

# ОСНОВЫ АВТОМАТИЗАЦИИ



АРТЕМ РОМАНОВ



# АРТЕМ РОМАНОВ

Инженер по обеспечению качества



[romanov-artyom](#)



[@Artem\\_Telegram](#)



[Романов Артём](#)



# ПЛАН ЗАНЯТИЯ

1. Цели автоматизации
2. Автоматизация
3. Инструменты
4. Итоги



# ЦЕЛИ АВТОМАТИЗАЦИИ



# АВТОМАТИЗАЦИЯ

Поговорим об автоматизации.

*Вопрос к аудитории:*

- 1. Что вы понимаете под термином автоматизация тестирования?*
- 2. Что вы знаете о классификации тестирования по степени автоматизации?*

# ТЕРМИНЫ, ОПРЕДЕЛЕНИЯ, ПОДХОДЫ И Т.Д.

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.

# КЛАССИФИКАЦИЯ

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 удастся достаточно редко.

# ПРОЦЕСС ТЕСТИРОВАНИЯ

Напоминаем, что существует организация ISTQB, которая в том числе пытается унифицировать терминологию, так вот:

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.

(выдержка из перевода ISTQB на русский язык)



---

# АВТОМАТИЗАЦИЯ

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.

Таким образом, во всём курсе под автоматизацией мы будем понимать автоматизацию любых частей процесса тестирования (и не будем каждый раз уточнять, что это относится не только к прогону тестов).

# ЦЕЛИ АВТОМАТИЗАЦИИ

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.)

# ВОПРОСЫ

**Q:** Зачем мы обсуждаем эту теорию, вместо того, чтобы начать автоматизировать?

**A:** Вы должны понимать, зачем вы автоматизируете и какой профит это принесёт вашему проекту. Если вы этого не понимаете и не можете измерить (у вас нет ответа на этот вопрос), то вместо профита вы можете попросту навредить своему проекту.

**Q:** Т.е. от автоматизации может быть вред?

**A:** Да, конечно.

# ОТРИЦАТЕЛЬНЫЕ СТОРОНЫ АВТОМАТИЗАЦИИ

У автоматизации достаточно много отрицательных сторон и ограничений.

Например:

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



## ПРОМЕЖУТОЧНЫЕ ВЫВОДЫ

Итак, мы обсудили понятие автоматизации тестирования, выяснили наиболее распространённые цели и отрицательные стороны.

При внедрении автоматизации и последующем контроле постоянно держите их в голове.

Мы постоянно будем возвращаться к ним, чтобы вы о них не забывали.



# АВТОМАТИЗАЦИЯ

# ЗАДАЧА

Реальная задача. Одна из продуктовых сетей внедрила бонусную систему для своих покупателей — при каждой покупке пользователю, предъявившему бонусную карту, начисляются бонусные баллы по следующей логике:

1. Если покупка меньше 1000 рублей, бонусы не начисляются;
2. При покупке свыше 1000 рублей начисляется один бонусный балл за каждые полные 100 рублей;
3. За одну покупку не может быть начислено более 100 баллов.

*Вопрос к аудитории: какие недостатки в формулировке задачи можно выделить, которые мешают вам протестировать корректную логику работы системы?*

*Вопрос к аудитории: что вы можете сделать, чтобы устранить их?*

# ВОПРОСЫ

Это не выдуманный пример, достаточно часто заказчики формулируют задачи именно подобным образом.

«Додумывать» за заказчика — плохая идея, так или иначе придётся уточнять следующие моменты:

1. Есть ли бонусы ровно с 1000 рублей, или это должно быть 1000 руб. и 1 копейка, чтобы начислить бонус (типичная проблема граничного значения).
2. С 2000 рублей бонус должен быть 20 или 10 (т.е. считается ли бонус с первой тысячи, или только сверх неё).
3. А если получилось ровно 100 баллов? Выдаются они или уже нет?



