



Введение в юнит-тестирование

Цели и смысл тестирования



Оглавление

Введение	3
Термины, используемые в лекции	3
Ошибки	4
Основные типы ошибок:	5
Почему важно найти ошибку	8
Принципы тестирования программного обеспечения	11
Цикл разработки	13
Написание простого теста	16
Ручное тестирование	16
Пограничные случаи	18
Утверждения (Assert)	23
Определение	23
Assert vs. exceptions	25
Недостатки assert	27
Библиотека AssertJ	27
Разница между фреймворком и библиотекой	27
Подключение AssertJ	28
Перепишем калькулятор	30
Возможности AssertJ	32
Домашнее задание	32
Используемая литература	33

Введение

В этом курсе изучаются как общие принципы написания эффективных тестов, так и современные инструменты для тестирования Java кода, курс познакомит с основами модульного тестирования (Unit-тестирования) на примере работы с фреймворком JUnit 5, также мы рассмотрим некоторые другие виды тестирования. Этот курс является вводным в дисциплину фреймворков автоматизированного тестирования и может быть полезен тем, кто хочет узнать больше о тестировании.

План курса



По окончании курса вы научитесь:

- Писать тесты для Java-кода.
- Применять современные подходы к тестированию.
- Понимать принципы модульного и функционального тестирования.
- Научитесь находить и исправлять ошибки в коде, используя принципы тестирования.

В этом материале будут рассмотрены следующие темы:

- Определение тестирования программного обеспечения.
- Что мы можем обнаружить в процессе тестирования?
- Принципы тестирования ПО.
- Цикл разработки.
- Assert.
- Пример теста на Java с использованием библиотеки AssertJ.

Термины, используемые в лекции

Программное обеспечение (ПО) — это совокупность программных и документальных средств для создания и эксплуатации систем обработки данных средствами вычислительной техники.

Бизнес-требования — определяют назначение ПО, описываются в документе о видении и границах проекта.

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

Программная ошибка (жарг. баг) — означает ошибку в программе или в системе, из-за которой программа выдаёт неожиданное поведение и, как следствие, результат.

IDE (англ. Integrated development environment) — комплекс программных средств, используемый программистами для разработки программного обеспечения, например, IntelliJ IDEA(Java).

Компилятор — программа, переводящая написанный на языке программирования текст в набор машинных кодов.

Жизненный цикл ПО (Software Development Lifecycle) — это последовательность этапов, на которых происходит разработка ПО.

Фрэймворк (иногда фреймвóрк; англицизм, неологизм от framework — остов, каркас, рама, структура) — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Библиотека (от англ. library) в программировании — сборник подпрограмм или объектов, используемых для разработки программного обеспечения (ПО).

Граничные значения — это значения, в которых один класс эквивалентности переходит в другой.

Класс эквивалентности (equivalence class) — одно или несколько значений ввода, к которым программное обеспечение применяет одинаковую логику.

Текущий интерфейс (англ. fluent interface — в значении «плавный» или «гладкий» «интерфейс») в разработке программного обеспечения — способ реализации объектно-ориентированного API, нацеленный на повышение читабельности

исходного кода программы. Название придумано Эриком Эвансом и Мартином Фаулером.

Ошибки

Ошибка — это ситуация когда то, что мы ожидаем, отличается от того, что получаем в результате.

В компьютерной технике тестирование — это процесс запуска/выполнения программы с целью найти ошибки. Ошибки могут быть как программные, так и аппаратные.

- **Программные** — это ошибки в программе (неправильная работа команды, сбой программы, неправильная передача данных).
- **Аппаратные** — ошибки в аппаратной части (неисправность процессора, видеокарты, жёсткого диска, оперативной памяти).

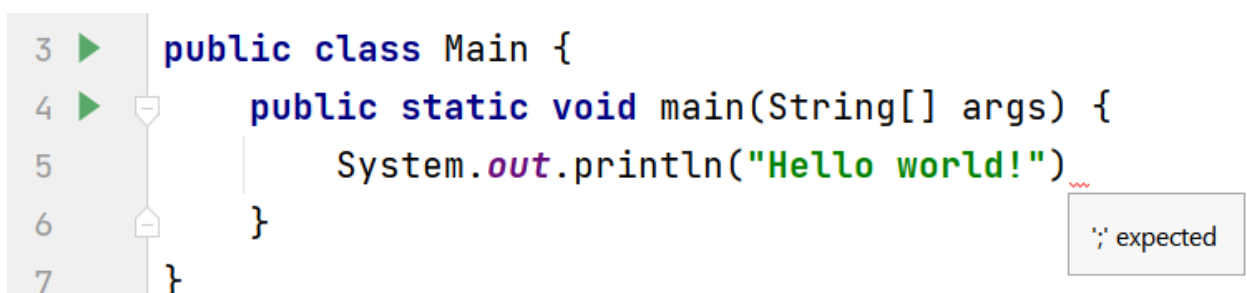
Далее мы будем разбирать только программные ошибки. Они возникают, когда разработчик допускает ошибку во время проектирования или конструирования приложения. Обнаруживать такие ошибки мы можем либо вручную, либо автоматизировано с помощью программ. До перехода к процессу тестирования важно разобраться в сути ошибок и в том, как они могут повлиять на конечный результат разработки.

Основные типы ошибок

1. Синтаксические

Это ошибки, которые в основном связаны с нарушением структуры языка — его синтаксиса. По сути, это ошибка в программе, не связанная с её функционалом (например, опечатка в тексте).

Пример: синтаксис Java требует завершать строку оператором точка с запятой в конце строки и если проигнорировать это требование, то IDE не даст запустить программу. Например, на скриншоте IDE подсказывает, что ожидается «;» и показывает где.



Ошибки такого типа легко обнаруживаются статическими анализаторами кода, встроенными в компилятор языка. Для их устранения достаточно просто исправить исходный код.

2. Логические (семантические)

Могут привести к ошибочным результатам и в некоторых случаях к сбоям в работе программы. Признак того, что существует семантическая ошибка – программа запускается, отработывает без ошибок выполнения, но не даёт желаемого результата.

Логическая ошибка возникает, когда программист неправильно пишет логику условного оператора или условия цикла. Вот простой пример с логической ошибкой программы для сравнения двух чисел:

```
public class Main {
    // Типы ошибок
    public static void main(String[] args) {
        compareNumbers(2, 2); // Вызывается метод сравнения двух
чисел
    }
    private static void compareNumbers(int a, int b) {
        if (a > b) {
            System.out.printf("%d more than %d", a,b);
        }
        if (a <= b) { // Допущена ошибка - знак <= вместо <
            System.out.printf("%d less than %d",a,b);
        }
    }
}
```

В примере выше описан метод для сравнения двух чисел a и b, результат сравнения выводится в консоль. Условие на 12 строке написано с ошибкой, в результате которой возможно получение неверного результата («2 less than 2»).

Как правило, компилятор не обнаруживает семантических ошибок (хотя в некоторых случаях умные компиляторы могут генерировать предупреждения).

Правила семантики сложно формализовать, поэтому тестирование и отладка логических ошибок зачастую становится сложной задачей, которую решает такая область программирования, как фреймворки автоматизированного тестирования.

