клиентом» примеряя на себя ту роль, которую собираетесь тестировать.

Т.е. заказываете карту, становитесь клиентом, анализируете Интернет Банк.

При этом в процессе вы узнаёте много того, чего сам Заказчик и его Сотрудники могут не знать, т.к. находятся «по другую сторону стойки».

Естественно, не всегда этот подход возможен (чаще всего возможен только для публичных сервисов для физ.лиц) и вы должны согласовывать подобную активность с Заказчиком.

Специализация

Вы начинаете сознательно специализироваться на сервисах определённого типа и отслеживать их развитие.

С одной стороны, это сужает ваш профиль, с другой стороны вы как специалист в этой области начинаете цениться больше, т.к. ваша экспертиза выше.

Это как на стройке: есть разнорабочие, которые умеют всё, и есть узкопрофильные специалисты.

При этом, естественно, вы должны вкладывать своё время и усилия, чтобы постоянно углублять свои знания и быть в курсе того, как работают те системы, на которых вы специализируетесь, и куда движется их развитие.

Проверяем перевод денег с карты клиента (у которого одна дебетовая карта) на карту другого клиента того же банка по номеру телефона.

Лирическое отступление - но простор для выбора сценариев огромен:

- у клиента недостаточный баланс на карте;
- клиент ввёл отрицательную сумму перевода (минус на минус будет плюс!)*;
- карта клиента заблокирована;
- номер получателя введён неверно;
- получатель заблокирован;
- и другие варианты (например, превышен лимит перевода).

Happy Path - если клиент сделал всё правильно, то должен получить правильный результат.

Sad Path - если клиент сделал что-то неправильно, то должен получить корректное уведомление.

Evil Path - если клиент попытался "взломать систему", то система должна выстоять + необходимо уведомить соответствующие службы.

Обычно, рекомендуют сначала проверять Happy Path, поскольку именно он осуществляет выполнение задач клиента. Но это не всегда верно: например, в финансовых системах то, что работа функции перевода не приводит к «взлому системы» может оказаться гораздо важнее, чем её работоспособность. В рамках лекции мы с вами рассмотрим Happy Path.

Вернёмся к сценариям тестирования.

В целом, сценарий для перевода на карту другого клиента можно было бы спроектировать описанным ниже способом.

От лица первого пользователя:

- осуществляем процедуру входа в Интернет Банк (нужно ещё подтвердить кодом из SMS или PUSH);
- выбираем операцию перевода денег другому клиенту;
- вводим телефон другого клиента и сумму, нажимаем перевести (нужно ещё подтвердить кодом из SMS или PUSH);
- удостоверяемся, что на счёте нашей карты стало денег меньше (будем считать, что перевод без комиссии);
- выходим из Интернет Банка.

От лица второго пользователя:

- осуществляем процедуру входа в Интернет Банк (нужно ещё подтвердить кодом из SMS или PUSH);
- удостоверяемся, что на счёте нашей карты стало денег больше;
- выходим из Интернет Банка.

Для перевода денег между своими картами:

- осуществляем процедуру входа в Интернет Банк (нужно ещё подтвердить кодом из SMS или PUSH);
- выбираем операцию перевода денег с карты на карту;
- выбираем карты и сумму, нажимаем перевести (подтверждать кодом не нужно);
- удостоверяемся, что на счёте одной карты стало денег больше;
- удостоверяемся, что на счёте другой карты стало денег меньше;
- выходим из Интернет Банка.

И тут у нас возникает куча вопросов:

- откуда брать данные пользователей?
- что делать с SMS/PUSH?
- как устранить дублирующие операции?
- не будет ли интерфейс изменяться, если мы просто вышли из текущего пользователя (например, нам предложат ввести только пароль, но не логин)?
- а что если тестов будет несколько и мы будем использовать одних и тех же пользователей (тогда нам может не хватить баланса для перевода)?

Design Patterns (шаблоны проектирования) - это общеизвестный подход к решению определённой часто встречающейся проблемы.

Т.е. формулируется проблема, апробируются различные подходы к её решению, и одно из решений формулируется как паттерн.

Ключевое:

1. Если нет проблемы, то и паттерна нет (он просто не нужен);
2. Если нет решения, то и паттерна нет.