## ПРИМЕР

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



# ТЕСТИРУЕМ

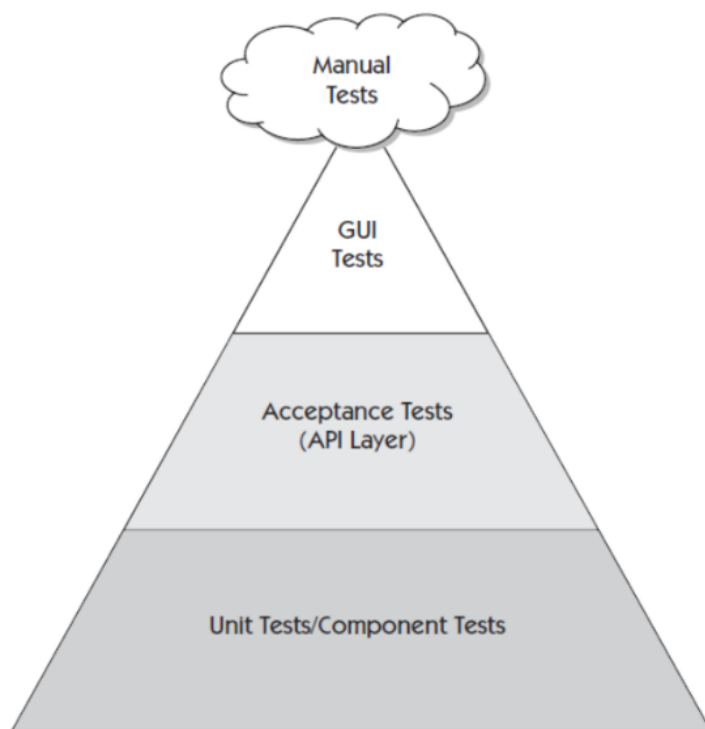
Итак, чтобы протестировать, нам нужно всего лишь:

1. Найти магазин (или тестовую систему), в которой уже внедрены бонусы;
2. Получить карту (или её аналог);
3. Совершить (или эмулировать) покупки на различные суммы.

И, как обычно бывает, разработчики нам говорят, что сервис они написали, но в основную систему не интегрировали, внедренцы говорят, что внедрять это будут только через месяц и т.д.

# ПИРАМИДА ТЕСТИРОВАНИЯ

В автоматизации широко используется пирамиды тестирования, которая описывает соотношения количества автоматизированных тестов для каждого уровня:



Изображение из книги «Agile Testing»

# УРОВНИ ТЕСТИРОВАНИЯ

Давайте разберёмся с уровнями:

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

*Вопрос к аудитории: как вы думаете, почему размер уровней (количество тестов на них) построен именно в такой пропорции?*

# КОД СЕРВИСА

```
1  package ru.netology.unit;
2
3  public class BonusService {
4      public int calculateBonus(int amount) {
5          if (amount < 1000) {
6              return 0;
7          }
8
9          int bonus = (amount - 1000) / 100;
10
11         if (bonus > 100) {
12             return 100;
13         }
14
15         return bonus;
16     }
17 }
```

# UNIT-ТЕСТИРОВАНИЕ

Имея код сервиса, мы можем уже написать на него авто-тесты, проверяя как работает этот метод.

Обратите внимание, что наш сервис прекрасно подходит для тестирования — у него нет внутреннего состояния, и он ни от кого не зависит (ни от каких других классов).

Всё, что нужно сделать для работы с ним, это создать объект класса и вызвать нужный метод:

```
1 package ru.netology.unit;
2
3 public class Main {
4     public static void main(String[] args) {
5         BonusService service = new BonusService();
6         int result = service.calculateBonus(1000);
7         System.out.println(result);
8     }
9 }
```

# UNIT-ТЕСТИРОВАНИЕ

Мы даже можем сюда накидать проверку остальных сценариев, прописать сравнение фактического и ожидаемого результатов:

```
1 package ru.netology.unit;
2
3 public class Main {
4     public static void main(String[] args) {
5         BonusService service = new BonusService();
6         {
7             System.out.println("It should return 0 for 1000");
8             int result = service.calculateBonus(1000);
9             System.out.println(0 == result);
10        }
11        {
12            System.out.println("It should return 10 for 2000");
13            int result = service.calculateBonus(2000);
14            System.out.println(10 == result);
15        }
16    }
17 }
```

А что, если тестов станет 200, 300, 500? Нам все сравнивать глазами?



# ИНСТРУМЕНТЫ





# ЗАДАЧА

Итак, первая задача — определить, что нам необходимо, чтобы автоматизированно подготавливать тестовое окружение, запускать авто-тесты и формировать отчётность.

Всё это мы будем рассматривать сегодня — в контексте Unit-тестирования, а в целом — в контексте тестирования UI-интерфейсов веб-приложений (в пирамиде - GUI).

# ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ

Возникает вопрос: какие инструменты тестирования использовать? Т.к. выбор инструментов и их сочетаемость зачастую определяют весь остальной набор технологий.

Хотя, можно идти и по-другому пути, отталкиваясь от навыков и знаний команды: если команда программирует на Java, наверное, не особо логично выбирать инструменты тестирования, где нужно писать на C#/Ruby/etc.

*Вопрос к аудитории: как бы вы предложили организовать процесс инструментов тестирования?*

# 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).

Примечание\*: галочкой отмечено то, что будем использовать мы.



# ИНСТРУМЕНТЫ

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

Для хранения кода будем использовать Git и GitHub (с заранее подготовленным [.gitignore](#))

# JUNIT JUPITER

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 и т.д.);
- `junit-jupiter-params` — API для написания параметризованных авто-тестов.

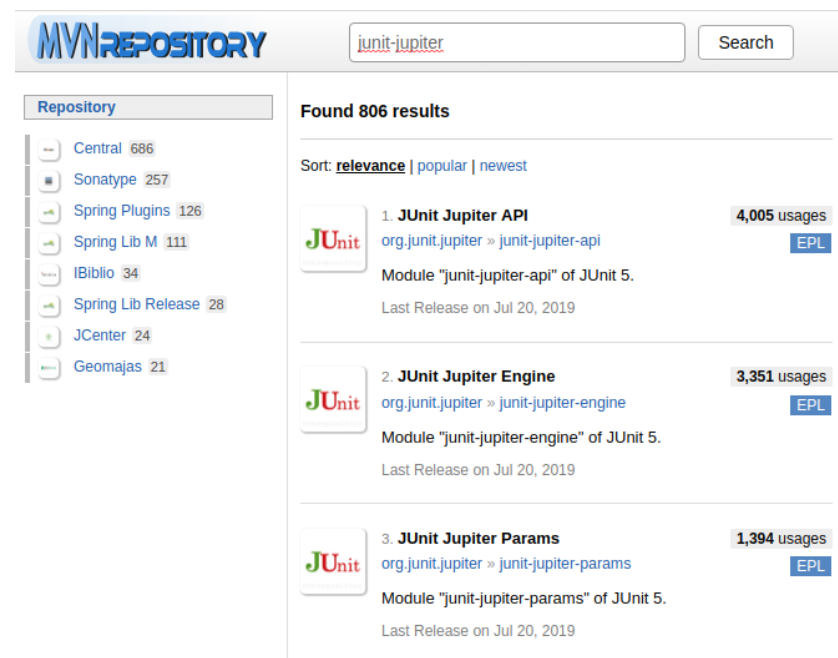
Где же их взять?

# MAVEN CENTRAL

В мире Java есть репозитории для хранения готовых библиотек.

Самый популярный из них — [Maven Central](https://mvnrepository.com/). Именно там хранятся готовые версии библиотек, и JUnit в их числе.

Для поиска в Maven Central и других репозиториях можно воспользоваться сайтом <https://mvnrepository.com/>.



The screenshot shows the Maven Repository website. The search bar at the top contains 'junit-jupiter' and the search button is labeled 'Search'. On the left, a sidebar lists various repositories with their respective artifact counts: Central (686), Sonatype (257), Spring Plugins (126), Spring Lib M (111), IBiblio (34), Spring Lib Release (28), JCenter (24), and Geomajas (21). The main content area displays 'Found 806 results' and offers sorting options: relevance (selected), popular, and newest. Three results are listed:

- JUnit Jupiter API** (4,005 usages)  
org.junit.jupiter » junit-jupiter-api  
Module "junit-jupiter-api" of JUnit 5.  
Last Release on Jul 20, 2019
- JUnit Jupiter Engine** (3,351 usages)  
org.junit.jupiter » junit-jupiter-engine  
Module "junit-jupiter-engine" of JUnit 5.  
Last Release on Jul 20, 2019
- JUnit Jupiter Params** (1,394 usages)  
org.junit.jupiter » junit-jupiter-params  
Module "junit-jupiter-params" of JUnit 5.  
Last Release on Jul 20, 2019

# JAR

Библиотеки распространяются в формате JAR-архивов (это обычный zip-архив с расширением `.jar`, в который запакована скомпилированная библиотека и ресурсы).

