

Coding convention and Unit testing

Nội dung

| | | |
|-------|--|----|
| 1 | Coding convention | 2 |
| 1.1 | Kiểm tra quy tắc code với CheckStyle | 2 |
| 1.1.1 | Cài đặt CheckStyle trong Eclipse | 2 |
| 1.1.2 | Cấu hình CheckStyle | 2 |
| 1.1.3 | Thay đổi quy tắc kiểm tra | 3 |
| 1.1.4 | Thực hiện kiểm tra code với StyleCheck | 6 |
| 1.1.5 | Đọc thêm | 8 |
| 1.2 | Phân tích mã nguồn với PMD | 8 |
| 1.2.1 | Cài đặt PMD | 8 |
| 1.2.2 | Thực thi PMD từ dòng lệnh | 8 |
| 1.2.3 | Sử dụng PMD Eclipse plugin | 10 |
| 2 | Unit testing | 12 |
| 2.1 | Unit testing là gì? | 12 |
| 2.2 | Unit testing tự động với JUnit | 12 |
| 2.2.1 | Giới thiệu JUnit | 12 |
| 2.2.2 | Kiểm thử tự động với JUnit trong Eclipse | 13 |
| 2.2.3 | Đọc thêm | 17 |
| 3 | Bài tập | 17 |
| 3.1 | Kiểm thử tự động với JUnit - MyMath | 17 |
| 3.2 | Kiểm tra mã nguồn và kiểm thử lớp Triangle | 18 |
| 3.3 | Kiểm tra mã nguồn và kiểm thử lớp ShoppingCart | 18 |

1 Coding convention

Coding convention là những “qui ước” viết code, thường là chi tiết và chặt chẽ hơn các quy định về cú pháp của ngôn ngữ lập trình. Ví dụ, quy ước đặt tên biến trong Java là tên biến phải bắt đầu bằng một ký tự hoặc dấu gạch dưới, theo sau bởi các ký tự, số hoặc dấu gạch dưới. Tuy nhiên, trong qui ước “Java coding style” của Google¹ thì có nhiều qui tắc hơn như: tên lớp thì phải được đặt theo UpperCamelCase, tên phương thức, tên thuộc tính thì phải theo lowerCamelCase,... Các qui ước này giúp cho mã nguồn của chương trình nhất quán, khoa học, dễ đọc và dễ bảo trì hơn. Coding convention là không bắt buộc đối với ngôn ngữ lập trình (các trình thông dịch, biên dịch), nhưng có thể là bắt buộc đối với 1 công ty, tổ chức hay nhóm phát triển phần mềm.

Một số coding convention thông dụng:

- Oracle Java coding convention:
<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
- Google Java style guide: <https://google.github.io/styleguide/javaguide.html>
- PSR PHP coding convention: <https://www.php-fig.org/psr/psr-1/>
- PEP 8 – Style guide for Python code: <https://peps.python.org/pep-0008/>
- GNU coding standard: https://www.gnu.org/prep/standards/html_node/Writing-C.html
- MS .NET coding conventions:
<https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>

1.1 Kiểm tra quy tắc code với CheckStyle

CheckStyle là một công cụ mã nguồn mở dùng để kiểm tra code có tuân theo các quy tắc coding đã được quy định hay không? Công cụ này có thể được sử dụng thông qua một IDE (VD như Eclipse) hoặc thông qua Maven hoặc Gradle. Tài liệu này sẽ hướng dẫn sử dụng CheckStyle trong giao diện của Eclipse.

