Приложение

Содержание

1 Обоснование актуальности

Сначала надо понять различия между ручными и автоматизированными тестами. Ручное тестирование проводится непосредственно человеком, который нажимает на кнопки в приложении или взаимодействует с программным обеспечением или API с необходимым инструментарием. Это достаточно затратно, так как это требует от тестировщика установки среды разработки и выполнения тестов вручную. Имеет место вероятность ошибки за счет человеческого фактора, например, опечатки или пропуска шагов в тестовом сценарии.

Автоматизированные тесты, с другой стороны, производятся машиной, которая запускает тестовый сценарий, который был написан заранее. Такие тесты могут сильно варьироваться в зависимости от сложности, начиная от проверки одного единственного метода в классе до отработки последовательности сложных действий в UI, чтобы убедиться в правильности работы. Такой способ считается более надежным, однако его работоспособность все еще зависит от того насколько скрипт для тестирования был хорошо написан.

Автоматизированные тесты – это ключевой компонент непрерывной интеграции (Continuous Integration) и непрерывной доставки (continuous delivery), а также хороший способ масштабировать ваш QA процесс во время добавления нового функционала для вашего приложения.

Во время ручного тестирования мы подготавливаем окружение (зависимости), запускаем приложение, вводим тестовые данные, проверяем работоспособность, вводим другие тестовые данные и так пока не проверим все возможные варианты входов и выходов, разумеется, если мы хотим, чтобы наше приложение работало. Все это занимает очень много времени и чем больше становиться приложение, тем сложнее выполнять этот процесс. В результате, программист практически перестает заниматься проверкой функциональности, особенно старой, что может привести к неожиданному

поведению программы, а возможно даже к критическим для бизнеса ошибкам и потерям денег и времени. Похожий сценарий может произойти с автоматизированными тестами плохого качества.

1.1 Цель юнит-тестирования

Главная цель юнит-тестирования – обеспечение стабильного роста программного проекта. В начале жизненного цикла проекта, его развивать довольно просто, но намного сложнее поддерживать развитие с прошествием времени

На рисунке 1 изображена динамика роста типичного проекта без тестов. Все начинается быстро, потому что ничего вас не тормозит. Еще не приняты неудачные архитектурные решения; еще нет существующего кода, который необходимо прорабатывать и поддерживать. Однако с течением времени вам приходится тратить все больше времени, чтобы написать тот же по объему функционал, что и в начале проекта. Со временем скорость разработки существенно замедляется — иногда даже до состояния, в котором проект вообще перестает двигаться вперед.

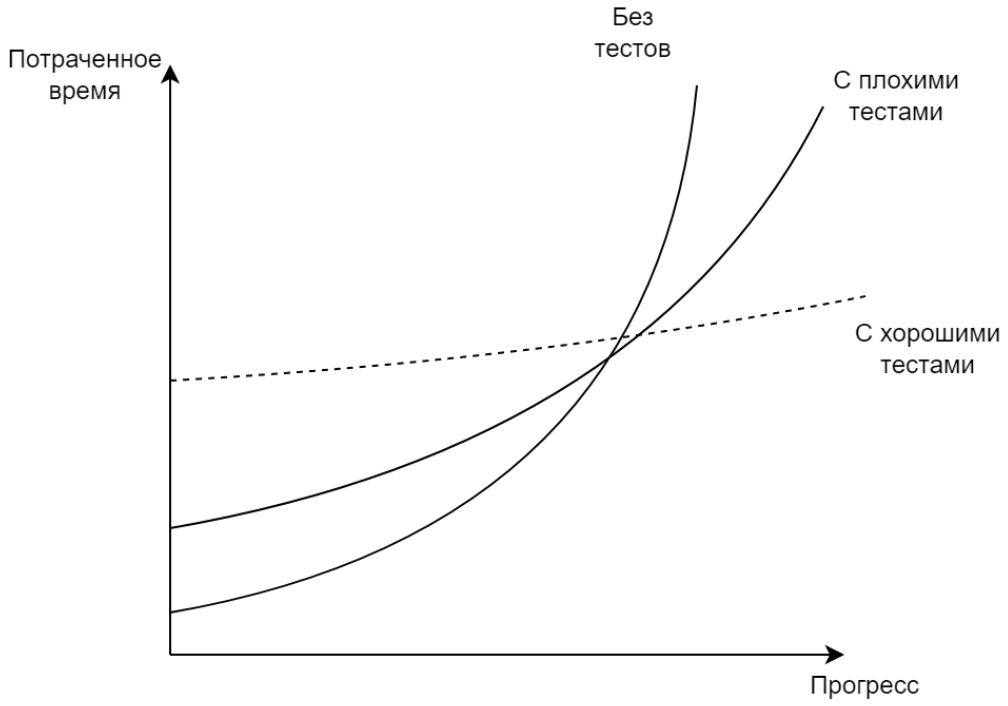

Рисунок 1 – Динамика роста проекта с тестами и без

Если не принять должных мер (например, постоянной чистки и рефакторинга), код постепенно усложняется и дезорганизуется. Исправление одной ошибки приводит к появлению новых ошибок, а изменение в одной части проекта нарушает работоспособность в нескольких других — возникает своего рода «эффект домино». Со временем код становится ненадежным. И что еще хуже, его становится все труднее вернуть в стабильное состояние. Тесты помогают справиться с этой тенденцией. Они становятся своего рода «подушкой безопасности» — средством, которое обеспечивает защиту против большинства регрессий. Тесты помогают удостовериться в том, что существующая функциональность работает даже после разработки новой функциональности или рефакторинга кода.

Недостаток юнит-тестирования заключается в том, что тесты требуют начальных вложений, и иногда весьма значительных. Но в долгосрочной перспективе они окупаются, позволяя проекту расти на более поздних стадиях. Разработка большинства нетривиального программного обеспечения без помощи тестов практически невозможна.

1.2 Связь между юнит-тестированием и структурой кода

Сама возможность покрытия кода тестами — хороший критерий определения качества этого кода, но он работает только в одном направлении. Это хороший негативный признак — он выявляет низкокачественный код с относительно высокой точностью. Если вдруг обнаружится, что код трудно протестировать, это верный признак того, что код нуждается в улучшении. Плохое качество обычно проявляется в сильной связности (tight coupling) кода; это означает, что части кода недостаточно четко изолированы друг от друга, что в свою очередь создает сложности с их раздельным тестированием. Но, к сожалению, возможность покрытия кода тестами является плохим позитивным признаком. Тот факт, что код проекта легко тестируется, еще не означает, что этот код написан хорошо. Качество кода может быть плохим даже в том случае, если он не страдает сильной связностью.