Мы фокусируемся на трёх областях приложения этих подходов:

1. Тестовые данные (наши пользователи, их карты);
2. Тестовая логика (авторизация, проверка баланса);
3. Интеграция с внешними системами (SMS/PUSH).

Ключевое: разделение этих частей и решение проблем каждой

При работе с тестовыми данными возникает два вопроса:

1. Откуда их брать;
2. Сколько их нужно;
3. Как их «положить» в SUT.

Важно: стоить запомнить, что идеального решения не существует, любое решение будет содержать как минусы, так и плюсы.

Есть несколько подходов:

- Взять реальные данные из продакшн системы;
- Захардкодить (использовать фиксированные данные);
- Сгенерировать (рандомно).

Для тестирования Happy Path лучше, конечно же, брать данные, максимально приближенные к реальным.

Т.к. именно реальные данные отражают самые часто используемые значения и параметры. И если система не работает на них - то всё очень плохо.

Кроме того, если реальных данных много, то можно попробовать избежать эффекта пестицида, рандомизировав сам выбор данных

Мы специально взяли для примера именно Интернет Банк, потому что вы прекрасно понимаете, что почти никогда вам не дадут данные реальных пользователей (слишком большие риски).

Максимум, только после процедуры анонимизации их можно будет использовать, но в результате этого вы как раз можете потерять отражение реальной статистики.

Кроме того, сам процесс их получения будет достаточно трудоёмким, т.к. их ещё нужно выгрузить из продакшн системы и передать в тесты.

А также нужно автоматизировать процесс их извлечения, хранения и передачи в тестовую систему, учитывая необходимость не создавать лишнюю нагрузку на продакшн систему.

Но это не значит, что не следует использовать реальные данные в тестах.

Наоборот, если такая возможность есть, обязательно старайтесь проверять на реальных данных.

Хардкодинг

В принципе, это то, что мы с вами использовали в предыдущих тестах.

Данные просто зашиваются в SUT и в наши тесты.Ключевой плюс: простота.

Ключевой минус: если SUT из себя представляет чёрный ящик, то мы не

можем из тестов или системы CI управлять этими данными*.

Генерация

Мы можем воспользоваться существующими решениями или самостоятельно написать генератор данных.

Из существующих решений мы можем использовать библиотеку Faker.

Подключается стандартным образом в Gradle:

```
...
dependencies {
    ...
    testImplementation 'com.github.javafaker:javafaker:1.0.1'
    ...
}
...
```

А дальше генерируем данные нужного типа, используя нужную нам локаль*:

```
private Faker faker;

@BeforeEach
void setUpAll() {
    faker = new Faker(new Locale("ru"));
}

@Test
void shouldPreventSendRequestMultipleTimes() {
    String name = faker.name().fullName();
    String phone = faker.phoneNumber().phoneNumber();
    String cardNumber = faker.finance().creditCard(CreditCardType.MASTERCARD);
    System.out.println(name);
    System.out.println(phone);
    System.out.println(cardNumber);
}
```

Важно понимать, что генерация не ограничена только возможностью генерировать данные в самих тестах

- это может быть организовано как угодно, главное, чтобы ваши тесты знали, как эти данные получить и ими воспользоваться.

Понятно, что генерировать поля по-одному не всегда удобно, и если вам нужен «целый» пользователь, то вы можете написать утилитный класс, который вам и будет генерировать подобные объекты:

```
public class DataGenerator {
    private DataGenerator() {}

    public static class Registration {
        private Registration() {}

        public static RegistrationByCardInfo generateByCard(String locale) {
            Faker faker = new Faker(new Locale("ru"));
            return new RegistrationByCardInfo(
                faker.name().fullName(),
                faker.finance().creditCard(CreditCardType.MASTERCARD),
                LocalDate.now().plusYears(1)
            );
        }
    }
}
```

Вы можете использовать Lombok для генерации подобных классов:

```
@Data
@RequiredArgsConstructor
public class RegistrationByCardInfo {
    private final String name;
    private final String card;
    private final LocalDate cardExpire;
}
```

Обратите внимание, что подобные объекты этих классы стараются делать иммутабельными (т.е. неизменяемыми) во избежание того, чтобы вы их случайно не изменили в тесте.