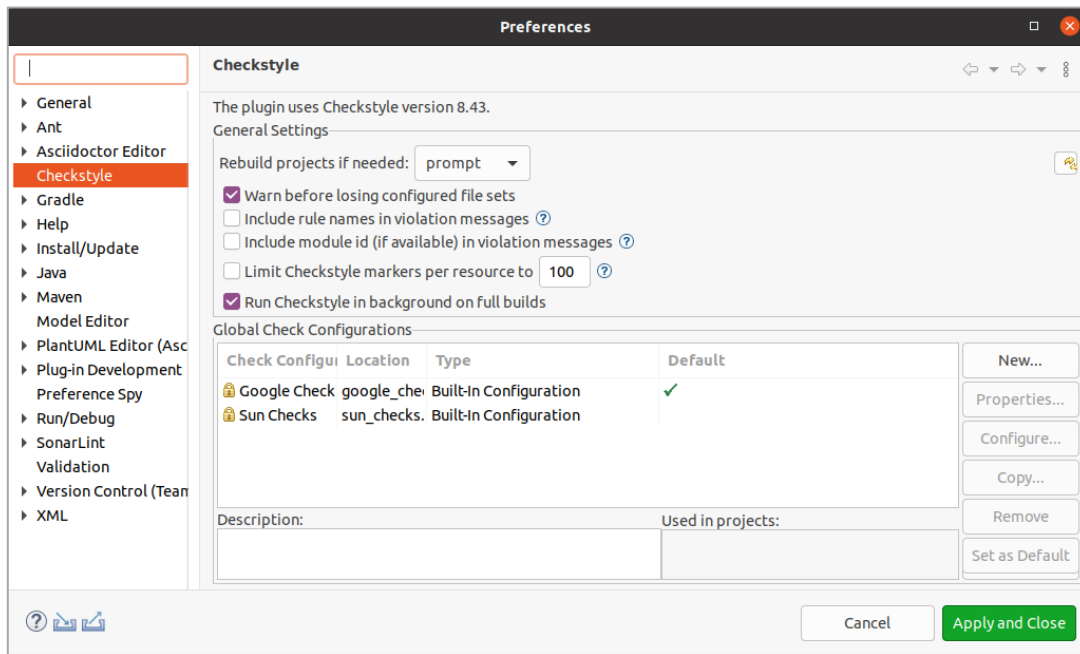
1.1.1 Cài đặt CheckStyle trong Eclipse

Để cài đặt CheckStyle trong Eclipse, ta chọn menu **Help > Install new Software** và nhập vào URL sau: <https://checkstyle.org/eclipse-cs-update-site>.

1.1.2 Cấu hình CheckStyle

Để cấu hình CheckStyle, ta chọn **Window > Preferences > Checkstyle** (hoặc trên hệ điều hành MacOS thì chọn **Eclipse > Preferences > Checkstyle**).