3. Ошибки выполнения

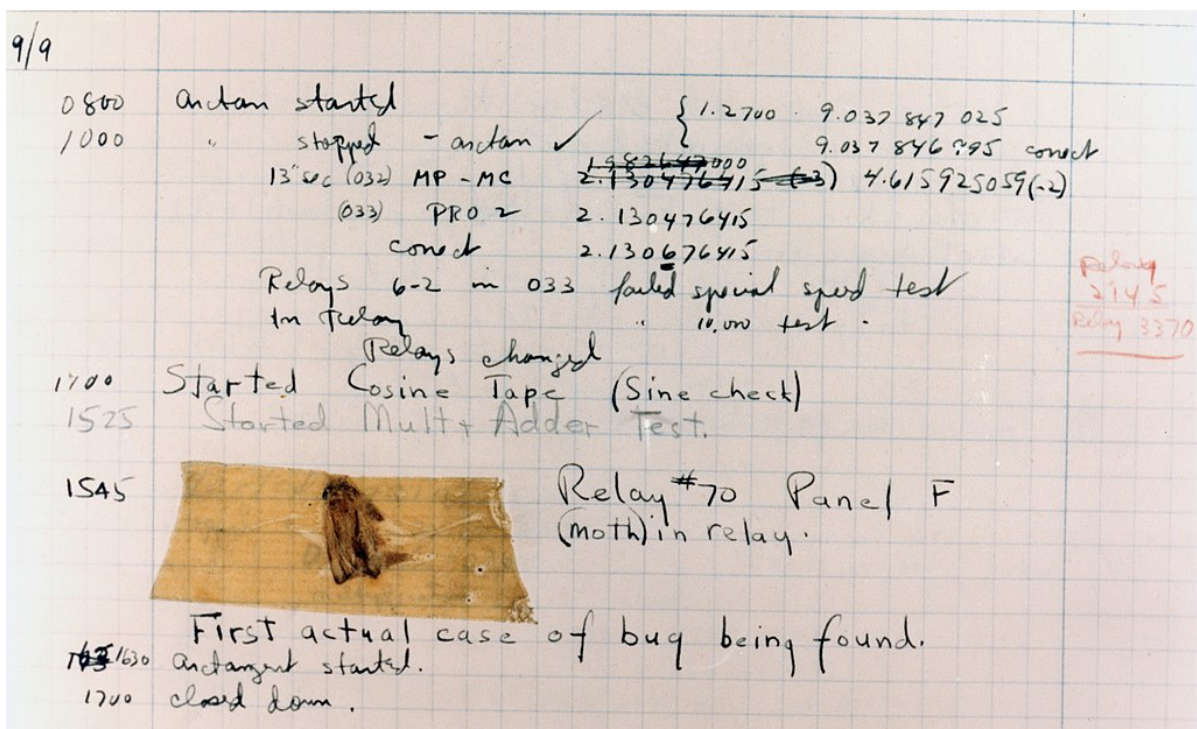
Ошибка будет обнаружена только во время выполнения программы. В отличие от семантических ошибок, эти прерывают программу и препятствуют её дальнейшему выполнению. Они обычно вызваны неожиданным результатом некоторых вычислений в исходном коде.

Пример:

```
public static void main(String[] args) {  
    int a = 10, b = 0;  
    System.out.printf("Result: %d", a/b);  
}
```

В этом примере целое число *a* пытаются разделить на другое целое число *b*, значение которого равно нулю, что приводит к *java.lang.ArithmeticException* (Название класса ошибки подсказывает природу ошибки).

В случае с ошибками выполнения можно использовать в качестве защиты от ошибок выполнение проверки входных данных, результата и, конечно, тестирование.



💡 По одной из версий, в отношении программной ошибки термин «баг» впервые был применён 9 сентября 1947 года Грейс Хоппер, которая работала в Гарвардском университете с вычислительной машиной Harvard Mark II. Проследив возникшую ошибку в работе программы до электромеханического реле машины, она нашла между замкнутыми контактами сгоревшего мотылька. Извлечённое насекомое было вклеено скотчем в технический дневник с

сопроводительной иронической надписью: «Первый реальный случай обнаружения жучка» (англ. First actual case of bug being found).

Существуют и другие классификации ошибок, например:

- По важности (ошибки разделяются в зависимости от того, насколько они лишают программу полезности).
- По времени появления (в зависимости от периодичности появления ошибки).

Бывают также классификации, создаваемые внутри продукта или команды, которые возникают в процессе организации разработки, такие ошибки связаны со спецификой разрабатываемого продукта. Например, ошибки в бекенд- и фронтенд-части.

Почему важно найти ошибку

В зависимости от характера ошибки, программы и среды исполнения, ошибка может проявляться сразу или долгое время оставаться незамеченной. Тестирование позволяет отладить процесс поиска ошибок в изменяющемся продукте и уменьшить цену ошибки, которая в зависимости от этапа разработки, может быть весьма высокой. Несколько примеров из жизни:



Авария космического аппарата Маринер-1 в 1962 году, который должен был отправиться к Венере, но почти сразу после старта антенна аппарата потеряла связь с наводящей системой на Земле в результате управление взял на себя бортовой компьютер, программа которого содержала ошибку (ошибка в ручном переводе математического символа в спецификации программы, а точнее — потерянная черта над символом). Ракета сильно отклонилась от курса, что создало серьёзную угрозу падения на землю. Для предотвращения возможной катастрофы NASA было принято решение запустить систему самоуничтожения ракеты.



В 2010 году ракета-носитель «Протон-М», которая должна была вывести на орбиту три спутника Глонасс-М, отклонилась от курса на 8 градусов. Причиной нештатного полёта явилось превышение массы разгонного блока ДМ-03 вследствие конструкторской ошибки в формуле расчёта дозы заправки жидкого кислорода в инструкции по эксплуатации системы контроля заправки (было залито чрезмерное количество топлива. В результате этого, спутники вышли на незамкнутую орбиту и упали в несудоходном районе Тихого океана. Причина в ошибке в формуле расчёта дозы заправки жидкого кислорода.



Случай в 2012 году с одной из крупнейших тогда фирм-биржевых трейдеров Knight Capital: где инженеры не проверили новое ПО для автоматизации торгов перед внедрением, в результате чего компания потеряла почти полмиллиарда долларов меньше чем за час. Из-за спешки не все серверы получили обновление, а устаревшая система не предназначалась для работы в реальных условиях.

В крупных проектах ошибка может быть очень дорогой и привести к серьёзным последствиям, ещё примеры, о которых можно почитать подробнее:

- Авария ракеты-носителя «Ариан-5» (4 июня 1996) — пример одной из самых дорогостоящих компьютерных ошибок в истории.
- Ошибки в программном обеспечении медицинского ускорителя Therac-25 привели к превышению доз облучения нескольких людей.
- Случай с Tesla Model S. 7 мая 2016 Джошуа Браун.
- Проблема 2038 года.
- Therac-25 — баг с аппаратом лучевой терапии.

💡 Для минимизации ошибок подобного рода люди придумали так называемые дополнительные стандарты кодирования, например, для моторной техники (всё от мопеда до шаттла) в США есть стандарт MISRA (Motor Industry Software Reliability Association) который предписывает писать код так или иначе, чтобы уменьшить вероятность ошибки. Таких стандартов довольно много и они направлены на разные области деятельности, но преследуют одну цель — повышение надёжности. По сути, MISRA — это методичка по написанию безопасного кода для микроконтроллеров, многие фреймворки тестирования берут за основу именно её, то есть сначала проверяют по ISO языка, потом по дополнительному стандарту, потом уже по бизнес-задачам кода.

Сложные программы сегодня всё активнее входят в нашу жизнь: они управляют транспортом, помогают в работе, облегчают быт. Любая программа создаётся человеком и, чем она сложнее, тем опаснее может быть её сбой.



Какой же выход? Модернизировать процесс разработки!

Комплексное использование современных методик тестирования существенно повышает вероятность того, что баг в коде будет обнаружен до публикации.