Соответственно, остаётся только аккуратно зайти в каждую библиотеку, скачать нужный JAR, сохранить в проект и т.д. (это должен будет делать каждый участник команды):

[Home](#) » [org.junit.jupiter](#) » [junit-jupiter-api](#) » **5.5.1**



## JUnit Jupiter API » 5.5.1

Module "junit-jupiter-api" of JUnit 5.

License	<a href="#">EPL 2.0</a>
Categories	<a href="#">Testing Frameworks</a>
HomePage	<a href="https://junit.org/junit5/">https://junit.org/junit5/</a>
Date	(Jul 20, 2019)
Files	<a href="#">jar (138 KB)</a> <a href="#">View All</a>
Repositories	<a href="#">Central</a>
Used By	<b>4,005 artifacts</b>



# ПЕРВОЕ ПРАВИЛО АВТОМАТИЗАЦИИ

**Повторяющиеся рутинные операции должны быть автоматизированы.**

Вы не должны тратить своё время на то, чтобы руками скачивать библиотеки, класть их куда-то и т.д.

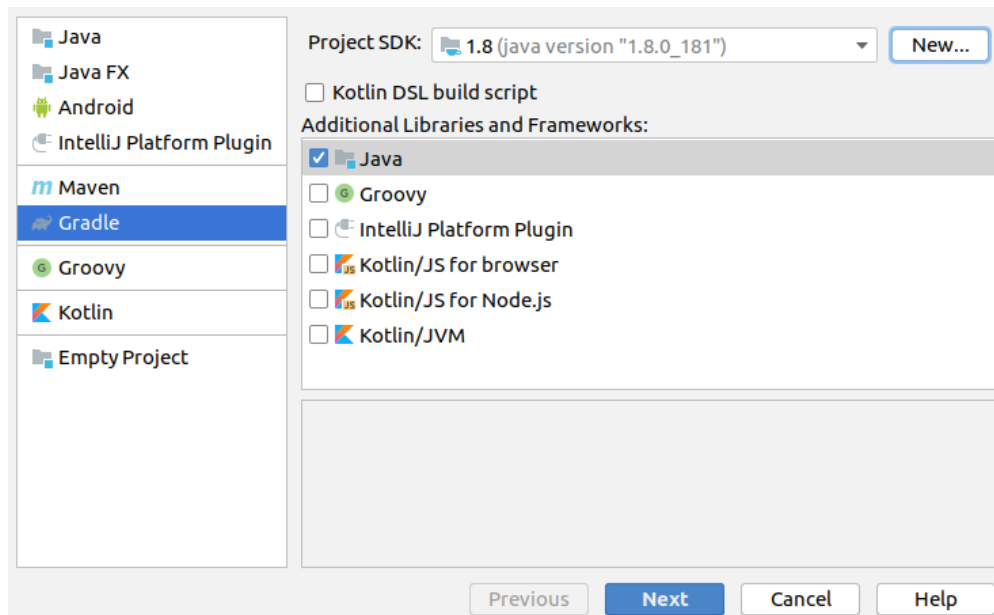
Тем более, нужная вам библиотека может зависеть от других библиотек, а те, в свою очередь от других, и ... вы замучаетесь всё выкачивать.



# GRADLE

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.).

Создадим проект на базе Gradle (в IDEA при создании выбрать Gradle):



# ARTIFACT COORDINATES

У каждого создаваемого 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`.

# ARTIFACT COORDINATES

Maven

Gradle

SBT

Ivy

Grape

Leiningen

Buildr

```
// https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.5.1'
```

☒ Include comment with link to declaration

Можно использовать сокращённую запись вида:

```
testCompile 'org.junit.jupiter:junit-jupiter-api:5.5.1'
```

# ЗАВИСИМОСТИ

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

```
1  plugins {
2      id 'java' // плагин, понимающий, как работать с Java
3  }
4  group 'ru.netology'
5  version '1.0-SNAPSHOT'
6  sourceCompatibility = 1.8 // работаем с Java 8
7  repositories {
8      mavenCentral() // репозитории: подключен только Maven Central
9  }
10 dependencies { // нужные нам зависимости
11     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.1'
12     testImplementation 'org.junit.jupiter:junit-jupiter-params:5.5.1'
13     testRuntime 'org.junit.jupiter:junit-jupiter-engine:5.5.1'
14     // все три зависимости можно заменить одной:
15     // testImplementation 'org.junit.jupiter:junit-jupiter:5.5.1'
16 }
17 test {
18     useJUnitPlatform() // включаем поддержку JUnit Jupiter
19 }
```

# implementation, testImplementation


**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 компиляции);
- `testRuntimeOnly` – только для запуска тестов (но не для компиляции).