2 Обзор решения

2.1 Определение юнит-теста

Юнит-тестом называется автоматизированный тест, который:

- проверяет правильность работы небольшого фрагмента кода (также называемого юнитом);
- делает это быстро;
- и поддерживая изоляцию от другого кода;

Неоднозначность может вызывать первый и третий пункт. Вопрос изоляции и вопрос того, что является юнитом - это основное различие между лондонской и классической школой юнит-тестирования.

2.2 Структура юнит-теста

Юнит тест состоит из 3ех частей: Arrange – подготовка, Act – действие, Assert – проверка. Этот паттерн называется AAA или 3A.

```
@Test
public void test() {
    //Arrange
    var sut = new Calculator();
    //Act;
    var result = sut.sum(2, 2);
    //Assert
    assertEquals(4, result);
}
```

В секции подготовки тестируемая система (system under test, SUT) и ее зависимости приводятся в нужное состояние; Секция подготовки самая большая.

В секции действия вызываются методы SUT, передаются подготовленные зависимости и сохраняется выходное значение (если оно есть); Секция действия должна состоять из одной строки. Если она состоит из двух и более, это может указывать на проблемы с API тестируемой системы. Иногда это правило можно нарушить, но это относится не к бизнес-логике, а к служебному или инфраструктурному.

В секции проверки проверяется результат, который может быть представлен возвращаемым значением, итоговым состоянием тестируемой

системы и ее коллабораторов или методами, которые тестируемая система вызывает у этих коллабораторов.

2.3 Аспекты хорошего юнит-теста

Хороший юнит-тест должен обладать следующими тремя атрибутами:

- защита от багов;
- устойчивость к рефакторингу;
- быстрая обратная связь;

Они могут использоваться для анализа любых автоматизированных тестов, будь то юнит-, интеграционные или сквозные(end-to-end) тесты. В каждом тесте до той или иной степени проявляется каждый из атрибутов.

Первый аспект: защита от багов. Как правило, такие ошибки возникают после внесения изменений в код — обычно после написания новой функциональности.

Чтобы оценить, насколько хорошо тест проявляет себя в отношении защиты от багов, необходимо принять во внимание, следующее:

- объем кода, выполняемого тестом;
- сложность этого кода;
- важность этого кода с точки зрения бизнес-логики.

Как правило, чем больше кода тест выполняет, тем выше вероятность выявить в нем баг (если, конечно, он там есть). Само собой, тест также должен иметь актуальный набор проверок, просто выполнить код недостаточно. Важен не только объем кода, но и его сложность и важность с точки зрения бизнес-логики. Код, содержащий сложную бизнес-логику, важнее инфраструктурного кода — ошибки в критичной для бизнеса функциональности наносят наибольший ущерб. Как следствие, тестирование тривиального кода обычно не имеет смысла. Этот код слишком простой и не содержит сколько-нибудь значительного объема бизнес-логики. В тестах, покрывающих тривиальный код, вероятность нахождения ошибок невелика.

Примером тривиального кода служит однострочное свойство следующего вида:

```
public Storage getStorage() {
    return storage;
}

public void setStorage(Storage storage) {
    this.storage = storage;
}
```

Так же помимо вашего кода должны учитываться библиотеки и фреймворки. Для обеспечения оптимальной защиты тест должен проверять, как ваш код работает в комбинации с этими библиотеками, фреймворками и внешними системами.

Второй аспект: устойчивость к рефакторингу. Эта устойчивость определяет, насколько хорошо тест может пережить рефакторинг тестируемого им кода без выдачи ошибок. Например, вы провели рефакторинг нескольких участков кода, все выглядит лучше, чем прежде, но тесты не проходят. Новая функциональность работает так же хорошо, как и прежде. Это тесты были написаны так, что они падают при любом изменении тестируемого кода. И это происходит независимо от того, внесли вы ошибку в этот код или нет. Такая ситуация называется ложным срабатыванием. Это ложный сигнал тревоги: тест показывает, что функциональность не работает, тогда как в действительности все работает как положено. Такие ложные срабатывания обычно происходят при рефакторинге кода, когда вы изменяете имплементацию, но оставляете поведение приложения без изменений.

Почему столько внимания уделяется ложным срабатываниям? Потому что они могут и меть серьезные последствия для всего приложения. Целью юнит-тестирования является обеспечение устойчивого роста проекта. Устойчивый рост становится возможным благодаря тому, что тесты позволяют добавлять новую функциональность и проводить регулярный рефакторинг без внесения ошибок в код. Здесь имеются два конкретных преимущества:

– Тесты становятся системой раннего предупреждения при поломке существующей функциональности. Благодаря таким ранним предупреждениям вы можете устранить ошибки задолго до того, как ошибочный код будет развернут, где исправление ошибок потребует значительно больших усилий.

– Вы получаете уверенность в том, что изменения в вашем коде не приведут к багам. Без такой уверенности вы будете проводить гораздо меньше рефакторинга, что в свою очередь приведет к постепенному ухудшению качества кода проекта.

Ложные срабатывания негативно влияют на оба эти преимущества:

– Если тесты падают без веской причины, они притупляют вашу готовность реагировать на проблемы в коде. Со временем вы привыкаете к таким сбоям и перестаете обращать на них внимание. А это может привести к игнорированию настоящих ошибок.

– С другой стороны, при частых ложных срабатываниях вы начинаете все меньше и меньше доверять вашим тестам. Они уже не воспринимаются как что-то, на что вы можете положиться. Отсутствие доверия приводит к уменьшению рефакторинга, так как вы пытаетесь свести к минимуму потенциальные ошибки. Эта история типична для большинства проектов с хрупкими тестами. Сначала разработчики серьезно относятся к падениям тестов и стараются их починить. Через какое-то время они устают от того, что тесты постоянно поднимают тревогу, и все чаще игнорируют их. Рано или поздно наступает момент, когда в работу попадают настоящие ошибки, так как разработчики проигнорировали их вместе сложными срабатываниям.

Количество ложных срабатываний, выданных тестом, напрямую связано со структурой этого теста. Чем сильнее тест связан с деталями имплементации тестируемой системы, тем больше ложных срабатываний он порождает. Уменьшить количество ложных срабатываний можно только одним способом: отвязав тест от деталей имплементации тестируемой системы. Тест должен

проверять конечный результат — наблюдаемое поведение тестируемой системы, а не действия, которые она совершает для достижения этого результата. Тесты должны подходить к проверке SUT с точки зрения конечного пользователя и проверять только результат, имеющий смысл для этого пользователя.