На приведённом ниже графике хорошо видна взаимосвязь между этапами жизненного цикла разработки (цикл рассмотрим позднее) и затратами на исправление ошибок:

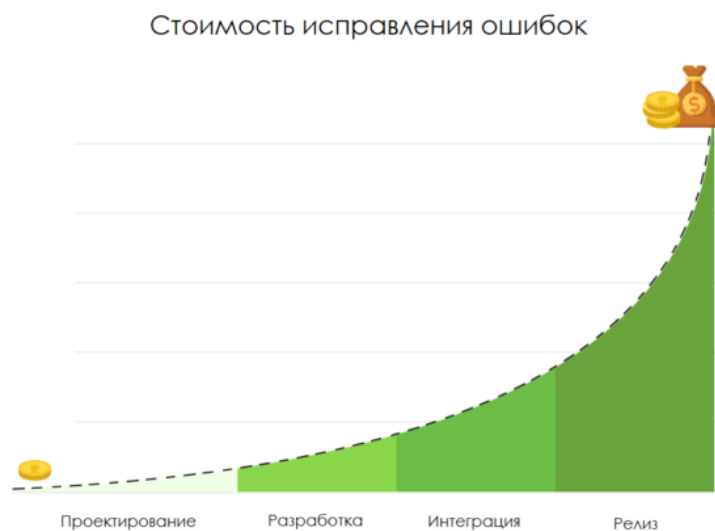


График также называют Кривой Бозма, это упрощённый вариант. Кривая показывает экспоненциальный рост стоимости исправления дефектов.

Гораздо дороже исправить ошибку, чем предотвратить её. Исправление одной ошибки может повлечь за собой другие, и количество проблем будет быстро увеличиваться, поэтому тестирование, как средство определения наличия ошибок, должно осуществляться с самого начала разработки программного обеспечения.

Для построения тестов нет чётких определений, как в физике, математике, которые при перефразировании становятся абсолютно неверными. Поэтому важно понимать процессы и подходы. В следующем разделе разберём основные принципы теории тестирования.

Принципы тестирования программного обеспечения

Принципы, представленные ниже, были разработаны в последние 40 лет и являются общим руководством для тестирования в целом. Они определяют, как тестировать и что тестировать. Понимание этих принципов даёт возможность избежать множества ошибок, а также оптимизировать затраты времени, сил, и нервов на тестирование.

1. Тестирование показывает наличие дефектов.

Тестирование только снижает вероятность наличия дефектов, которые находятся в программном обеспечении, но не гарантирует их отсутствия, другими словами, тестирование не исправит плохой код, а только покажет, где он плох.

2. Исчерпывающее тестирование невозможно.

Полное тестирование с использованием всех входных комбинаций данных физически невыполнимо. То есть для тестов надо подобрать правильные тестовые данные, например, какие-то граничные значения, явно невозможные значения и т. д. Чтобы проследить корректность поведения программы в этих случаях, при этом мы делаем важное допущение, что в остальных случаях программа тоже поведёт себя верно.

3. Раннее тестирование.

Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше. Это связано с тем, что намного дешевле исправить дефект на ранних стадиях разработки. Рекомендуется начинать поиск с момента, когда описаны требования системы.

4. Кластеризация дефектов (или скопление дефектов).

Принцип, который предполагает, что небольшое количество модулей содержат в себе большинство багов. Это яркий пример применения в тестировании принципа Парето: 80% проблем таятся в 20% модулей.

Можно просто запомнить что, если вы нашли несколько багов в каком-то модуле, то стоит изучить его тщательнее, скорее всего, есть ещё скрытые дефекты.

5. Парадокс пестицидов.

Если повторять те же тестовые сценарии снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты. Чтобы решить эту проблему, тесты должны регулярно пересматриваться и обновляться, должны добавляться новые тесты, ориентированные на другую функциональность или тестирующие уже существующие функции, но по другому.

Это понятие ввёл Борис Бейзер в своей книге «Software Testing Techniques». Он привёл следующую аналогию между повторным выполнением тестов и обработкой сельскохозяйственных полей химикатом (в частности, пестицидом), который уже был применён ранее. После первой обработки химикатом большая часть вредителей погибает, однако всё же некоторая часть из них выживает, потому что их организм оказался устойчив к яду. Те, кто выжил, с большой вероятностью переживут и последующие обработки пестицидом.

6. Тестирование зависит от контекста.

Тестирование проводится по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем новостной портал: то есть, для обеспечения безопасности приложения важнее сфокусироваться на тестировании авторизации пользователей, а для новостного портала важнее будет проверить, правильно ли отображается текст публикаций.

7. Заблуждение об отсутствии ошибок.

Отсутствие найденных дефектов при тестировании не всегда означает готовность продукта к публикации. Система должна быть удобна пользователю в использовании и удовлетворять его ожиданиям и потребностям.

Этот принцип говорит о том, что поиск и устранение багов не поможет, если построенная система изначально неправильно построена и не соответствует требованиям клиента. Поэтому, если формально все тесты пройдены, есть ещё очень много всяких «хотелок», которые нужно реализовывать, при этом важно, что реализация хотелок должна также быть в рамках тестов.

Цикл разработки

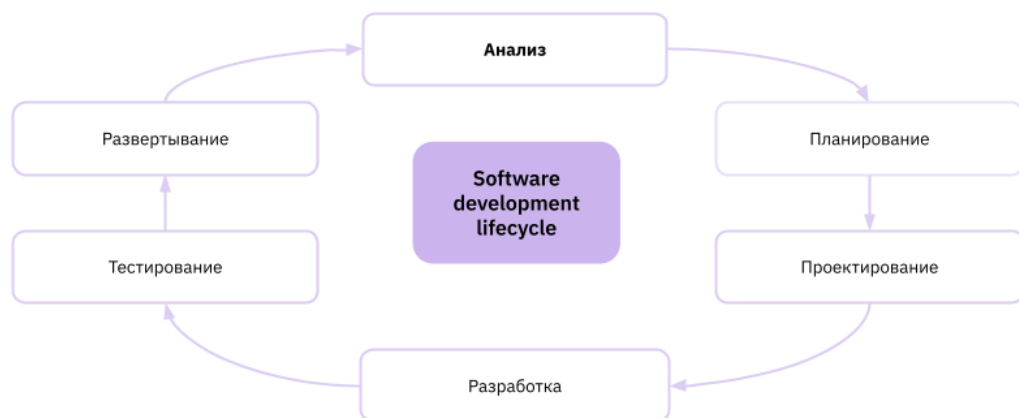
Жизненный цикл программного обеспечения — это ряд этапов, через которые проходит любой программный продукт от момента идеи и до момента выпуска ПО в широкое пользование. По сути, жизненный цикл разработки ПО — это и есть сам процесс разработки и развитие программного продукта.

При этом важно отметить, что существует множество моделей жизненного цикла, например:

- каскадная,
- инкрементная,
- спиральная,
- и др. (дополнительно о моделях можно почитать тут — [Модели и методологии разработки ПО | GeekBrains](#))

Любая модель регулирует лишь реализацию стадий жизненного цикла, но не сам жизненный цикл, который в принципе остаётся неизменным.

Цикл разработки. SDLC



У программного обеспечения, как у живого существа есть свой жизненный цикл. И, как и у каждого живого организма он имеет свои этапы развития. Жизненный цикл программного продукта – это последовательность этапов, на которых происходит разработка ПО от замысла до вывода продукта на рынок.

Каждый этап имеет свою цель и задачи, которые должны быть понятны всем участникам, например, тестировщикам нужно понимать входные и выходные данные своей деятельности, чтобы правильно и качественно выполнить свою работу.