# IDEA

После внесения изменений в файл `build.gradle` IDEA предложит нам применить их (чтобы включить автоподсказки на основании изменений и скачать зависимости):

 **Gradle projects need to be imported**

[Import Changes](#) [Enable Auto-Import](#)

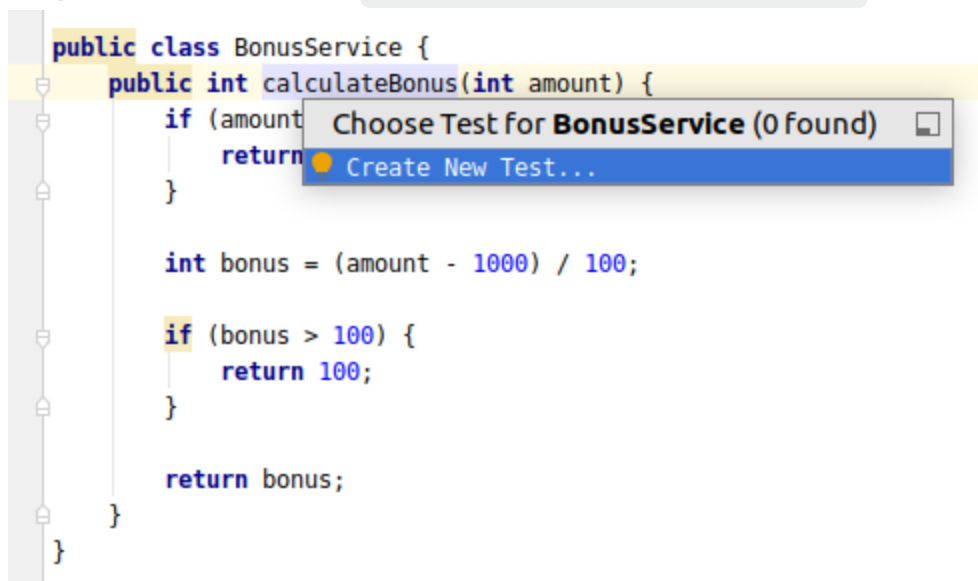
## External Libraries

- ▶  < 1.8 > /opt/bin/jdk1.8.0\_181
- ▶  Gradle: org.apiguardian:apiguardian-api:1.1.0
- ▶  Gradle: org.junit.jupiter:junit-jupiter-api:5.5.1
- ▶  Gradle: org.junit.jupiter:junit-jupiter-params:5.5.1
- ▶  Gradle: org.junit.platform:junit-platform-commons:1.5.1
- ▶  Gradle: org.opentest4j:opentest4j:1.2.0

# НАПИСАНИЕ АВТОТЕСТА

Попробуем наконец написать наш первый автотест с использованием JUnit.

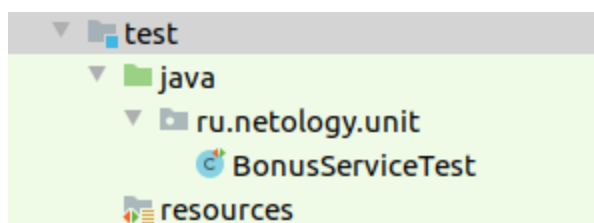
Мы можем это сделать вручную, либо в IDEA над нужным нам методом сервиса нажать **Ctrl + Shift + T**:



# НАПИСАНИЕ АВТОТЕСТА

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

```
1 package ru.netology.unit;  
2  
3 import org.junit.jupiter.api.Test;  
4  
5 import static org.junit.jupiter.api.Assertions.*;  
6  
7 class BonusServiceTest {  
8  
9     @Test  
10    void calculateBonus() {  
11    }  
12 }
```





# ЗАПУСК

Автотест есть, пока он, конечно, ничего не делает, но давайте попробуем его запустить:

```
1 | ./gradlew test # *nix
2 | gradlew test # Windows
```