¹ <https://google.github.io/styleguide/javaguide.html>



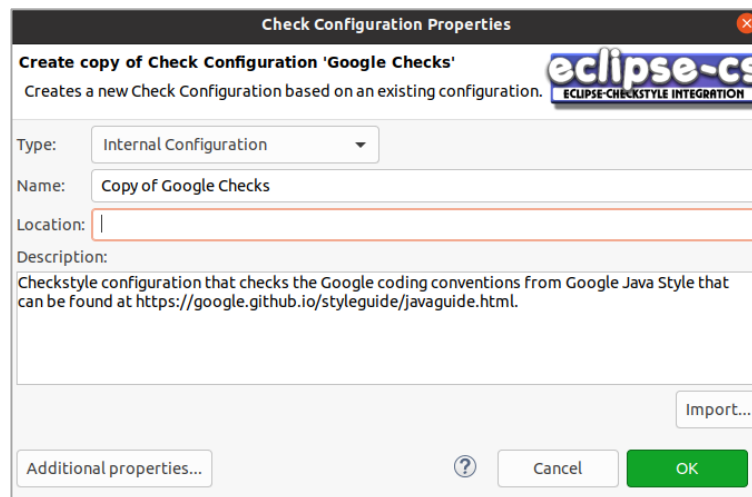
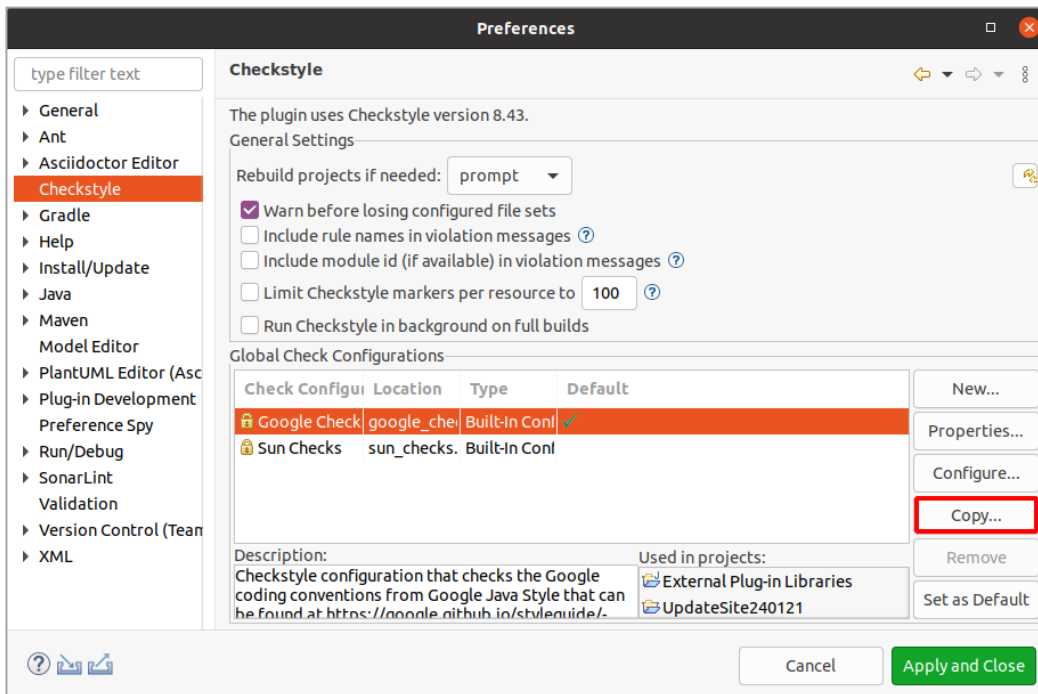
Mặc định, CheckStyle hỗ trợ 2 style mặc định là “Google Check” và “Sun Checks”. Các qui ước của Google Check có thể được tham khảo tại địa chỉ: <https://bit.ly/3CdxSB8>² và Sun Check tại <https://bit.ly/34ig4Ze>³.

