AUTOMATIZATION

Сразу стоит отметить, что нет единственно верно устоявшейся и устраивающей всех терминологии, взглядов на тестирование и на автоматизацию.

То же самое касается и подходов: то, что работает в одной команде и компании, может быть бесполезным и даже вредным в другой команде и компании.

Поэтому вы должны:

1. Достаточно критически относиться к любым категоричным мнениям и суждениям из серии «нужно делать только так»;
2. Достаточно лояльно относиться к формулировкам и чужому опыту;
3. Постоянно проверять и адаптировать *ваши знания, навыки и подходы*

В подходах к классификации тестирования вводят классификацию *и по*

степени автоматизации (насколько автоматизирован процесс тестирования или его части):

— ручное — ничего не автоматизировано;

— с элементами автоматизации — присутствует автоматизация;

— автоматизированное — всё автоматизировано*.

Остаётся понять, что здесь подразумевается под автоматизацией.

Примечание: конечно, нужно понимать, что полностью всё

автоматизировать удаётся достаточно редко

Процесс тестирования (test process): Фундаментальный процесс

тестирования охватывает планирование тестирования, анализ и дизайн тестов, внедрение и выполнение тестов, оценку достижения критериев выхода и отчетность, а также работы по завершению тестирования.

Ключевое, что сопоставляя определения из классификации и ISTQB, мы делаем вывод: автоматизировать можно не только непосредственный прогон тестов, но также и подготовку тестового окружения, формирование отчётности, работы по завершению и много другое.

Т.е. вы не должны думать об автоматизации как о роботе, который вместо человека прогоняет тесты. Автоматизация имеет более широкую область применимости.

Цели автоматизации могут включать в себя следующие:

— Сокращение стоимости тестирования — замещение некоторых ручных операций делает процесс дешевле;

— Сокращение времени на тестирование — можем тестировать быстрее;

— Увеличение покрытия тестирования — можем тестировать больше функциональности;

— Проведение особых видов тестирования, которые человек не в состоянии провести — например, нагрузочное и т.д.;

— Увеличение частоты тестирования — особенно важно в гибких методологиях с несколькими релизами в день;

— Быстрая обратная связь

— Исключение человеческого фактора при ручном тестировании

(утомляемость, невнимательность и т.д.)

1. Дополнительная стоимость — на разработку тестов, поддержку, запуск и т.д.;
2. Повышение требований к уровню тестировщиков — тестировщики должны уметь программировать.
3. Сложность или невозможность автоматизации — что делать с системами, защищающимися от роботов (капчи и т.д.), или системами машинного обучения, выдающей вероятностный прогноз?
4. Ложные срабатывания — что если ошибка не в ПО, а в самих авто-тестах?
5. Сложность сравнения результатов (ожидаемый/фактический).

Ключевое: всегда просите (если позволяют обстоятельства) у заказчика пример расчёта, где формулировка не словесная, а в виде примера:

```
за  1 000 рублей начисляется   0 баллов
за  1 100 рублей начисляется   1 балл
за 11 000 рублей начисляется 100 баллов
за 20 000 рублей начисляется 100 баллов
```

Давайте разберёмся с уровнями:

1. Manual — ручные тесты, их должно быть меньше всего (т.к. плохо масштабируются);
2. GUI — тестирование через графический интерфейс;
3. API — тестирование через API (например, REST API);
4. Unit — изолированное тестирование отдельно взятого программного компонента.

ВЫБОР ИНСТРУМЕНТОВ

Есть несколько подходов:

— провести пилотный проект, детально изучив возможности конкретного инструмента (или их набора);

— ✓ выбрать стандарт де-факто (либо самые распространённые) и начать внедрять их;

— ✓ выбрать инструменты, «родные» для тех технологий, которые используются в разработке на проекте;

— ✓ выбрать самый простой инструмент и попробовать его, если не получится, всегда можно перейти на другой;

— ✓ другие варианты (включая комбинацию описанных выше).

ИНСТРУМЕНТЫ

1. Git и GitHub — хранение кода, в том числе авто-тестов;
2. JUnit — платформа для написания авто-тестов и их запуска;
3. Java 8 — язык написания авто-тестов
4. Gradle — система управления зависимостями.

JUnit — это платформа для написания авто-тестов и их запуска. Включает в себя достаточно много компонентов, ключевые для нас будут:

— junit-jupiter-engine — ядро JUnit Jupiter;

— junit-jupiter-api — API для написания авто-тестов (готовый набор классов, аннотаций);

— junit-jupiter-params — API для написания параметризованных авто-тестов.

Библиотеки распространяются в формате JAR-архивов (это обычный zip-архив с расширением

, в который запакована скомпилированная библиотека и ресуры).