Пример неудачного теста:

```
@Test
public void test_inventory_implementation(){
    var storage = new Storage();
    storage.inventory.put(Item.WOOD, 10);

    Order order = customer.createOrder(storage,Item.WOOD,5);

    assertTrue(storage.inventory instanceof HashMap<Item,
Integer>);
    assertOrderCorrect(customer,storage,Item.WOOD,5,order);
    assertEquals(5,storage.inventory.get(Item.WOOD));
}
```

В этом тесте мы очень сильно привязываемся к структуре и деталям Storage. В случае, если мы захотим поменять хранилище и хранить не в ОЗУ, а в БД, нам придется переписывать данный тест. Он завязан на реализацию Storage, а не на наблюдаемое поведение. Хороший тест должен проверять, верен ли результат.

Пример более удачного теста:

```
@Test
public void orderItem_Enough() {
    var storage = new Storage();
    storage.addItems(Item.WOOD, 3);

    var order = customer.createOrder(storage, Item.WOOD, 1);

    assertOrderCorrect(customer, storage,Item.WOOD,1, order);
    assertEquals(2, storage.getQuantity(Item.WOOD));
}
```

В этом тесте проверяются выходные данные, состояние системы и не затрагиваются детали реализации. Данный тест более устойчив к рефакторингу.

Типы ошибок показаны на рисунке 2.

Типы ошибок		Функциональность	
		Работает правильно	Работает неправильно
Тест	проходит	Истинное отрицательное срабатывание	Ошибка II рода (ложноотрицательное срабатывание)
	не проходит	Ошибка I рода(ложное срабатывание)	Истинное срабатывание

Рисунок 2 – Типы ошибок

Третий аспект: быстрая обратная связь. Быстрая обратная связь является одним из важнейших свойств юнит-теста. Чем быстрее работают тесты, тем больше их можно включить в проект и тем чаще вы их сможете запускать. Быстро выполняемые тесты сильно ускоряют обратную связь. В идеальном случае тесты начинают предупреждать вас об ошибках сразу же после их внесения, в результате чего затраты на исправление этих ошибок уменьшаются почти до нуля. С другой стороны, медленные тесты увеличивают время, в течение которого ошибки остаются необнаруженными, что приводит к увеличению затрат на их исправление. Дело в том, что медленные тесты отбивают у разработчика желание часто запускать их, поэтому в итоге он тратит больше времени, двигаясь в ошибочном направлении.

К сожалению, создать идеальный тест невозможно. Дело в том, три атрибута являются взаимоисключающими. Невозможно довести их до максимума одновременно: одним из трех придется пожертвовать для максимизации двух остальных. Более того выдержать баланс еще сложнее. Нельзя просто обнулить один атрибут, чтобы сосредоточиться на остальных. Тест с нулевым значением в одной из трех категорий бесполезен. Следовательно, атрибуты нужно максимизировать так, чтобы ни один из них не падал слишком низко.

При максимизации двух атрибутов за счет третьего, возникает следующая картина (рисунок 3).

Рисунок 3 – Максимизация двух атрибутов

Сквозные тесты рассматривают систему с точки зрения конечного пользователя. Они обычно проходят через все компоненты системы, включая пользовательский интерфейс, базу данных и внешние приложения. Тем не менее наряду с преимуществами у сквозных тестов имеется крупный недостаток: они очень медленные.

В отличие от сквозных тестов, тривиальные тесты предоставляют быструю обратную связь. Кроме того, вероятность ложных срабатываний также мала, поэтому они обладают хорошей устойчивостью к рефакторингу. Тем не менее тривиальные тесты вряд ли смогут выявить какие-либо ошибки, потому что покрываемый ими код слишком прост. (сеттеры)

Также достаточно легко написать тест, который работает быстро и хорошо выявляет ошибки в коде, но делает это с большим количеством ложных срабатываний. Такие тесты называются хрупкими: они падают при любом рефакторинге тестируемого кода независимо от того, изменилась тестируемая ими функциональность или нет.

2.4 Результаты поиска хорошего юнит-теста

Выдержать баланс между атрибутами хорошего теста сложно. Тест не может иметь максимальных значений в каждой из первых трех категорий; а значит, придется идти на компромиссы. Рекомендуется максимизировать

устойчивость к рефакторингу, так как тест либо устойчив, либо нет, а другие атрибуты можно варьировать, получив три вида тестов (рисунок 4).

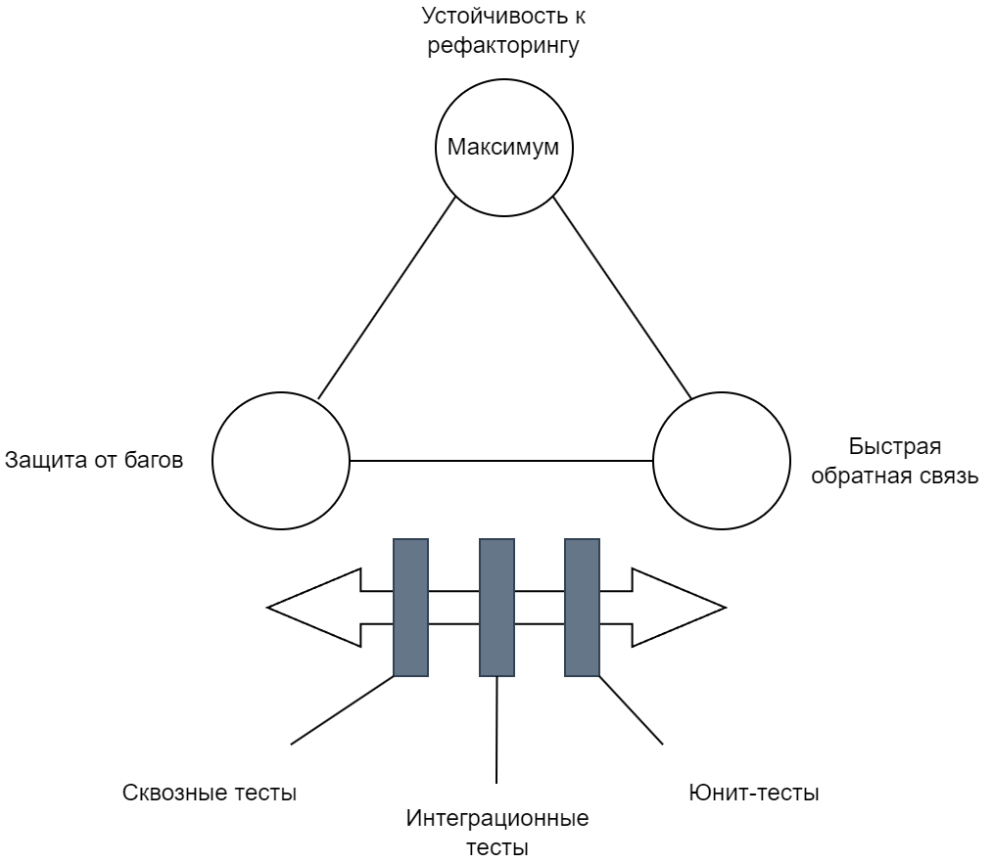

Рисунок 4 – Три вида тестов

Ни один из уровней не делает устойчивость к рефакторингу предметом для компромисса.