Но а дальше всё зависит только от сложности генерируемой вами системы: вы можете зашить любую логику генерации (например, генерировать карты с истёкшим сроком действия и т.д.).

При росте системы, количество тестов на переводы будет расти (ЖКХ,

услуги, покупки и т.д.) и нам постоянно придётся хранить сумму, превышающую общую потребность. А если мы захотим тестировать Sad Path - когда у клиента будет недостаточно средств для перевода? Тогда нужно просто прогонять тесты на недостаточный баланс после всех остальных? Чувствуете? Количество правил растёт: как требование к общей сумме,так и к порядку тестов.

Anti-pattern

Зависимость тестов по данным, как и зависимость тестов по последовательности исполнения - это ключевой анти-паттерн.

Если дизайн паттерн - общеизвестная проблема и чаще всего хорошее решение к ней, то анти-паттерн - общеизвестная проблема и плохое решение к ней.

Q: Почему зависимости между тестами плохо?

A: Потому что это приведёт к тому, что в разных комбинациях вы будете получать Fail только по причине того, что ваша SUT находится не в том состоянии, которого от неё ожидает тест. Например, тест на недостаточный баланс выполнился раньше, чем тест на перевод денег.

Кроме того, зависимость между тестами не позволит вам в дальнейшем выполнять эти тесты параллельно (чтобы их ускорить), а медленные тесты никто не любит.

В данном случае два самых часто применяемых подхода:

- сгенерировать столько данных, сколько нужно для независимости тестов по данных (каждому тесту - свои данные);
- запускать тесты так, чтобы даже если они используют одинаковые данные, то пересечения по данным не происходило*.

Как их положить в SUT

Всё зависит от того, какие вам возможности предоставили разработчики SUT:

1. Графический интерфейс (то же, через что тестируем);
2. API (например, отдельное REST API, позволяющее делать многие вещи быстрее);
3. Напрямую в СУБД или иное хранилище SUT;
4. Специальный «тестовый» режим SUT, при котором ей можно «скормить демо-данные»;
5. Другие варианты.

В современных системах Web UI строится на базе того API, которое используется мобильными клиентами, в более старых же версиях, Web UI «вшивался» в сам SUT.

С графическим интерфейсом всё достаточно сложно, т.к. чтобы «залить» тестовые данные, приходится писать отдельную логику.

Пример: мы можем воспользоваться интерфейсом регистрации, для того чтобы создать тестового пользователя.

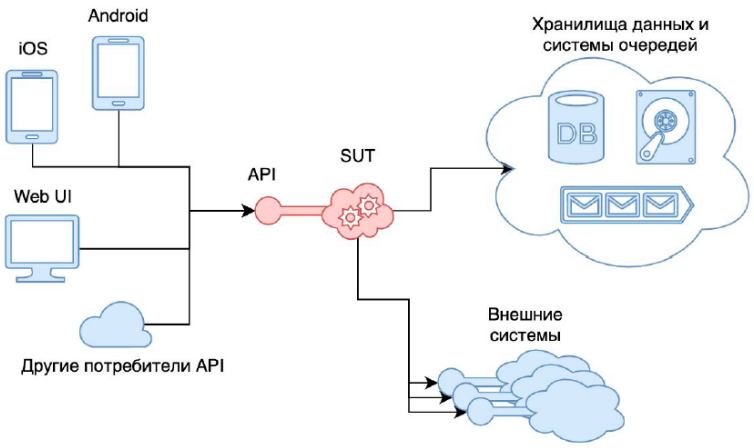
Но теперь вопрос, нужно ли тогда писать тест на регистрацию пользователя?

И самое главное (в контексте Интернет Банка): а как мы «положим»

деньги на счёт этому пользователю?