1.1.3 Thay đổi quy tắc kiểm tra

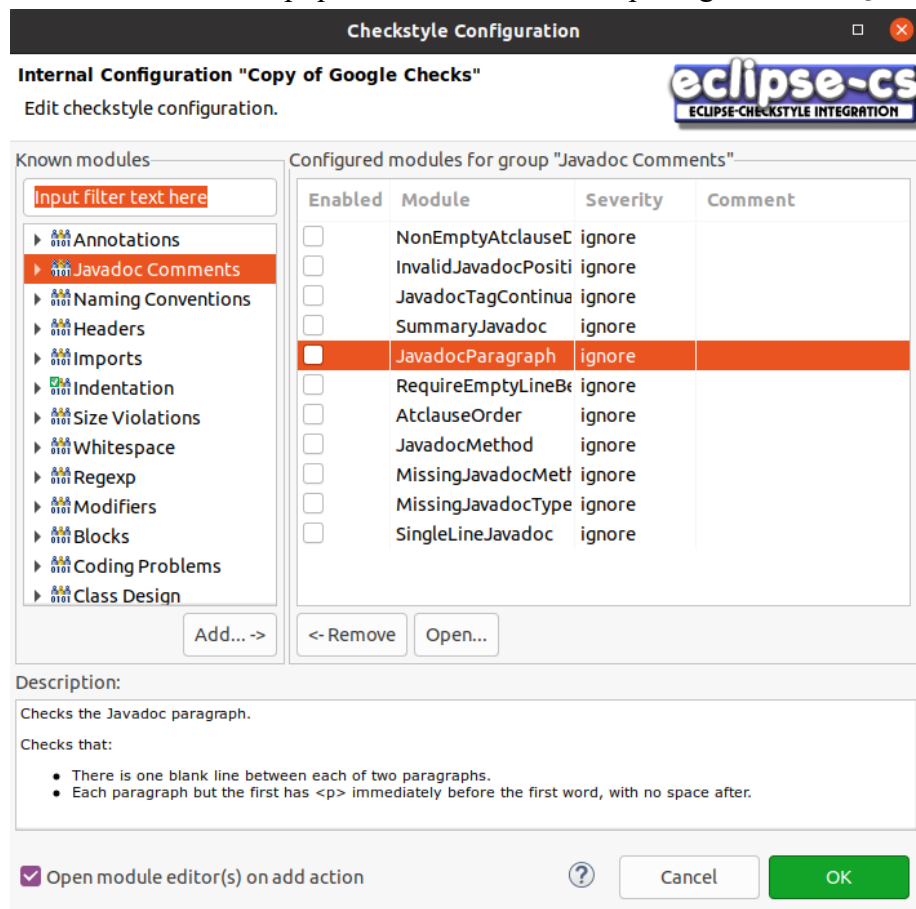
Ta có thể tự định nghĩa các qui tắc kiểm tra riêng cho StyleCheck bằng cách chọn New... trong cửa sổ Preferences của StyleCheck. Trong trường hợp ta muốn thay đổi một số qui tắc trong các style mặc định thì đầu tiên là sao chép các qui tắc chuẩn đó bằng nút Copy...

² https://checkstyle.sourceforge.io/google_style.html

³ https://checkstyle.sourceforge.io/sun_style.html

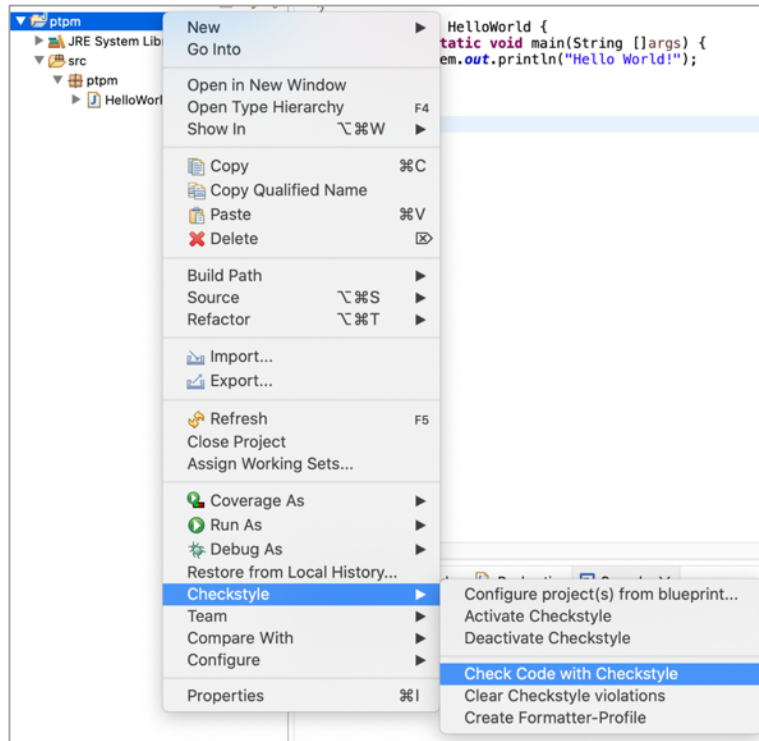


Và sau đó chỉnh sửa trên tập qui tắc mới được sao chép bằng nút Configure...