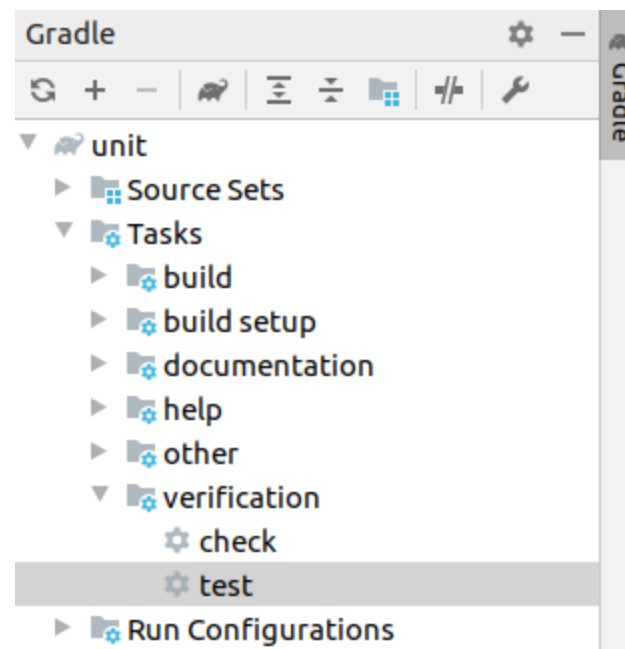
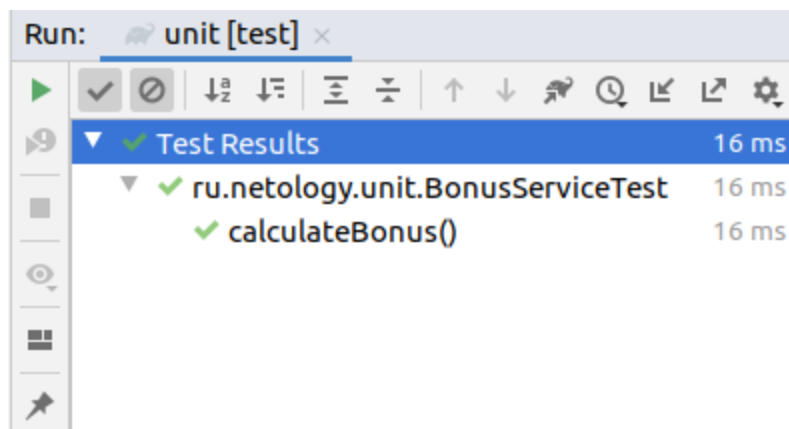
```
coursar > ... > netology > QA > unit > ./gradlew test
```

```
BUILD SUCCESSFUL in 4s
3 actionable tasks: 3 executed
```

Зелёная надпись означает, что всё ок.

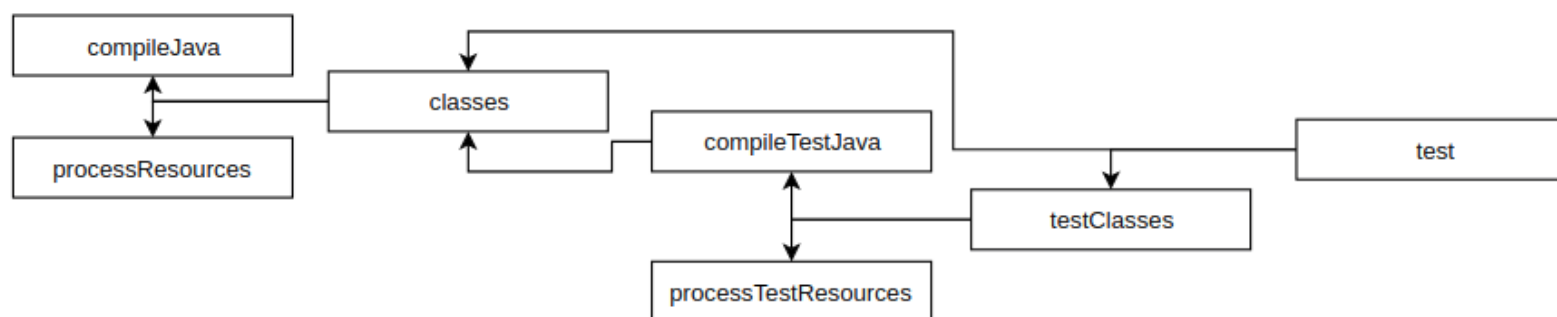
Детальное описание работы в консоли будет размещено в Support-репозитории (ознакомьтесь с ним).