Существует некая вариативность в прохождении этапов ЖЦ во время разработки и внедрения продукта на рынок. Для каждого продукта это происходит

по-своему, но чтобы процессом как-то управлять были сформулированы модели жизненного цикла ПО — упрощённое и обобщённое представление о том, как развивается продукт. В реальности жизнь продукта может не соответствовать модели.

Этапы ЖЦ:

1. Анализ.

На этом этапе формируются бизнес-требования к продукту, учитываются пожелания и потребности пользователей, оцениваются потенциальные риски и возможности. К работе привлекаются специалисты по продукту. В результате всех этих усилий должны появиться ответы на вопросы: *«Какие проблемы должно решать ПО?»*, *«Что именно (какой продукт) необходимо сделать?»* и *«Как именно это должно работать?»*.

2. Планирование.

На этапе планирования руководители проекта оценивают условия проекта. Это включает в себя расчёт затрат на рабочую силу и материалы, составление графика с указанием целевых показателей, а также создание команд проекта и структуры руководства.

3. Проектирование.

На этапе проектирования моделируется то, как будет работать программное приложение. На этом этапе обсуждается архитектура и другие тонкости системы. Также создание прототипа может быть частью этапа проектирования. Прототип демонстрирует основную идею о том, как приложение выглядит и работает.

4. Разработка.

Это фактическое написание программы. Небольшой проект может быть написан одним разработчиком, в то время как большой проект может быть разделён и работать несколькими командами.

5. Тестирование.

Очень важно протестировать приложение, прежде чем сделать его доступным для пользователей. Большая часть тестирования может быть автоматизирована, например, тестирование безопасности. Другое тестирование может быть выполнено только в определённой среде. Тестирование должно гарантировать, что каждая функция работает правильно. Различные части приложения также должны быть протестированы на бесперебойную совместную работу —

интеграционные тесты, чтобы уменьшить любые зависания или задержки в обработке.

После этого продукт можно внедрять и интегрировать со сторонним программным обеспечением. Процесс разработки на этом не заканчивается — он продолжается, пока не будут внесены доработки.


6. Развёртывание.

На этапе развёртывания приложение становится доступным для пользователей. Многие компании предпочитают автоматизировать этап развёртывания.

 Тестирование в разработке играет две важные роли:

1. Проверка соответствия бизнес-требованиям.
2. Обнаружение проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта. **(поддержание стабильного качества продукта)**.

Важно понимать, что нет чёткой границы между этими этапами, например, модульное тестирование, которому посвящён курс, начинается уже на этапе разработки, когда разработчик тестирует свой код. В рамках модульного тестирования проверяется одна или несколько конкретных функций, которые не зависят друг от друга. В частности, это могут быть процедуры и функции, написанные на языке программирования, или встроенные элементы системы.

 Unit-тесты могут быть тесно связаны со сложной технической частью ПО, тестировщик просто не сможет грамотно написать тексты для какого-то класса, поэтому чаще всего unit-тестами занимаются разработчики, которые писали тестируемый код. Следующие этапы тестирования уже могут выполняться тестировщиками, но в разных командах разные правила на этот счёт.

Написание простого теста

Ручное тестирование

Рассмотрим пример. Допустим, перед нами стоит задача написать простой консольный калькулятор на Java. Это будет программа, принимающая на вход числа и действия, которые нужно с ними сделать, и выводящая результат вычислений в консоль.

Calculator.calculation — принимает два операнда, и оператор, результат вычисления возвращается пользователю в виде int-переменной:

```
public static int calculation(int firstOperand, int
secondOperand, char operator) {
    int result = 0;
    switch (operator) {
        case '+':
            result = firstOperand + secondOperand;
            break;
        case '-':
            result = firstOperand - secondOperand;
            break;
        case '*':
            result = firstOperand * secondOperand;
            break;
        case '/':
            result = firstOperand / secondOperand;
            break;
    }
    return result;
}
```

Хорошей практикой является написание тестов сразу, а иногда и до написания основного кода. Поэтому приступим к тестированию. Что мы делаем, когда хотим протестировать метод? Правильно, запускаем и смотрим, как он работает, с разными вариантами входных данных, сверяя полученный результат с ожидаемым:

```
public class Calculator {
    public static void main(String[] args) {
        System.out.printf("Результат операции: %s \n",
calculation(2,2,'+')); //Результат операции: 4
        System.out.printf("Результат операции: %s \n",
calculation(2,1,'-')); // Результат операции: 1
        System.out.printf("Результат операции: %s \n",
calculation(2,3,'*')); // Результат операции: 6
        System.out.printf("Результат операции: %s \n",
calculation(8,2,'/')); // Результат операции: 4
    }

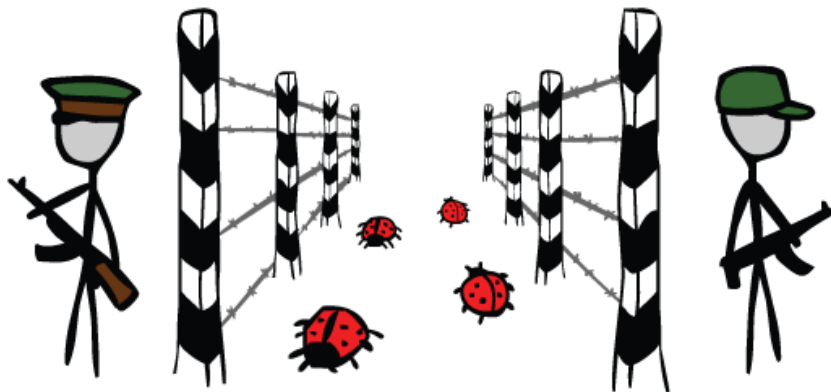
    public static int calculation(int firstOperand, int
secondOperand, char operator) {
        int result = 0;
        switch (operator) {
```

```
        case '+':
            result = firstOperand + secondOperand;
            break;
        case '-':
            result = firstOperand - secondOperand;
            break;
        case '*':
            result = firstOperand * secondOperand;
            break;
        case '/':
            result = firstOperand / secondOperand;
            break;
    }
    return result;
}
```

Таким образом, мы провели **ручное тестирование**, которое, на первый взгляд, подтвердило, что базовые операции выполняются без ошибок. То есть, мы просто глазами проверяли результат работы метода. Однако, с таким подходом, мы не можем точно знать, что проверили абсолютно все возможные исходы, и хотя обычно это не нужно, существуют условные правила, определяющие, как выбирать значения для проверки методов, такие проверки называют проверки пограничных случаев (Boundary Value).

Пограничные случаи

Баги водятся на границах



🔥 Ошибки в пограничных случаях — самая частая причина логических ошибок в программах. Программисты всегда забывают что-нибудь учесть.

Такие ошибки часто проявляются не сразу, и могут долгое время не приводить к видимым проблемам. Программа продолжает работать, но в какой-то момент обнаруживается, что в результатах есть ошибки.

Пограничные случаи в зависимости от специфики программы могут быть разными, в нашем случае с `calculation` это случаи, когда в метод передаются необычные варианты аргументов: ноль, отрицательные числа, числа с плавающей точкой, неправильный тип переменной (строка вместо числа), или её переполнение (например, переменные типа `int` способны хранить целые числа в диапазоне от -2 миллиарда до +2 миллиарда, как справится наш метод с большими числами?)

Проверим некоторые случаи:

Неправильное значение переменной: как метод поведёт себя, в случае если в качестве аргументов передать не тот тип, который ожидается? Проверим, выведем в консоль результат метода, допустим, мы ввели случайно знак подчёркивания вместо минуса:

```
System.out.printf("Результат операции: %s \n", calculation(8,
6, '_')); // Результат операции: 0
```

Получаем неожиданный результат — всё дело в том, что мы не учли, что в качестве оператора может быть введено некорректное значение. В таком случае будет правильным выводить ошибку о том, что такой операции нет. Можно подумать, что это не проблема: функция работает в нормальных условиях, и просто не нужно передавать ей «плохие» аргументы. В идеальном мире — да, но в реальном мире ваш код будет запускаться в разных ситуациях, с разными комбинациями условий и данных. Нельзя быть уверенным, что аргументы всегда будут корректными, поэтому нужно учитывать все случаи, в рамках здравого смысла.