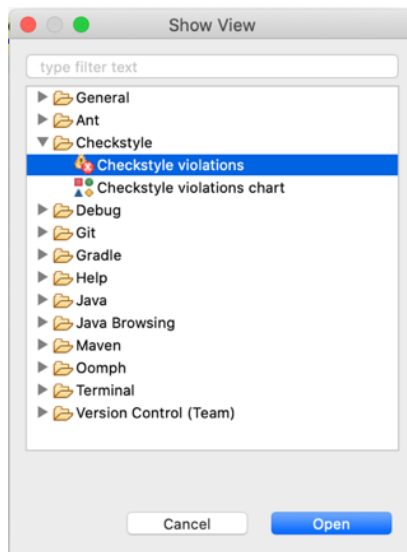


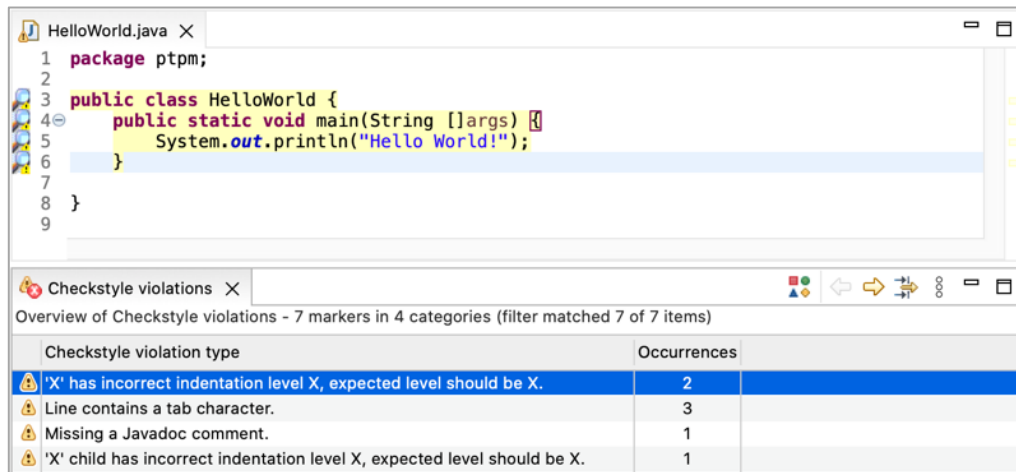
1.1.4 Thực hiện kiểm tra code với StyleCheck

Để thực hiện kiểm tra code với StyleCheck, ta right click lên project và chọn Checkstyle > Check Code with Checkstyle.

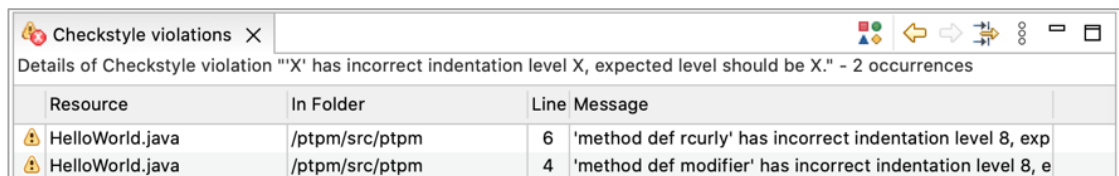


Sau khi kiểm tra xong, ta có thể xem tổng hợp kết quả (các vi phạm qui tắc) bằng cách chọn menu Window > Show > View > Others > Checkstyle