# ЗАПУСК ИЗ IDEA



Но мы настоятельно рекомендуем тренироваться в запуске из консоли, т.к. вам это очень пригодится при настройке системы Continuous Integration и анализе результатов сборки.

# 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).

Детальное описание работы с Gradle будет размещено в Support-репозитории (ознакомьтесь с ним).

# АННОТАЦИИ

Давайте разбираться с самим тестом:

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.*;` ).



# ТЕСТ-КЕЙСЫ

Давайте вспомним, как мы писали тест-кейсы:

1. Описание входных данных;
2. Описание шагов;
3. Ожидаемый результат.

И при прогоне тест-кейсов отмечали также фактический результат и факт совпадения/не совпадения его с ожидаемым.

Попробуем реализовать эту же схему.



# МЕТОДОЛОГИЯ

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.

# МЕТОДОЛОГИЯ

```
1  class BonusServiceTest {
2      @Test
3      void calculateBonus() {
4          // подготовка
5          BonusService service = new BonusService();
6          int amount = 2000;
7
8          // выполнение целевого действия
9          int actual = service.calculateBonus(amount);
10         int expected = 10;
11
12         // сравнение ожидаемого и фактического
13         assertEquals(expected, actual);
14     }
15 }
```

# ASSERTIONS

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.



# FAILING

Всё здорово, наш тест запускается и работает, но как понять, тестирует ли он что-то на самом деле?

Обычно, стараются сначала специально «уронить» тест, указав неправильное ожидаемое значение. Увидеть, что он действительно падает, а затем указать уже правильное:

```
✗ Tests failed: 1 of 1 test – 17 ms
expected: <20> but was: <10>
org.opentest4j.AssertionFailedError: expected: <20> but was: <10> <5 internal calls>
  at ru.netology.unit.BonusServiceTest.calculateBonus(BonusServiceTest.java:17) <31 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <42 internal calls>
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) <1 internal call>
  at java.lang.Thread.run(Thread.java:748)
```

Рекомендуем вам потренироваться в чтении сообщений JUnit'a, чтобы научиться быстро разбираться, что и где пошло не так.

После того, как мы убедились, что тест «падает», можно вернуть его в рабочее состояние.

# ОТЧЁТЫ

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

## Test Summary

1

0

0

0.016s

100%

tests

failures

ignored

duration

successful

Packages

Classes

Package	Tests	Failures	Ignored	Duration	Success rate
<a href="#">ru.netology.unit</a>	1	0	0	0.016s	100%

## Test Summary

1	1	0	0.018s	0%
tests	failures	ignored	duration	successful

Failed tests	Packages	Classes
--------------	----------	---------

[BonusServiceTest](#). [calculateBonus\(\)](#)

## НЕСКОЛЬКО ТЕСТОВ

Итак, один тест мы осилили. Давайте попробуем написать ещё парочку. Но тут сразу возникнет проблема: мы не можем наш новый тест назвать так же как старый `calculateBonus` и «навесить» на него аннотацию `@Test`.

Значит, нужно каким-то образом договориться о тех правилах, в соответствии с которыми мы будем именовать наши тестовые методы.

# ИМЕНОВАНИЕ ТЕСТОВ

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.

# РЕАЛИЗАЦИЯ ТЕСТОВ

```
1  @Test
2  void shouldReturn10IfAmountIs2000() {
3      BonusService service = new BonusService();
4      int amount = 2000;
5
6      int actual = service.calculateBonus(amount);
7      int expected = 10;
8
9      assertEquals(expected, actual);
10 }
11
12 @Test
13 void shouldReturnZeroIfAmountLowerThan1000() {
14     BonusService service = new BonusService();
15     int amount = 900;
16
17     int actual = service.calculateBonus(amount);
18     int expected = 0;
19
20     assertEquals(expected, actual);
21 }
```

## ВАЖНО

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\*.

Примечание\*: хотя можно и на этот счёт встретить разные точки зрения.



# ИТОГИ

Итак, мы рассмотрели достаточно много важных тем на сегодня:

- поговорили об автоматизации вообще, целях и подводных камнях;
- научились подключать зависимости через Gradle;
- написали наши первые автотесты;



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.





**Задавайте вопросы и напишите отзыв о лекции!**

**АРТЕМ РОМАНОВ**

 [romanov-artyom](#)

 [@Artem\\_Telegram](#)

 [Романов Артём](#)