Соотношение количества тестов (рисунок 5).

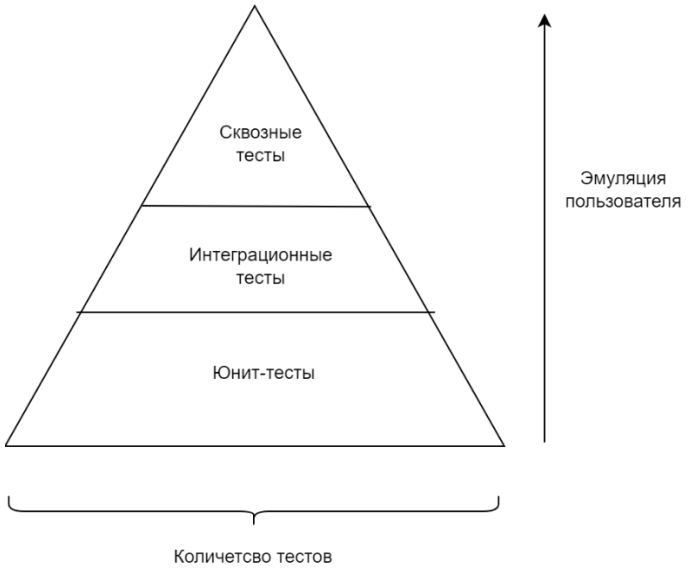

Рисунок 5 – Соотношение количества тестов

Как правило в приложении должно поддерживаться отношение, как на рисунке 6. Однако, оно справедливо не для всех. Если ваше приложение является CRUD скорее пирамида будет напоминать прямоугольник с юнит и интеграционными тестами Юнит-тесты менее полезны в ситуациях, в которых отсутствует алгоритмическая или бизнес сложность, — они быстро вырождаются в тривиальные тесты. В то же время интеграционные тесты полезны даже в таких случаях; каким бы простым код ни был, важно проверить, как он работает в интеграции с другими подсистемами (например, базой данных). В результате в CRUD-приложениях у вас будет меньше юнит-тестов и больше интеграционных. В самых тривиальных случаях интеграционных тестов может быть даже больше, чем юнит-тестов.

3 Выбор подходов к решению

3.1 Лондонская школа

Подход лондонской школы подразумевает изоляцию тестируемого кода, от его зависимостей (коллабораторов). То есть все зависимости класса нужно заменить на тестовые заглушки. Тестовая заглушка – объект, который имеет такой же интерфейс, как и заменяемый, но ведет себя так, как нам нужно для тестирования. Разновидности и создание тестовых заглушек рассмотрено позже. Под юнитом в лондонской школе подразумевается один класс или методов.

3.2 Классический подход

В классическом подходе изолируются друг от друга не классы, а тесты. То есть последовательность выполнения не влияет на результат. Тестируемые классы не должны обращаться к общему состоянию, через которое они могу влиять друг на друга. Примерами таких зависимостей являются БД и файловая система.

При использовании классического подхода юнит не обязан ограничиваться классом.

3.3 Сравнение лондонской и классической школы

В лондонской школе, происходит изоляция юнитов, юнит – это класс и заглушки используются для коллабораторов. Допускается не использовать заглушки для объектов без изменяемого состояния (объектов-значений)

В классической школе, изолируют юнит-тесты, юнит – это класс или набор классов, а заглушки используются для совместных зависимостей.

Иерархия зависимостей (рисунок 6).

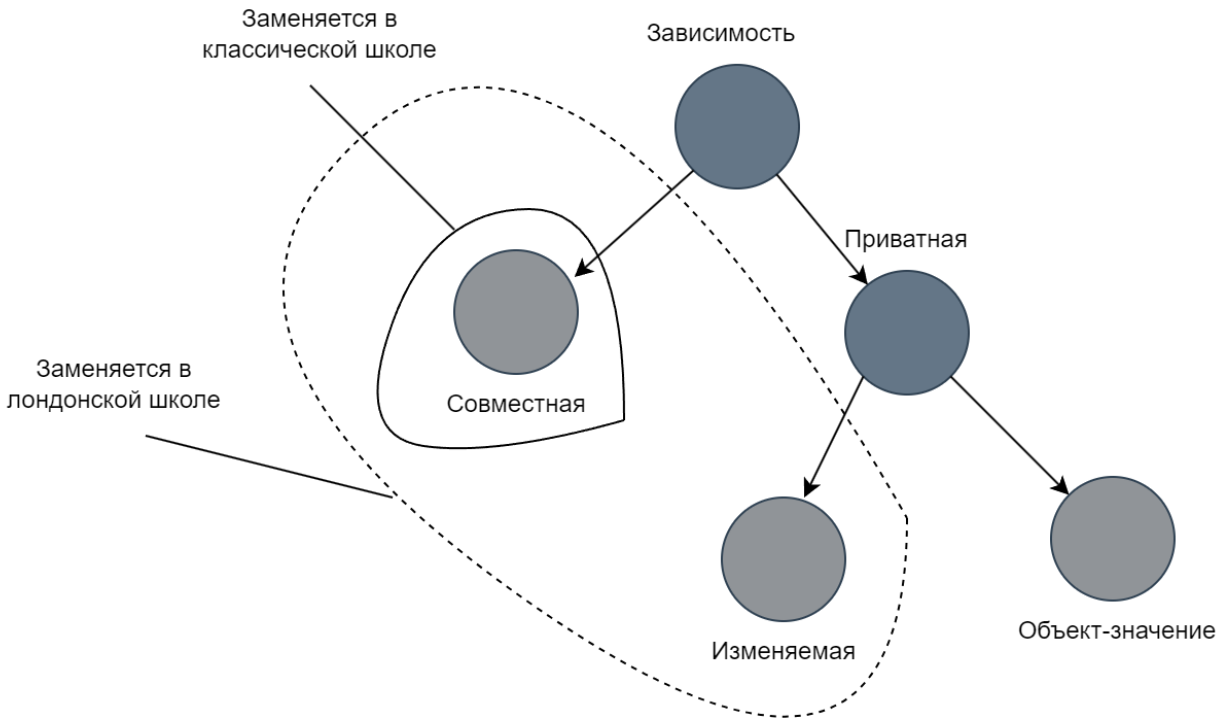

Рисунок 6 – Иерархия зависимостей

Лондонская школа обладает следующими преимуществами:

– Улучшенная детализация. Тесты высоко детализированы и проверяют только один класс за раз.