Также часто ошибки могут возникать, когда функция неправильно обрабатывает нулевые аргументы. Например, в нашем методе интересно будет проверить результат деления на ноль:

```
System.out.printf("Результат операции: %s \n", calculation(8,
0, '/')); 
```


В результате программа завершается с ошибкой:


```
(Exception in thread "main" java.lang.ArithmeticException: / by
```

```
zero)
```

Прежде чем исправлять ошибки в коде калькулятора, сделаем тесты более автоматизированными: оформим тесты в виде отдельного класса (принято выносить тесты в отдельный класс, чтобы тесты никак не влияли на основную логику) и перепишем с использованием исключений:

```
public class CalculatorTest {
    public static void main(String[] args) {
        if (8 != Calculator.calculation(2, 6, '+')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (0 != Calculator.calculation(2, 2, '-')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (14 != Calculator.calculation(2, 7, '*')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (1 != Calculator.calculation(2, 2, '/')) {
            throw new AssertionError("Ошибка в методе");
        }
    }
}
```

 **AssertionError** Выбрасывается, чтобы указать, что утверждение завершилось неудачей.

 Класс `AssertionError` расширяет `Error`, который сам по себе расширяет `Throwable`. Это означает, что `AssertionError` является непроверенным исключением.

Делается это для удобства, тк использование метода `System.out.printf...` хоть и помогло выявить ошибки в методе, но нежелательно, как плохая практика (лишнее взаимодействие с пользователем и метод должен возвращать примитивные значения, а не печатать их).

И это всё ещё ручное тестирование, и в таком подходе есть свои минусы. Например, ошибка может быть семантической (не вызвать исключение), в таком случае, если проверок будет много, ошибку можно банально не заметить, поэтому использование автоматизированного тестирования более продуктивно.

Теперь исправим места в коде, которые выбрасывали исключения и напишем тесты для этих функций:

Calculator.calculation в итоге выглядит так:

```
public static int calculation(int firstOperand, int
secondOperand, char operator) {
    int result;
    switch (operator) {
        case '+':
            result = firstOperand + secondOperand;
            break;
        case '-':
            result = firstOperand - secondOperand;
            break;
        case '*':
            result = firstOperand * secondOperand;
            break;
        case '/':
            if (secondOperand != 0) {
                result = firstOperand / secondOperand;
                break;
            } else {
                throw new ArithmeticException("Division by zero
is not possible");
            }
        default:
            throw new IllegalStateException("Unexpected value
operator: " + operator);
    }
    return result;
}
```

Теперь добавим в CalculatorTest проверку, в этом случае есть некоторая особенность: основные тесты, которые нужно писать, это тесты на успешные сценарии работы (**позитивные** сценарии). Но в ситуации с делением на ноль мы проверяем, как поведёт себя в **негативном** сценарии. Код должен выбрасывать определённое исключение, мы можем проверить тип ошибки и сообщение, выбрасываемое программой. Посмотрите на код:

```
// Случаи с неправильными аргументами
// аргумент operator типа char, должен вызывать исключение,
если он получает не базовые символы (+-*/)
try {
    Calculator.calculation(8, 4, '_');
} catch (IllegalStateException e) {
```

```
    if (!e.getMessage().equals("Unexpected value operator: _"))
    {
        throw new AssertionError("Ошибка в методе");
    }
}
```

Для нас, как для пользователей, ничего не изменилось. Метод выбрасывает тот же, что и раньше, но если взглянуть на то, как проверяется код, тогда становится понятно, что теперь исключение это ожидаемое поведение и мы контролируем изменения в этой части кода. И всё-таки не слишком удобно, то, что удачное прохождение теста вызывает исключение. Мы исправим это, используя утверждения (assert), но сначала посмотрите на **итоговый код программы**:

Основной класс:

```
public class Calculator {
    public static int calculation(int firstOperand, int
secondOperand, char operator) {
        int result;
        switch (operator) {
            case '+':
                result = firstOperand + secondOperand;
                break;
            case '-':
                result = firstOperand - secondOperand;
                break;
            case '*':
                result = firstOperand * secondOperand;
                break;
            case '/':
                if (secondOperand != 0) {
                    result = firstOperand / secondOperand;
                    break;
                } else {
                    throw new ArithmeticException("Division by
zero is not possible");
                }
            default:
                throw new IllegalStateException("Unexpected
value operator: " + operator);
        }
        return result;
    }
}
```

Класс с тестами :

```
public class CalculatorTest {
    public static void main(String[] args) {
        // Проверка базового функционала с целыми числами:
        if (8 != Calculator.calculation(2, 6, '+')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (0 != Calculator.calculation(2, 2, '-')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (14 != Calculator.calculation(2, 7, '*')) {
            throw new AssertionError("Ошибка в методе");
        }
        if (2 != Calculator.calculation(100, 50, '/')) {
            throw new AssertionError("Ошибка в методе");
        }
        // Случаи с неправильными аргументами
        // аргумент operator типа char, должен вызывать
        // исключение, если он получает не базовые символы (+-*/)
        try {
            Calculator.calculation(8, 4, '_');
        } catch (IllegalStateException e) {
            if (!e.getMessage().equals("Unexpected value
operator: _")) {
                throw new AssertionError("Ошибка в методе");
            }
        }
    }
}
```

Утверждения (Assert)

Определение

Утверждения (Assert) — это встроенный в java механизм проверки правильности предположений. Он в основном используется для тестирования во время разработки. Утверждения реализуются с помощью оператора `assert` и `java.lang.Class AssertionError`. В коде это выражается следующим образом:

```
assert booleanExpression;
```

Если значение `booleanExpression` — `true`, то ничего не происходит. То есть считается, что предположение верно, если значение после ключевого слова `assert` будет равно `false`, тогда возникнет ошибка (будет выброшено `AssertionError`).

Это очень похоже на то, что мы уже делали раньше. Код просто становится более компактным и читаемым. Перепишем проверки базового функционала с целыми числами, используя `assert`:

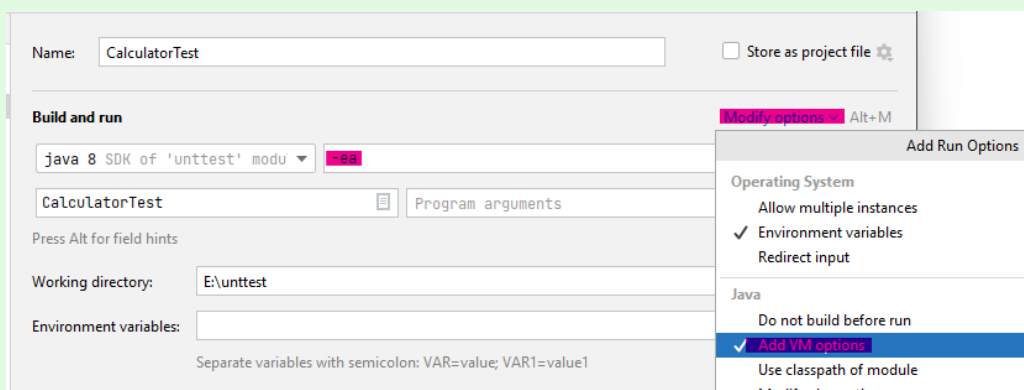
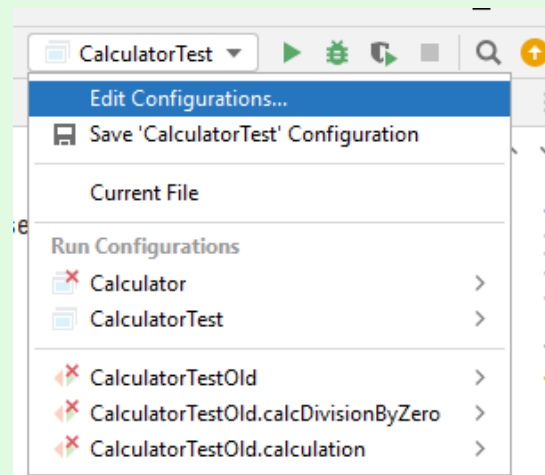
```
// Проверка базового функционала с целыми числами:
if (8 != Calculator.calculation(2, 6, '+')) {
    throw new AssertionError("Ошибка в методе");
}
if (0 != Calculator.calculation(2, 2, '-')) {
    throw new AssertionError("Ошибка в методе");
}
if (14 != Calculator.calculation(2, 7, '*')) {
    throw new AssertionError("Ошибка в методе");
}
if (2 != Calculator.calculation(100, 50, '/')) {
    throw new AssertionError("Ошибка в методе");
}
// Проверка базового функционала с целыми числами, с
использованием утверждений:
assert 8 == Calculator.calculation(2, 6, '+');
assert 0 == Calculator.calculation(2, 2, '-');
assert 14 == Calculator.calculation(2, 7, '*');
assert 2 == Calculator.calculation(100, 50, '/');
```

Возможно, вы заметили разницу между утверждением и логикой обнаружения ошибок. Утверждение проверяет `x == result`, а логика обнаружения ошибок проверяет `x != result`. Утверждение оптимистично: мы предполагаем, что аргумент в порядке. Напротив, логика обнаружения ошибок пессимистична: мы предполагаем, что аргумент не в порядке. Утверждения документируют правильную логику, в то время как исключения документируют неправильное поведение во время выполнения.

💡 Поскольку утверждения Java используют ключевое слово `assert`, для импорта не требуются библиотеки и пакеты.

💡 Обратите внимание, что до Java 1.4 было совершенно законно использовать слово «`assert`» для именования переменных, методов и т. д. Это потенциально создаёт конфликт имён при использовании более старого кода с более новыми версиями JVM.

💡 Поэтому для обеспечения обратной совместимости JVM по умолчанию отключает возможность использования утверждений. Они должны быть явно включены с помощью аргумента командной строки `-enableassertions` или его сокращения `-ea`. В IDEA можно это сделать так:



Assert vs. exceptions

В нашем примере, кроме количества строк кода, ничего не изменилось: `assert` вызывают тот же тип исключения и тесты проходят одинаково, тогда в чём же разница?

Так как `assert` завершает программу сразу же после обнаружения некорректных данных, он позволяет быстро локализовать и исправить баги в программе, которые привели к некорректным данным. Это его основное назначение.

Разработчики используют механизм исключений Java для реагирования на не критические (например, нехватка памяти) ошибки во время выполнения, которые могут быть вызваны факторами окружающей среды, такими как плохо написанный код или попытка разделить на 0. Обработчик исключений часто пишется для корректного восстановления после ошибки, чтобы программа могла продолжать работать.

Утверждения не заменяют исключений. В отличие от исключений, утверждения не поддерживают восстановление после ошибок (утверждения обычно немедленно останавливают выполнение программы — `AssertionError` не предназначен для перехвата); обычно `assert`'ы оставляют включёнными во время разработки и тестирования программ, но отключают в релиз-версиях программ.

Лучшие практики

Самое важное, что нужно помнить об утверждениях — это то, что они могут быть отключены, поэтому никогда не предполагайте, что они будут выполнены.

При использовании утверждений имейте в виду следующее:

💡 Всегда проверяйте наличие нулевых значений.

💡 Избегайте использования `Assert` для проверки входных данных в публичный метод и вместо этого используйте `unchecked` исключение, такое как `IllegalArgumentException` или `NullPointerException`.

💡 Не вызывайте методы в условиях утверждения, а вместо этого присваивайте результат метода локальной переменной и используйте эту переменную с `assert`.

💡 Утверждения отлично подходят для мест в коде, которые никогда не будут выполнены, например, для оператора `switch` по умолчанию или после цикла, который никогда не завершается.

Когда следует использовать исключения, а когда утверждения?

Ответ: везде, где проверка может быть полезна, нужно использовать `assert`'ы, но нужно учитывать, что они могут быть удалены на этапе компиляции либо во время исполнения программы, поэтому они не должны изменять выполнение программы. Если в результате удаления `assert`'а поведение программы может измениться, то это явный признак неправильного использования `assert`'а.

Исключения можно писать там, где после обнаружения бага, может потребоваться логика дальнейшей обработки ошибок (`try-catch`).

Недостатки `assert`

Несмотря на то что `assert`'ы, позволяют сделать код более читаемым и понятным, они имеют ряд недостатков, из-за чего их довольно редко используют:

- `Assert`'ы, как уже писалось, по умолчанию выключены.

- В проверяемых `assert`-ом функциях может быть только булево выражение, это ограничивает использование утверждений, приходится изменять код для проверок с `assert`-ом.

Библиотека AssertJ

Мы рассмотрим библиотеку AssertJ, в её основе те же механизмы что и в `assert`-ы, однако это не встроенный в язык инструмент, а отдельная подключаемая библиотека. Поэтому сначала разберёмся в том, что такое библиотека, а потом подключим к проекту и используем в коде.

Разница между фреймворком и библиотекой

Часто путают библиотеку с фреймворком, многие могут думать, что это синонимы. Рассмотрим сразу эти два понятия вместе, чтобы понять, в чём различия.

- **ФРЕЙМВОРК** — это набор взаимосвязанных классов и методов, которые позволяют создавать приложения, и которые можно использовать в других приложениях.
- **БИБЛИОТЕКА** — это просто набор классов. Например, классы, которые вы написали сами.

Фреймворк отличается от понятия библиотеки тем, что библиотека может быть использована в программном продукте просто как набор подпрограмм, не влияя на архитектуру программы и не накладывая на неё никаких ограничений. В то время как фреймворк диктует правила построения архитектуры приложения, задавая на начальном этапе разработки поведение по умолчанию — «каркас», который нужно будет расширять и изменять согласно указанным требованиям.

Пример фреймворка — JUnit (фреймворк для модульного тестирования программного обеспечения на языке Java, будет рассматриваться дальше в курсе), а пример библиотеки — AssertJ (Используется для написания более гибких и удобочитаемых утверждений, можно сказать, расширяет `assert`-ы, однако есть и отличия).