ПЕРВОЕ ПРАВИЛО АВТОМАТИЗАЦИИ

Повторяющиеся рутинные операции должны быть автоматизированы.

На помощь приходят инструменты, используемые в большинстве Java-проектов: Maven и Gradle — инструменты автоматизации сборки и управления зависимостями, которые сами выкачают нужные вам библиотеки (и зависимости этих библиотек и т.д.).

ARTIFACT COORDINATES

У каждого создаваемоего приложения/библиотеки есть уникальный набор координат:

- `group` (или `GroupId`) — чаще всего reverse domain name;
- `name` (или `ArtifactId`) — имя проекта (или выходного файла);
- `version` — текущая версия.

По этому набору (координатам) его (приложение) можно будет найти в репозиториях наподобие Maven Central.

В рамках курса мы будем использовать `GroupId` `ru.netology`, а имя проекта — в соответствии с тем проектом, что мы делаем, например, `unit`.

Подключим к проекту нужные библиотеки. Для этого изменим файл `build.gradle`, в котором определены настройки нашего проекта

```
plugins {
  id 'java' // плагин, понимающий, как работать с Java
}
group 'ru.netology'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8 // работаем с Java 8
repositories {
  mavenCentral() // репозитории: покдлючен только Maven Central
}
dependencies { // нужные нам зависимости
  testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.1'
  testImplementation 'org.junit.jupiter:junit-jupiter-params:5.5.1'
  testRuntime 'org.junit.jupiter:junit-jupiter-engine:5.5.1'
  // все три зависимости можно заменить одной:
  // testImplementation 'org.junit.jupiter:junit-jupiter:5.5.1'
}
test {
  useJUnitPlatform() // включаем поддержку JUnit Jupiter
}
```

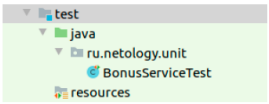
Q: Но что значит `testImplementation`? И какие ещё есть возможные значения?

A: Самые важные для нас:

- `implementation` — для компиляции приложения и работы приложения;
- `testImplementation` — для компиляции и запуска тестов приложения (но для работы самого приложения не нужна);
- `runtimeOnly` — только для работы приложения (но не для компиляциии);
- `testRuntimeOnly` — только для запуска тестов (но не для компиляциии).

С помощью конструктора выбрать, на какой метод мы будем писать тест и настроить доп.данные (мы оставим всё без изменений):

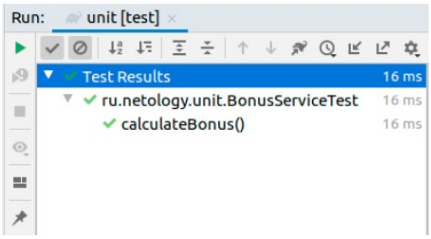
```
package ru.netology.unit;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class BonusServiceTest {

  @Test
  void calculateBonus() {
  }
}
```

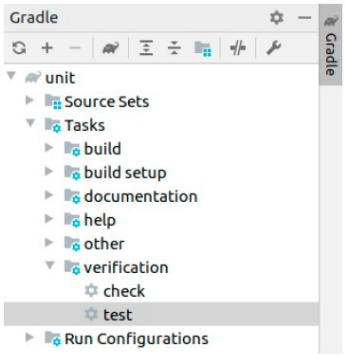


Запуск тестов на Gradle

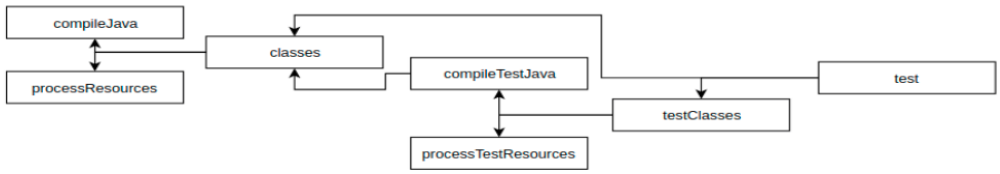
```
./gradlew test # *nix
gradlew test # Windows
```

ЗАПУСК ИЗ IDEA





GRADLE TASKS



`test` — это специальная задача (Task), запуск которой ведёт к выполнению целого цикла задач, а именно компиляции исходных кодов проекта, компиляции исходных кодов теста, обработке ресурсов и т.д.

Gradle сам следит, чтобы запускать нужные задачи в нужном порядке + запускать только те задачи, для которых что-то изменилось (если не меняли исходные коды проекта, а только автотесты, то не нужно перекомпилировать весь проект).