– Упрощение юнит-тестирования большого графа взаимосвязанных классов. Так как все коллабораторы заменяются тестовыми заглушками, вам не придется беспокоиться о них при написании теста.

– Если тест падает, вы точно знаете, в какой функциональности произошел сбой. Так как все коллабораторы заменены на заглушки, не может быть других подозреваемых, кроме самого тестируемого класса. Конечно, все еще возможны ситуации, в которых тестируемая система использует объект-значение, и изменение в этом объекте-значении приводит к падению теста. Однако такие случаи встречаются не так часто, потому что все остальные зависимости устранены в тестах.

Остаются еще два различия между классической и лондонской школами:

– подход к проектированию системы на базе методологии разработки через тестирование (TDD, Test-Driven Development);

– проблема излишней спецификации (over-specification);

Лондонский стиль юнит-тестирования ведет к методологии TDD по схеме «снаружи внутрь» (outside-in): вы начинаете с тестов более высокого уровня, которые задают ожидания для всей системы. Используя заглушки, вы указываете, с какими коллабораторами система должна взаимодействовать для достижения ожидаемого результата. Затем вы проходите по графу классов, пока не реализуете их все. Заглушки делают этот процесс разработки возможным, потому что вы можете сосредоточиться на одном классе за раз. Вы можете отсечь всех коллабораторов тестируемой системы и такимобразом отложить реализацию этих коллабораторов.

Классическая школа такой возможности не дает, потому что вам приходится иметь дело с реальными объектами в тестах. Вместо этого обычно используется подход по схеме «изнутри наружу» (inside-out). В этом стиле вы начинаете с модели предметной области, а затем накладываете на нее дополнительные слои, пока программный код не станет пригодным для конечного пользователя. Но, пожалуй, самое принципиальное различие между школами — проблема излишней спецификации (over-specification), то есть привязки теста к деталям имплементации тестируемой системы. Лондонский стиль приводит к тестам, завязанным на детали имплементации, чаще, чем классический стиль. И это становится главным аргументом против повсеместного использования заглушек и лондонского стиля в целом.

Пример. Есть покупатель и склад. Покупатель может заказать на складе товар, в результате чего создастся заказ и с инвентаря склада отнимется нужное количество предметов:

```
class Customer {
    public Order createOrder(Storage storage, Item item, int
quantity) {
        if (storage.hasEnoughItems(item, quantity))
        {
            storage.removeQuantity(item,  quantity);
            return new Order(storage, item, this, quantity);
        }
        throw new NotEnoughItems();
    }
```

```
}

class NotEnoughItems extends RuntimeException {}

class Storage {
    Map<Item, Integer> inventory;

    public Storage() {
        inventory = new HashMap<>();
    }

    boolean hasEnoughItems(Item item, int quantity) {
        return inventory.get(item) >= quantity;
    }

    public void addItems(Item item, int quantity) {
        if (inventory.containsKey(item))
            inventory.put(item, inventory.get(item) + quantity);
        else
            inventory.put(item, quantity);
    }

    public int getQuantity(Item item) {
        return inventory.get(item);
    }

    public void removeQuantity(Item item, int quantity) {
        inventory.put(item, inventory.get(item) - quantity);
    }
}

@Getter
@AllArgsConstructor
class Order {
    private final Storage storage;
    private final Item item;
    private final Customer customer;
    private final int Quantity;
}

enum Item {
    WOOD
}
```

Тесты:

```
@BeforeEach
void setUp() {
    customer = new Customer();
}
```

Классическая школа:

```
@Test
public void orderItem_NotEnough() {
    var storage = new Storage();
```

```
    storage.addItems(Item.WOOD, 1);

    assertThrows(NotEnoughItems.class, () ->
customer.createOrder(storage, Item.WOOD, 2));

    assertEquals(1, storage.getQuantity(Item.WOOD));
}

@Test
public void orderItem_Enough() {
    var storage = new Storage();
    storage.addItems(Item.WOOD, 3);

    var order = customer.createOrder(storage, Item.WOOD, 1);

    assertOrderCorrect(customer, storage,Item.WOOD,1, order);
    assertEquals(2, storage.getQuantity(Item.WOOD));
}
```

Лондонская школа:

```
@Test
public void london_orderItem_Enough(){
    var mock = mock(Storage.class);
    when(mock.hasEnoughItems(Item.WOOD,1)).thenReturn(true);

    var order = customer.createOrder(mock,Item.WOOD,1);

    assertOrderCorrect(customer, mock,Item.WOOD,1, order);
    verify(mock).removeQuantity(Item.WOOD,1);
}

@Test
public void london_orderItem_NotEnough(){
    var mock = mock(Storage.class);
    when(mock.hasEnoughItems(Item.WOOD,1)).thenReturn(false);

    assertThrows(NotEnoughItems.class, () ->
customer.createOrder(mock, Item.WOOD, 1));

    verify(mock,never()).removeQuantity(Item.WOOD,1);


}

private static void assertOrderCorrect(Customer customer,
Storage storage,Item item, int quantity, Order order) {
    assertEquals(item, order.getItem());
    assertEquals(quantity, order.getQuantity());
    assertEquals(customer, order.getCustomer());
    assertEquals(storage, order.getStorage());
}
```

3.4 Стили юнит-тестирования

Можно выделить три стиля юнит-тестирования:

- проверка входных данных;
- проверка состояний;
- проверка взаимодействий;

Проверка входных состояний – стиль юнит-тестирования, при котором в тестируемую систему подаются входные данные, после чего проверяется полученный результат. Этот стиль применим к коду, который не меняет глобального или внутреннего состояния, не имеет побочных эффектов. В проверке нуждается только его возвращаемое значение (рисунок 7).