Типичная архитектура

Стоит нарисовать архитектуру, по которой построено большинство SUT:

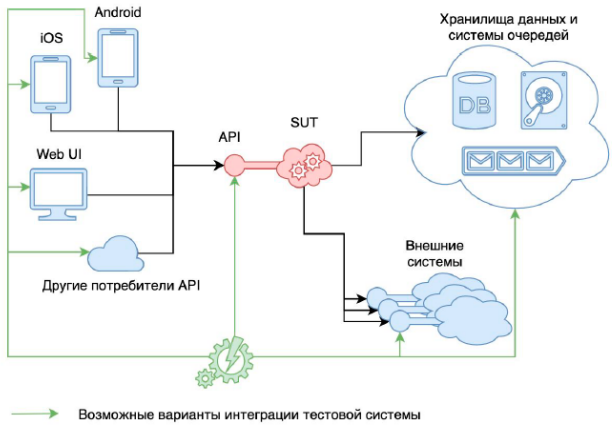


С графическим интерфейсом всё достаточно сложно, т.к. чтобы «залить» тестовые данные, приходится писать отдельную логику.

Пример: мы можем воспользоваться интерфейсом регистрации, для того чтобы создать тестового пользователя.Но теперь вопрос, нужно ли тогда писать тест на регистрацию пользователя?

И самое главное (в контексте Интернет Банка): а как мы «положим» деньги на счёт этому пользователю?

Мы, как автоматизаторы, можем взаимодействовать как через выделенные каналы: например, Web UI, так и через API, а, возможно, и напрямую со внешними системами.



Но в других сценариях это вполне допустимо: например, добавление комментария от анонимного пользователя на сайт или регистрация в менее «требовательном» сервисе.

В любом случае, пользоваться этим вариантом стоит только в отсутствии «более быстрых» альтернатив.

SUT может предоставлять интерфейс для программного взаимодействия

(API - Application Programming Interface).

Благодаря ему, вы можете программно «заносить» в SUT нужные данные.

Например, API социальной сети ВК позволяет вам программно создавать группы:

Но при этом методы создания пользователей через API не предоставляются (т.е. их придётся добавлять в SUT альтернативными методами).

Q: А что если создание группы - это функция, которую нужно протестировать?

A: обычно делают так: на создание группы пишется свой тест, в котором группа создаётся через UI.

Для других же тестов, в которых созданная группа является необходимыми данными, она создаётся через API.

Чаще всего программные сервисы не хранят данные в оперативной памяти. Для этого используются внешние системы, например, СУБД, файловые системы и другие хранилища.

Поэтому если у вас есть к ним доступ, вы можете напрямую из тестов записывать туда данные.При этом вы можете это делать в нужные вам моменты данных, организовывая своеобразный «refresh» состояния SUT.

Кроме того, в большинстве случаев - этот способ наиболее быстрый ипозволяет вам получить максимальный контроль над состоянием SUT.

Q: Если это самый лучший способ, то зачем нужны все остальные?

A: Это достаточно распространённый подход, но несёт в себе ряд сложностей, поскольку SUT уже не будет для вас «чёрным ящиком».

Вы должны достаточно хорошо знать внутреннее устройство SUT, модель хранения данных и взаимосвязи между данными, в противном случае своей записью в БД вы просто приведёте SUT в «неправильное» состояние - т.к. фактически, обходите все проверки SUT, взаимодействуя не через её интерфейс.

Кроме того, любое изменение в структуре СУБД, произведённое разработчиками, может «сломать» ваши тесты, при этом через UI и API всё будет по-прежнему работать.

Не совсем правильно говорить, что это отдельный подход - альтернатива UI, API или прямому доступу к хранилищу, скорее это возможность переключить SUT в такой режим, при котором вам станут доступны некоторые возможности, отсутствующие в продакшн варианте.

Например, при наличии флага запуска (определённой конфигурации, переменной окружения), SUT будет открывать дополнительное API для создания пользователей или загрузки их из файла при старте системы.

Таким образом, мы получаем для своих тестов необходимые нам возможности по управлению состоянием SUT:

- запись необходимых данных (например, пользователей и их баланса);
- чтение необходимых данных (например, кода, отправленного по SMS/PUSH);

К недостаткам можно отнести то, что этот режим нужно отдельно поддерживать и актуализировать, а это лишние ресурсы разработчиков.

Кроме того, нужно внимательно относиться к вопросам безопасности, поскольку включение подобного режима в продакшн среде приведёт к катастрофическим последствиям.

Также включение тестового режима может характеризоваться тем, что используются упрощённые варианты взаимодействия (встраиваемая СУБД, заглушки для внешних систем), таким образом, возрастает вероятность, что мы будем тестировать «тестовый режим», а не реальную систему.