1. Наш тест содержится в обычном Java-классе (но расположен в каталоге для тестов);
2. Сам тест представляет из себя обычный метод, отмеченный аннотацией `@Test`;
3. В тесте импортируется сама аннотация Test (`import org.junit.jupiter.api.Test;`) и статические методы из класса Assertions (`import static org.junit.jupiter.api.Assertions.*;`).

Разобьём код нашего теста на три составляющих:

1. Подготовка нужных объектов и данных (в нашем случае — создание сервиса);

2. Выполнение целевых действий (в нашем случае — вызов метода сервиса);

3. Сравнение ожидаемого и фактического результата

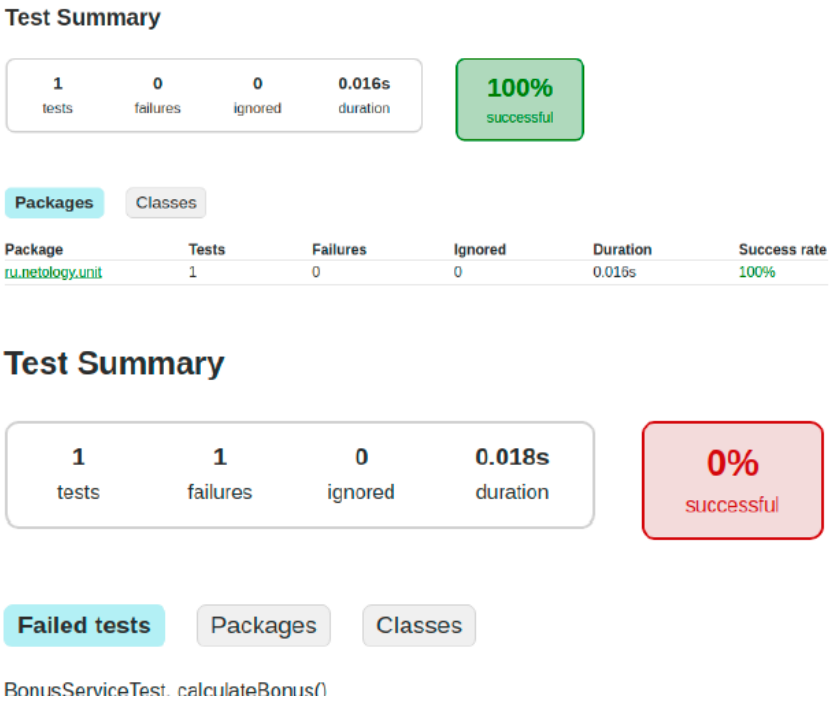
```
class BonusServiceTest {
  @Test
  void calculateBonus() {
    // подготовка
    BonusService service = new BonusService();
    int amount = 2000;

    // выполнение целевого действия
    int actual = service.calculateBonus(amount);
    int expected = 10;

    // сравнение ожидаемого и фактического
    assertEquals(expected, actual);
  }
}
```

JUnit предоставляет готовый набор методов для сравнения ожидаемого и фактического результата:
https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

Мы с вами использовали `assertEquals`, которая проверяет, что второй аргумент эквивалентен (равен в случае целых чисел) первому.

Запомните, что первым в JUnit всегда пишется ожидаемое значение.

Базовые версии отчётов в формате HTML можно найти в каталоге `build/reports/tests`:



Как всегда, подходов к именованию есть достаточно много, но в рамках курса мы (чтобы не усложнять) будем придерживаться следующего:

- Пример: сервис должен возвращать 0 баллов, если сумма покупки меньше 1000 рублей: `shouldReturnZeroIfAmountLowerThan1000`;
- Пример: сервис должен возвращать 10 баллов, если сумма покупки равна 2000 рублей: `shouldReturn10IfAmountIs2000`.

Ключевое: вам нужно использовать именно те соглашения по именованию, которые приняты в вашей команде.

Вы прекрасно знаете, что в соответствии с классами эквивалентности, у

нас будут тест-кейсы, которые тестируют недопустимые значения, например:

— текст вместо числа;

— очень большое число (за границами int);

— отрицательные числа.

Нужно понимать, что в большинстве случаев, обработка этих ситуаций не входит в перечень задач сервиса BonusService и валидацию входных значений для него должен производить какой-то другой компонент системы*.

```java
@Test
void shouldReturn10IfAmountIs2000() {
  BonusService service = new BonusService();
  int amount = 2000;

  int actual = service.calculateBonus(amount);
  int expected = 10;

  assertEquals(expected, actual);
}

@Test
void shouldReturnZeroIfAmountLowerThan1000() {
  BonusService service = new BonusService();
  int amount = 900;

  int actual = service.calculateBonus(amount);
  int expected = 0;

  assertEquals(expected, actual);
}
```