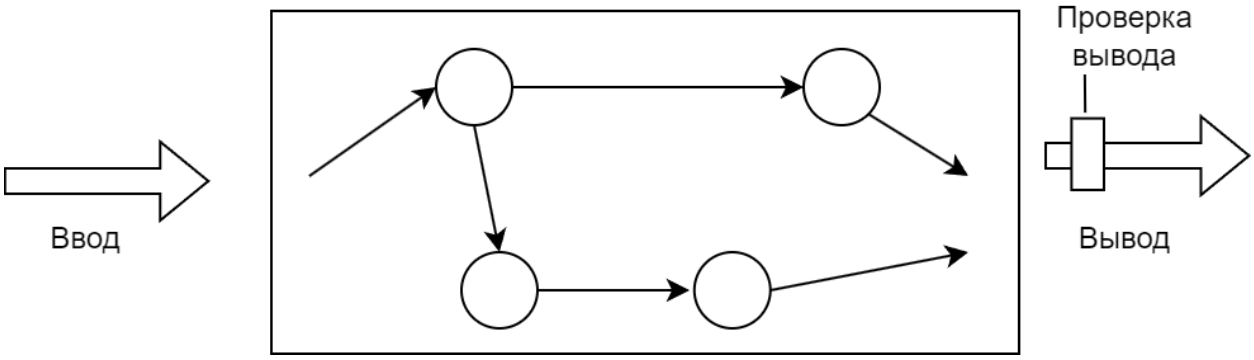

Рисунок 7 – Структура теста с проверкой выходного значения

Пример. Необходимо удалить из строки все, кроме букв:

```
class NameNormalizer {
    public String normalize(String name) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < name.length(); i++) {
            if (Character.isAlphabetic(name.charAt(i)) ||
name.charAt(i) == ' ')
                sb.append(name.charAt(i));
        }

        return sb.toString().trim();
    }
}

@Test
public void normalize_russian(){
    var sut = new NameNormalizer();

    var result = sut.normalize("Абв вбаф @ --112");

    assertEquals("Абв вбаф",result);
}
```

```
@Test
public void normalize_english(){
    var sut = new NameNormalizer();

    var result = sut.normalize(":;14@ :D asf %212");

    assertEquals("D asf",result);
}
```

Проверка выходных данных так же называется функциональным стилем юнит-тестирования.

Проверка состояния – стиль при котором тест проверяет состояние системы, после завершения операции (рисунок 8).

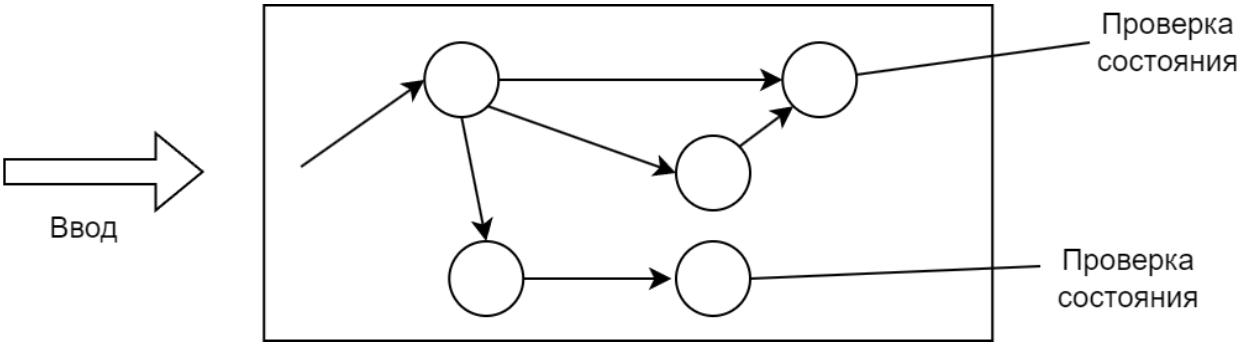

Рисунок 8 – Структура теста с проверкой состояния

Пример:

```
@Test
public void storage_remove_inventory(){
    var storage = new Storage();
    storage.addItems(Item.WOOD, 500);

    storage.removeQuantity(Item.WOOD, 125);

    assertEquals(375, storage.getQuantity(Item.WOOD));
}
```

Проверка взаимодействий – стиль юнит-тестирования при котором производится проверка взаимодействий между тестируемой системой и ее коллабораторами (рисунок 9).

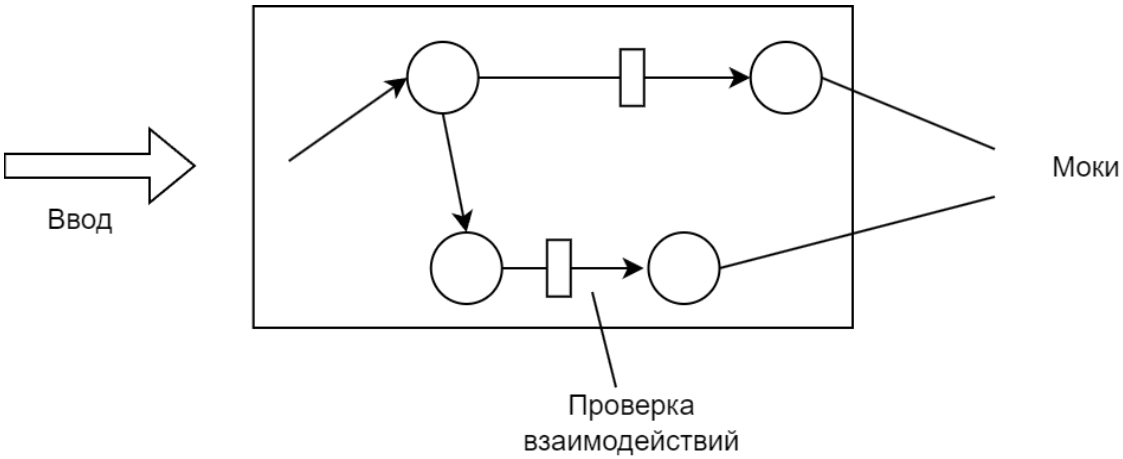

Рисунок 9 – Структура теста с проверкой взаимодействия

```
Пример:
@Test
public void get_item_quantity(){
    var stub = mock(ItemPort.class);
    var sut = new GetItemQuantityUseCase(stub);
    when(stub.getItemQuantity(Item.WOOD)).thenReturn(10);

    var quantity = sut.execute(Item.WOOD);

    assertEquals(10, quantity);
}
```

Классическая школа юнит-тестирование предпочитает проверку состояния, а не проверку взаимодействий. Лондонская школа делает противоположный выбор.

3.5 Сравнение стилей юнит тестирования

Сравним стили по четырем атрибутам:

- защита от багов;
- устойчивость к рефакторингу;
- быстрая обратная связь;
- простота поддержки

Защита от багов:

Эта метрика не зависит от конкретного стиля юнит-тестирования и является производной от трех характеристик:

- объем кода, выполняемого в ходе теста;
- сложность этого кода;
- важность этого кода с точки зрения бизнес-логики

Вы можете писать тесты, которые выполняют произвольный объем кода, никакой конкретный стиль не имеет преимуществ в этой области. То же относится к сложности кода и важности предметной области. Единственным исключением является проверка взаимодействий: злоупотребления могут привести к появлению поверхностных тестов, которые проверяют тонкий срез кода и заменяют все остальное моками. Но эта поверхностность не является определяющей особенностью тестирования взаимодействий, это скорее крайний случай злоупотребления этим стилем тестирования.