Подключение AssertJ

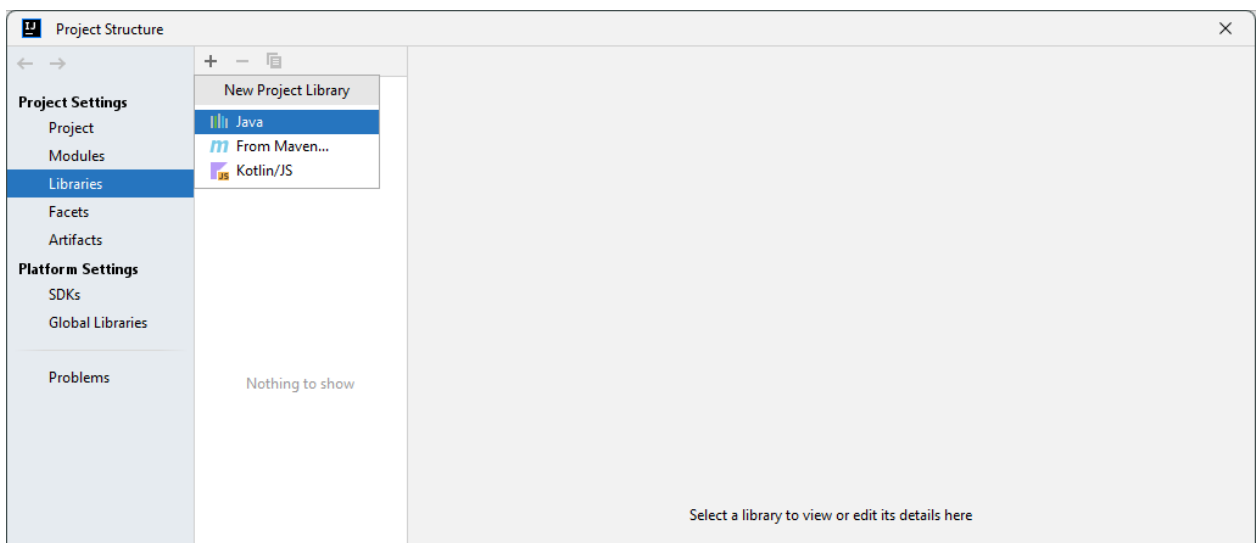
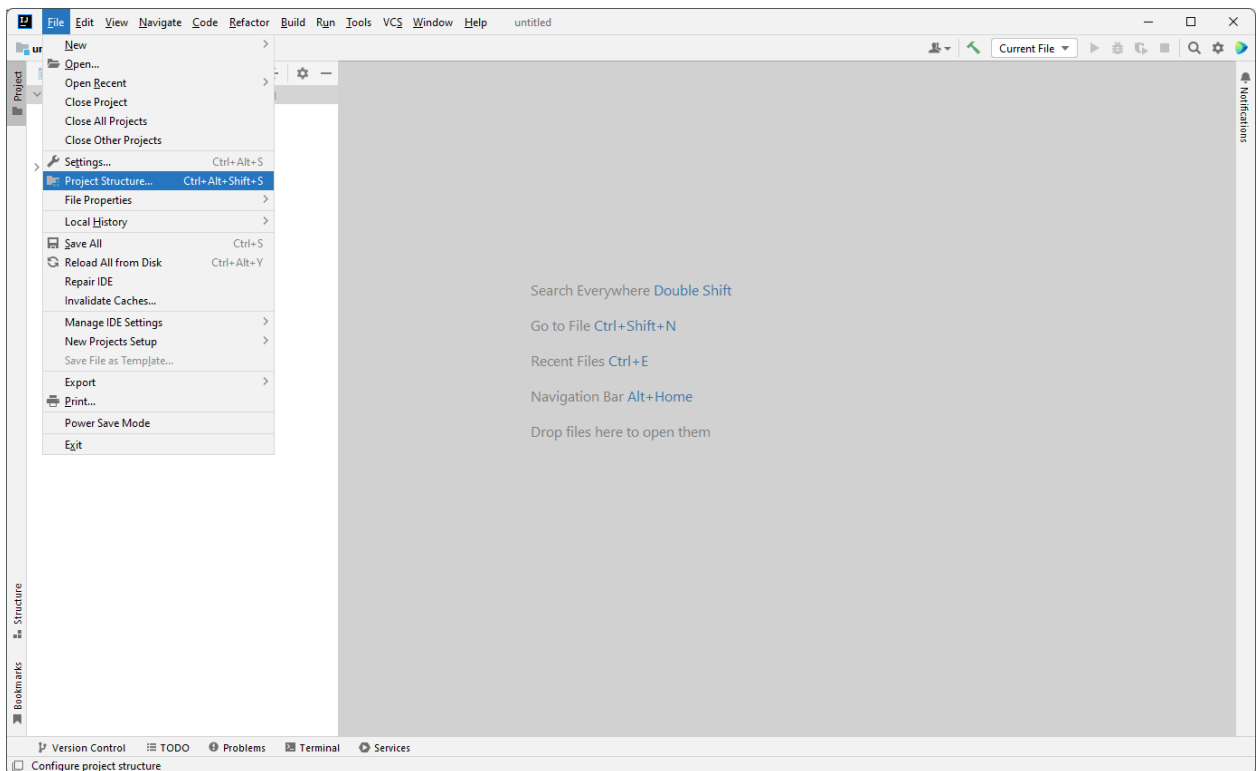
Чтобы начать использовать AssertJ в нашем коде, подключим зависимость:

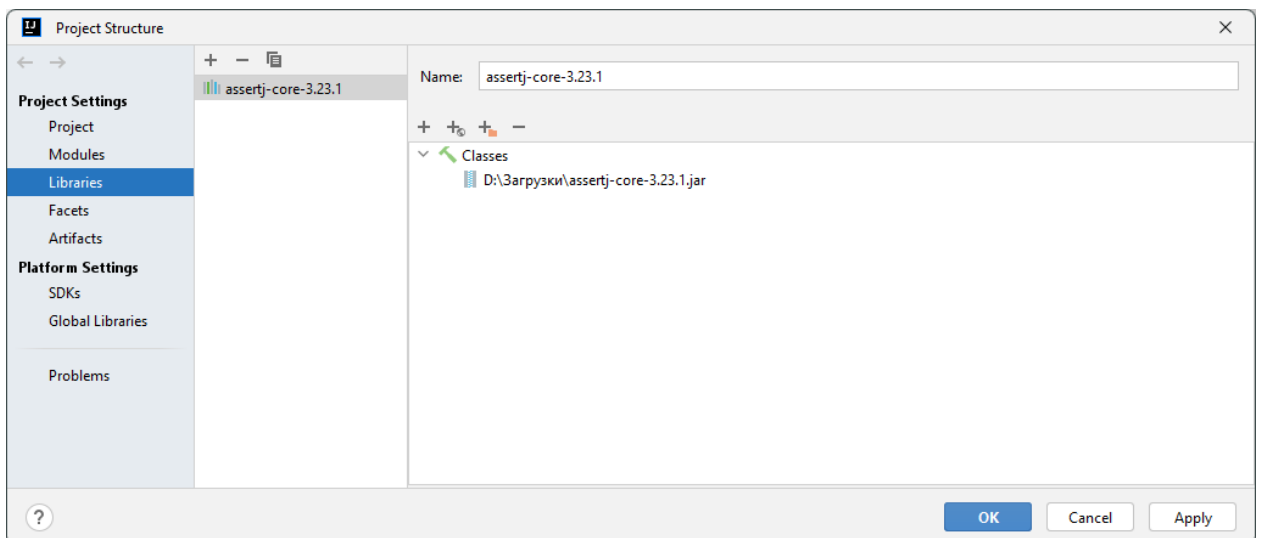
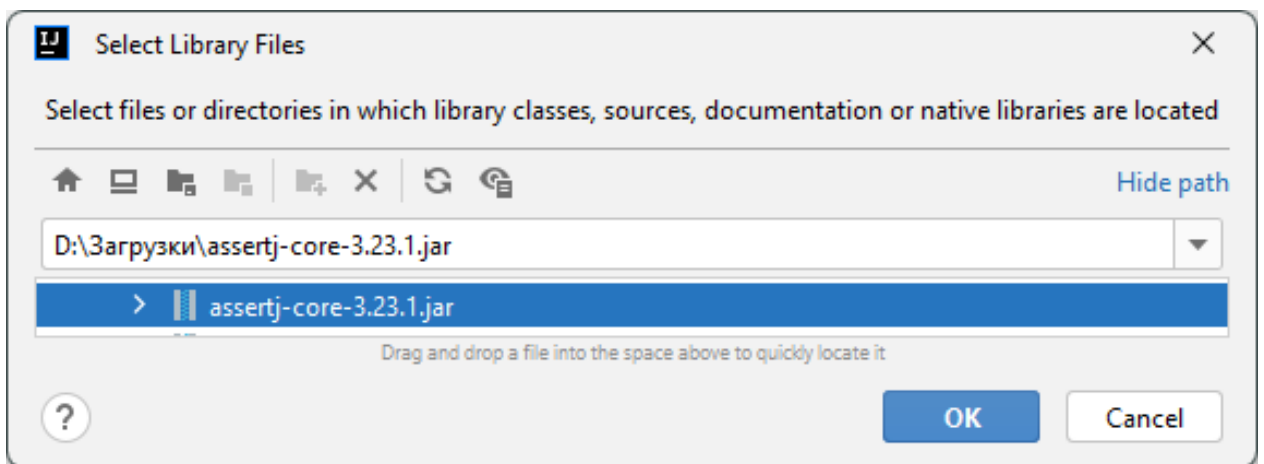
1. На сайте официальной документации библиотеки ([AssertJ — fluent assertions java library](https://assertj.github.io/doc/)) скачиваем jar-архив библиотеки. Для этого