Поэтому разработчики, специалисты по безопасности и некоторые тестировщики достаточно негативно относятся к этому варианту.

Одной из идей удобного управления состоянием SUT (похожим на тестовый) является идея сервисного доступа.

Когда обладая определённым кодом доступа (или токеном), или выполняя запросы с определённых доверенных IP-адресов, SUT предоставляет расширенные возможности через своё API.

Плюсы и минусы такие же, как и в тестовом режиме, за исключением:

- нет риска, что мы тестируем «тестовый режим»;
- выше риск, что сервисный доступ получат злоумышленники.

Внешние системы

При работе с внешними системами ключевой проблемой является получение доступа к ним.

Например, система отправки SMS - является внешней по отношению к Интернет Банку и мы не можем оттуда просто так получить значение кода подтверждения.

Варианты, которые у нас есть:

- заглушки для внешних систем;
- прокси для внешних систем;
- доступ к SUT:
  - СУБД или хранилище SUT (если SUT передаёт данные в систему рассылки SMS);
  - сервисный доступ SUT;
  - тестовый режим SUT (код может быть фиксированным и никакой отправки в систему рассылки не происходит);

- один из вариантов доступа к системе рассылки (такие же, как были рассмотрены для SUT);
- реальная интеграция;

Также как и везде, используется один из вариантов или их комбинация вместе с учётом минусов и плюсов каждого подхода

Заглушки для внешних систем

Это некий аналог Mock'ов, которые мы делали в Unit-тестах, но уже на уровне целых систем.

Пишутся упрощённые варианты сервисов, а из тестов уже обращаемся к ним.

При этом нужно понимать, что эти заглушки нужно будет:

1. Написать
2. Поддерживать
3. Запускать во время прогона тестов
4. Настраивать SUT на использование заглушек

Из плюсов: полностью контролируемое поведение и API для ваших тестов.

Proxy для внешних систем

Это механизм, при которым мы перехватываем запросы от SUT к реальным системам (при этом можем возвращать ответ как от реальной системы, так и предопределённый.

Достаточно удобно, если SUT интегрируется с внешними системами по распространённым протоколам, например, HTTP.

Единственное, что стоит упомянуть, если мы не используем тестовый режим SUT, то придётся поднимать заглушки или интегрироваться свнешними системами

Реальная интеграция

Наверное, лучший способ, если вы хотите действительно всё проверить целиком.

Но самый ресурсоёмкий, медленный и дорогостоящий.

Вам нужно будет:

1. Построить отдельную систему (либо купить), которая умеет принимать SMS;
2. Настроить ожидания в своих тестах - SMS не приходит мгновенно (а может вообще не дойти);
3. Платить за SMS при каждом прогоне тестов (т.к. отправка SMS - это платная услуга).

Но зато вы проверите функциональность системы от и до

Тестирование на Production

Иногда вместо того, чтобы проводить такие «реальные» тесты на тестовой системе, предпочитают их проводить сразу на боевой, чтобы понимать, что продакшн система в данный момент функционирует для самых критичных сценариев.

При этом заводятся реальные аккаунты, для которых проверяются эти самые критичные сценарии: авторизация, оплата ключевых услуг (погуглите ТОП используемых услуг Интернет Банка), перевод со счёта на счёт.

Это Must Have практика для сервисов, критичных для функционирования бизнеса.

Тестовая логика

Page Object

```
class LoginPage {
    public void login(AuthInfo info) {
        // здесь инкапсулирована вся логика работы со страницей логина
    }
}
```

```
// в тестах, в которых необходим логин:
loginPage.login(info);

// вместо того, чтобы каждый раз вызывать:
$('[data-id=login]').setValue(info.getLogin());
$('[data-id=password]').setValue(info.getPassword());
$('[data-action=login]').click();
// эта логика будет зашита в самом методе login
```

Именно его мы рассмотрим на следующей лекции.

Для тестовой логики тоже существуют свои паттерны, в частности, для тестирования Web UI используется паттерн Page Object.

Этот паттерн позволяет скрыть логику взаимодействия с конкретными элементами страницы (поиск по селектору, ожидания, заполнение полейи т.д.) за интерфейсом определённого объекта)