Быстрота обратной связи:

Между стилями тестирования и быстротой обратной связи тоже корреляции нет. При условии, что ваши тесты не притрагиваются к внепроцессным зависимостям и, следовательно, остаются в области юнит-тестирования (а не интеграционного тестирования), все стили приводят к тестам с приблизительно одинаковой скоростью выполнения. Проверка взаимодействий может показывать чуть худшие результаты, потому что моки обычно создают дополнительную задержку во время выполнения. Впрочем, различия обычно пренебрежимо малы, если только количество тестов не измеряется десятками тысяч.

Устойчивость к рефакторингу:

Что касается метрики устойчивости к рефакторингу, ситуация выглядит иначе. Устойчивость к рефакторингу показывает количество ложных срабатываний (ложных сигналов тревоги), выдаваемых тестами при рефакторинге рабочего кода. Ложные срабатывания, в свою очередь, являются результатом привязки тестов к деталям имплементации вместо наблюдаемого поведения

Проверка выходных данных обеспечивает наилучшую защиту от ложных срабатываний, потому что получаемые тесты завязываются только на тестируемый метод. Проверка состояния обычно в большей степени подвержена ложным срабатываниям. Кроме тестируемого метода, такие тесты также работают с состоянием класса. С вероятностной точки зрения, чем

больше связей между тестом и проверяемым им кодом, тем больше вероятность того, что этот тест окажется завязанным на детали имплементации, раскрытые в результате неверно выстроенного API. Тесты, проверяющие состояние, связываются с большей поверхностью API — следовательно, и вероятность завязывания их на детали имплементации тоже выше. Проверка взаимодействий в наибольшей степени подвержена ложным срабатываниям. Многие тесты, проверяющие взаимодействия с тестовыми заглушками, в конечном итоге оказываются хрупкими. Моки хороши только тогда, когда они проверяют взаимодействия, выходящие за границу приложения, и только когда результат этих взаимодействий виден внешнему миру.

Простота поддержки:

Метрика простоты поддержки сильно зависит от стиля юнит-тестирования, но, в отличие от устойчивости к рефакторингу, вы мало что можете с этим сделать. Простота поддержки оценивает затраты на сопровождение юнит-тестов и определяется следующими двумя характеристиками:

– насколько трудно понять тест (зависит от размера теста);

– насколько трудно запустить этот тест (зависит от количества внепроцессных зависимостей, с которыми работает тест);

Большие тесты сложнее поддерживать, потому что их труднее понимать и изменять. Аналогичным образом тест, который работает напрямую с одной или несколькими внепроцессными зависимостями (например, базой данных), создает больше проблем с сопровождением, потому что вам приходится тратить время на поддержание этих внепроцессных зависимостей в работоспособном состоянии: перезапускать сервер базы данных, решать проблемы с сетевым подключением и т. д

3.6 Тестовые заглушки

Тестовая заглушка (test double) — общий термин, который описывает все разновидности фиктивных зависимостей в тестах. В действительности все разновидности можно разделить на два типа: моки и стабы

Различия между двумя типами сводятся к следующему:

– Моки помогают эмулировать и проверять выходные взаимодействия — то есть вызовы, совершаемые тестируемой системой к ее зависимостям для изменения их состояния.

– Стабы помогают эмулировать входные взаимодействия — то есть вызовы, совершаемые тестируемой системой к ее зависимостям для получения входных данных.

Однако, не все внепроцессные зависимости следует заменять моками и стабами. Если внепроцессная зависимость доступна только через ваше приложение, взаимодействие с ней не является частью наблюдаемого поведения. Она является частью вашего приложения.

Пример — база данных приложения, то есть база данных, которая используется только вашим приложением и к которой не имеют доступа внешние системы. Вы можете изменить формат взаимодействия между вашей системой и базой данных приложения так, как считаете нужным, при условии, что это не нарушит существующей функциональности. Так как база данных полностью скрыта от клиентов вашего приложения, ее можно даже полностью заменить другой базой данных — никто этого не заметит.

Использование моков для внепроцессных зависимостей, находящихся под вашим полным контролем, также делает тесты хрупкими. Тесты не должны падать каждый раз, когда вы разбиваете таблицу на две или изменяете тип одного из параметров хранимой процедуры. База данных и приложение должны рассматриваться как единая система.

В таком случае классы, которые используют базу данных, должны покрываться только интеграционными тестами, причем использовать только

ту СУБД, которая используется в приложении. Это позволит проверять миграции и особенности БД.

Пример использования заглушек:

```
interface EmailPort{
    void sendInviteEmail(String email);
}
interface ItemPort{
    int getItemQuantity(Item item);
}
@RequiredArgsConstructor
class GetItemQuantityUseCase{
    private final ItemPort port;
    public int execute(Item item){
        return port.getItemQuantity(item);
    }
}
@RequiredArgsConstructor
class SendInviteEmailUseCase{
    private final EmailPort port;
    public void execute(String email){
        port.sendInviteEmail(email);
    }
}
```

Тесты:

```
@Test
public void send_invite_email(){
    var mock = mock(EmailPort.class); //Мок для выходных
    var sut = new SendInviteEmailUseCase(mock);

    sut.execute("any@mail.ru");

    verify(mock).sendInviteEmail("any@mail.ru");
}
@Test
public void get_item_quantity(){
    var stub = mock(ItemPort.class); //Стаб для входных
    var sut = new GetItemQuantityUseCase(stub);
    when(stub.getItemQuantity(Item.WOOD)).thenReturn(10);

    var quantity = sut.execute(Item.WOOD);

    assertEquals(10, quantity);
}
```

В данном случае тестируемый код лишен бизнес-сложности, но помимо отправки и получения может выполнятся еще какая-либо бизнес логика.

4 Средства для реализации

4.1 Использование jUnit 5

Для использования jUnit5 в проекте необходимо подключить следующие зависимости:

- junit-jupiter-engine-5.9.2.jar;
- junit-jupiter-api-5.9.2.jar;

Для запуска тестов через код: junit-platform-launcher-1.9.2.jar.

Код для запуска тестов вручную:

```
SummaryGeneratingListener listener = new
SummaryGeneratingListener();

public void runAll(){
    LauncherDiscoveryRequest request =
LauncherDiscoveryRequestBuilder.request()
            .selectors(selectPackage("nik.test"))
           .filters(includeClassNamePatterns(".*Test")).build();
    Launcher launcher = LauncherFactory.create();
    TestPlan test = launcher.discover(request);
    launcher.registerTestExecutionListeners(listener);
    launcher.execute(request);
}
```

Мы создаем запрос на поиск тестовых методов, создаем план для запуска тестов, запускаем и записываем результат в SummaryGeneratingListener. Результат можно вывести при помощи PrintWriter.