переходим к пункту [Other build tools](#), затем переходим по ссылке в менеджер репозиторий и скачиваем [архив](#).

2. Далее нужно подключить библиотеку к проекту. Это можно сделать различными способами, рассмотрим на примере подключения с использованием ide IntelliJ IDEA. Можно подключить библиотеку разными способами, об этом можно почитать подробнее в статье — [Подключение библиотек в Java](#).

3. В IDEA переходим: File → Project Structure → Libraries → New Project Library → Java → выбираем скачанный в 1 пункте архив — подтверждаем.





4. Теперь переходим в класс Calculator в ide и импортируем AssertJ:

```
import static org.assertj.core.api.Assertions.*;
```

(Нужно вставить в самое начало кода класса строку выше, используя Директиву import.)

Перепишем калькулятор

Теперь, когда нам доступен функционал библиотеки. Рассмотрим пример тестов, переписанных с использованием этой библиотеки:

```
// Проверка базового функционала с целыми числами, с  
использованием утверждений AssertJ:  
assertThat(Calculator.calculation(2, 6, '+')).isEqualTo(8);  
assertThat(Calculator.calculation(2, 2, '-')).isEqualTo(0);  
assertThat(Calculator.calculation(2, 7, '*')).isEqualTo(14);  
assertThat(Calculator.calculation(100, 50, '/')).isEqualTo(2);
```

AssertJ применяется с использованием цепочек методов (Fluent interface) в результате чего, такие тесты очень легко читаются, как текст на английском языке:

Например:

```
assertThat(Calculator.calculation(2, 6, '+')).isEqualTo(8);
```

Можно просто перевести и смысл будет понятен:

«Утверждаю, что $2 + 6 = 8$ », если это не так, будет выброшено исключение с описанием вида:

```
Exception in thread "main" org.opentest4j.AssertionFailedError:
expected: Здесь будет ожидаемый результат
but was: Здесь фактический
```

С помощью AssertJ можно написать есть специальные проверки для ожидаемых исключений. Сравните старый код, который прерывал выполнение программы:

```
// Случай с неправильными аргументами
// аргумент operator типа char, должен вызывать исключение,
// если он получает не базовые символы (+-*/)
try {
    Calculator.calculation(8, 4, '_');
} catch (IllegalStateException e) {
    if (!e.getMessage().equals("Unexpected value operator: _"))
    {
        throw new AssertionError("Ошибка в методе");
    }
}
```

И новый код с использованием AssertJ:

```
assertThatThrownBy( () -> Calculator.calculation(8, 4, '_')
).assertInstanceOf(IllegalStateException.class);
```

💡 Пока вам может быть незнакома часть кода `() ->` — это синтаксис так называемых лямбда-выражений (или анонимных функций).

Можно сказать, что это специальный способ вызова функций, без объявления, т. е. метод как бы сразу описывается и используется.

💡 В `()` могут передаваться параметры, которые можно использовать в правой части, после `->` (пример — `... (p -> p.age > 30) ...`)

💡 Многие проверки в AssertJ используют лямбда-выражения, они позволяют раскрыть идеи Fluent interface про максимальную читаемость кода.

💡 Сейчас не обязательно понимать тонкости использования лямбда-выражений, достаточно знать что в `assertThatThrownBy` передаётся метод `calculation` с использованием лямбда-выражения

Выполнение программы в случае успешной проверки не прерывается, успешной проверкой в этом случае считается, если в результате вызова метода:

```
Calculator.calculation(8, 4, '_')
```

Будет выброшено исключение:

```
assertInstanceOf(IllegalStateException.class)
```

Выглядит этот код намного более читаемо и в случае, если будет выброшено исключение, библиотека подскажет, что именно пошло не так.

Возможности AssertJ

Кроме основных утверждений (`assertThat`) и утверждений об исключении (`assertThatThrownBy`) в AssertJ предоставлено ещё много полезных наборов утверждений, рассмотрим несколько на примерах из документации:

Помимо сравнений результатов вычислений, как в примере с калькулятором, можно проверять на соответствие строки, например:

```
assertThat(frodo.getName()).isEqualTo("Frodo");  
assertThat(frodo).isNotEqualTo(sauron);
```

Есть инструмент для работы с коллекциями, в примере ниже в передаваемом списке `fellowshipOfTheRing` размером 9 элементов утверждается наличие элементов `frodo`, `sam` и отсутствие `sauron`. Если будет передан список, который не удовлетворяет этим условиям, будет выброшено исключение.

```
assertThat(fellowshipOfTheRing).hasSize(9).contains(frodo,  
sam).doesNotContain(sauron);
```

Причём передаваемую коллекцию можно заранее отфильтровать:

```
assertThat(fellowshipOfTheRing).filteredOn(character ->  
character.getName().contains("o")).containsOnly(aragorn,  
frodo, legolas, boromir);
```

Есть возможность дополнить сообщение об ошибке с помощью `as()`, оно будет выводиться перед ошибкой:

```
assertThat(frodo.getAge()).as("check %s's age",  
frodo.getName()).isEqualTo(33);
```

Домашнее задание

1. Придумайте и опишите (можно в псевдокоде) функцию извлечения корня и необходимые проверки для него, используя граничные случаи.
2. Письменно ответьте: Как будет выглядеть проверка для случая деления на ноль? (с использованием AssertJ).
3. Письменно ответьте: Сравните одну и ту же проверку с использованием условий, ассертов, `assertJ` в каком случае стандартное сообщение об ошибке будет более информативным?
4. (Устно) Протестируйте системный калькулятор на вашем ПК на пограничные условия, как он реагирует? Как на нём обрабатывается деление на ноль?

Используемая литература

1. [Тестирование программного обеспечения](#)
2. [Программная ошибка](#)
3. [Что такое SDLC? Этапы, методология и процессы жизненного цикла программного обеспечения / Хабр](#)
4. [7 принципов тестирования. Часть 3 \(сайт IBS Training center\)](#)
5. [Assert. Что это? / Хабр](#)
6. [AssertJ – fluent assertions java library](#)
7. Е. Л. Кон, М. М. Кулагина «Надёжность и диагностика компонентов инфокоммуникационных и информационно-управляющих систем». — Изд-во Пермского нац. исслед. политех. университета, 2018. — 173 с.