Для интеграции запуска тестов в жизненный цикл проекта с maven необходимо добавить плагин в pom.xml:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0</version>
</plugin>
```

Для того чтобы плагин и IDE корректно обнаруживал и запускал тесты они должны находиться в папке src/test/java.

Следующий пример дает представление о минимальных требованиях для написания теста в JUnit Jupiter:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator; import
org.junit.jupiter.api.Test;
```

```
class MyFirstJUnitJupiterTests {
	private final Calculator calculator = new Calculator();

	@Test
	void addition() {
		assertEquals(2, calculator.add(1, 1));
	}
}
```

Аннотация @Test помечает метод, как тестовый метод. Этот метод будет добавлен в контейнер и запущен во время тестирования.

Для проверок утверждений jUnit содержит библиотеку jUnit Assertions. Она содержит статические методы для проверки на равенство, проверки булевых значений, проверку на выброс исключения, проверку на null значения. Так же существуют библиотеки для написания более читабельных и удобных утверждений (Fluent Assertions, Hamcrest Assertions). Для конфигурации тестов используются следующие аннотации:

Таблица 1 – Аннотации jUnit

Аннотация	Описание
`@Test`	Означает, что метод является тестовым методом.
@ParameterizedTest	Означает, что метод является параметризованным тестом.
@RepeatedTest	Означает, что метод является тестовым шаблоном для повторного теста.
@TestFactory	Означает, что метод является тестовой фабрикой для динамических тестов.
@TestTemplate	Означает, что метод является шаблоном для тестовых примеров, предназначенных для многократного вызова в зависимости от количества контекстов вызова, возвращаемых зарегистрированными поставщиками.
@TestClassOrder	Используется для настройки порядка выполнения тестового класса для @Nested тестовых классов в аннотированном тестовом классе.
@TestMethodOrder	Используется для настройки порядка выполнения тестового метода для аннотированного тестового класса.

@DisplayName	Объявляет пользовательское отображаемое имя для тестового класса или метода тестирования.
@DisplayNameGeneration	Объявляет пользовательский генератор отображаемых имен для тестового класса.
@BeforeEach	Означает, что аннотированный метод должен выполняться перед каждым методом @Test, @RepeatedTest, @ParameterizedTest или @TestFactory.
@AfterEach	Обозначает, что аннотированный метод должен выполняться после каждого метода @Test, @RepeatedTest, @ParameterizedTest или @TestFactory в текущем классе.
@BeforeAll	Означает, что аннотированный метод должен выполняться перед всеми методами @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе.
@AfterAll	Означает, что аннотированный метод должен выполняться после всех методов @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе.
@Nested	Означает, что аннотированный класс является нестатическим вложенным тестовым классом.
@Tag	Используется для объявления тегов для фильтрации тестов либо на уровне класса, либо на уровне метода.
@Disabled	Используется для отключения тестового класса или метода тестирования.
@Timeout	Используется для сбоя теста, фабрики тестов, шаблона тестирования или метода жизненного цикла, если его выполнение превышает заданную продолжительность.
@ExtendWith	Используется для декларативной регистрации расширений.

4.2 Использование Mokito

Для подключения Mokito в проекте подключим зависимость в проект:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.0</version>
</dependency>
```

Для создания мока/стаба используется статический метод mock().

```
List mockedList = mock(List.class);
```

Для проверки взаимодействия используется метод vetify():

```
mockedList.add("one");
mockedList.clear();

verify(mockedList).add("one");
verify(mockedList).clear();
```

Для настройки стаба используется when(*вызов метода стаба*).then(*действия при вызове*). Под действиями может подразумеваться возврат значения, выброс исключения, получение аргумента.

```
when(mockedList.get(0)).thenReturn("first");

// выведет “first"
System.out.println(mockedList.get(0));
```

5 Альтернативные средства реализации

Существует несколько альтернативных фреймворков для написания юнит-тестов и создания мок-объектов, некоторые из них:

1. TestNG - это альтернативный фреймворк для написания юнит-тестов на языке Java.
2. Spock - это фреймворк для написания тестов на языке Groovy. Он предоставляет более выразительный способ написания тестов, чем JUnit, включая поддержку BDD (Behavior-Driven Development) и проверку условий с помощью блоков "expect".
3. PowerMock - это библиотека, которая расширяет возможности JUnit и Mockito, позволяя создавать мок-объекты для статических методов и конструкторов.

Сравнительная характеристика:

- JUnit является наиболее распространенным фреймворком для написания юнит-тестов на языке Java, он имеет широкую поддержку в инструментах разработки и CI/CD системах;
- Mockito является наиболее популярным фреймворком для создания мок-объектов в Java. Он предоставляет простой и интуитивно понятный API для создания мок-объектов и проверки вызовов методов;
- TestNG предоставляет более широкий набор функций, чем JUnit;
- Spock предоставляет более выразительный способ написания тестов, чем JUnit, включая поддержку BDD и проверку условий с помощью блоков "expect".
- PowerMock позволяет создавать мок-объекты для статических методов и конструкторов, что не поддерживается JUnit и Mockito.

Список использованных источников

1) jUnit5 About: [Электронный ресурс]. URL: https://junit.org/junit5/ .

2) jUnit5 JavaDocs: [Электронный ресурс]. URL:https://junit.org/junit5/docs/current/api/ .

3) jUnit5 исходный код [Электронный ресурс]. URL:https://github.com/junit-team/junit5 .

4) Mockito framework site [Электронный ресурс]. URL: https://site.mockito.org/.

5) Mockito исходный код [Электронный ресурс].URL: https://github.com/mockito/mockito.

6) Mockito JavaDocs [Электронный ресурс]. URL: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html.

7) Статья: “Mockito и как его готовить” [Электронный ресурс]. URL: https://habr.com/ru/post/444982/.

8) Mockito Tutorial Baeldung [Электронный ресурс]. URL: https://www.baeldung.com/mockito-series

9) A Guide to Junit5 [Электронный ресурс]. URL: https://www.baeldung.com/junit-5.

10) Parallel Test Execution for Junit5 [Электронный ресурс]. URL: https://www.baeldung.com/junit-5-parallel-tests.

11) Эккель Б. Философия Java, 4-е изд. – Спб.: Питер, 2021. – 1168с.

12) Бек К. Экстремальное программирование: разработка через тестирование. – Спб.: Питер, 2022. – 224с.

13) Хориков В. Принципы юнит-тестирования. – Спб.: Питер, 2022. – 320с.

14) Mellor A. Test-Driven Development with Java. – Birmingham: Packt, 2023. – 325с.

15) Freeman S., Pryce N. Growing Object-Oriented Software, Guided by Tests. – Boston: Addison-Wesley, 2010. – 385с.

16) Мартин Р. Чистый код: создание, анализ и рефакторинг. – Спб.: Питер, 2022. – 464с