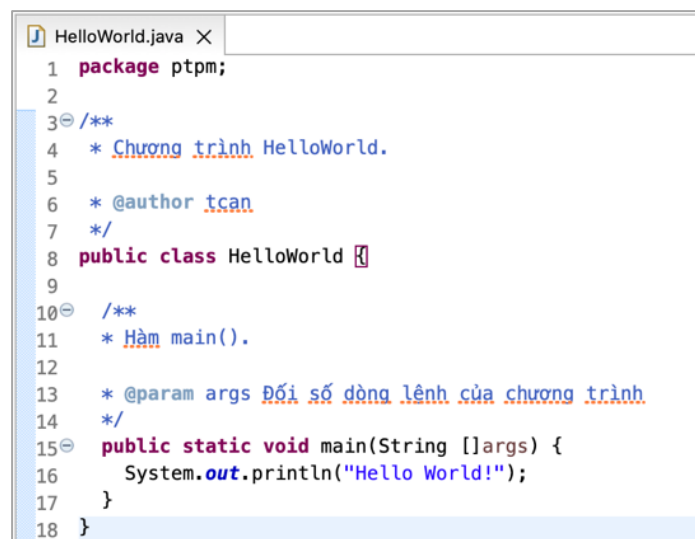
Muốn xem chi tiết của 1 vi phạm (violation), ta double-click vào violation tương ứng (dùng nút Back để trở lại màn hình chính của CheckStyle):



Một số vi phạm coding style trong ví dụ trên là:

- Canh lề (indentation) sai qui cách: trong Google Style, mỗi mức indenation là 2 khoảng trắng trong khi trong chương trình này thì dùng tab (dòng 4, 5 và 6).
- Thiếu comment cho Javadoc

Sửa lại chương trình trên như sau để fix các lỗi style (canh lề lại dùng 2 khoảng trắng cho mỗi indentation level, thêm Javadoc comment cho lớp HelloWorld và hàm main):



1.1.5 Đọc thêm

- Tùy biến rules: https://checkstyle.sourceforge.io/config_annotation.html
- Sử dụng Checkstyle và tùy biến checking rules trong Eclipse:
 - o <https://checkstyle.org/eclipse-cs/#!/custom-checks>
 - o <https://examples.javacodegeeks.com/desktop-java/ide/eclipse/eclipse-checkstyle-plugin-example/>

1.2 Phân tích mã nguồn với PMD

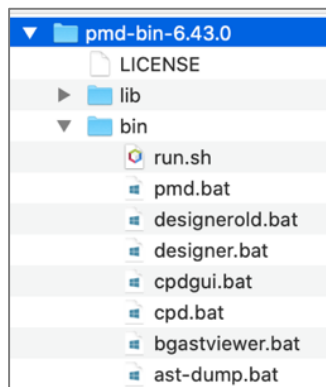
PMD (Programming Mistake Detector) là một công cụ phân tích code tĩnh (phân tích code trước khi thực thi. Công cụ này có thể tìm được các lỗi thường gặp như các biến không sử dụng, tạo các đối tượng không cần thiết, catch block không có lệnh,... Công cụ này hỗ trợ các ngôn ngữ lập trình thông dụng như Java, JavaScript, PLSQL, XML, XSL,...

Website của PMD: <https://pmd.github.io/>.

1.2.1 Cài đặt PMD

Yêu cầu: JDK 1.7 trở lên (hoặc Java 8 trở lên nếu muốn phân tích các ngôn ngữ JavaScript, Scala, VisualForce, Designer), chương trình nén/giải nén winzip hoặc 7-zip.

PMD được phân phối dưới dạng 1 file nén. Người sử dụng chỉ cần download file nén và giải nén để sử dụng. Link download: <https://github.com/pmd/pmd/releases> (download tập tin pmd-bin-xxx.zip, với xxx là phiên bản release trong phần Assets). Sau khi download xong thì ta chỉ cần giải nén tập tin này. Cấu trúc của thư mục PMD sau khi giải nén như sau:



1.2.2 Thực thi PMD từ dòng lệnh

Cú pháp thực hiện kiểm tra mã nguồn:

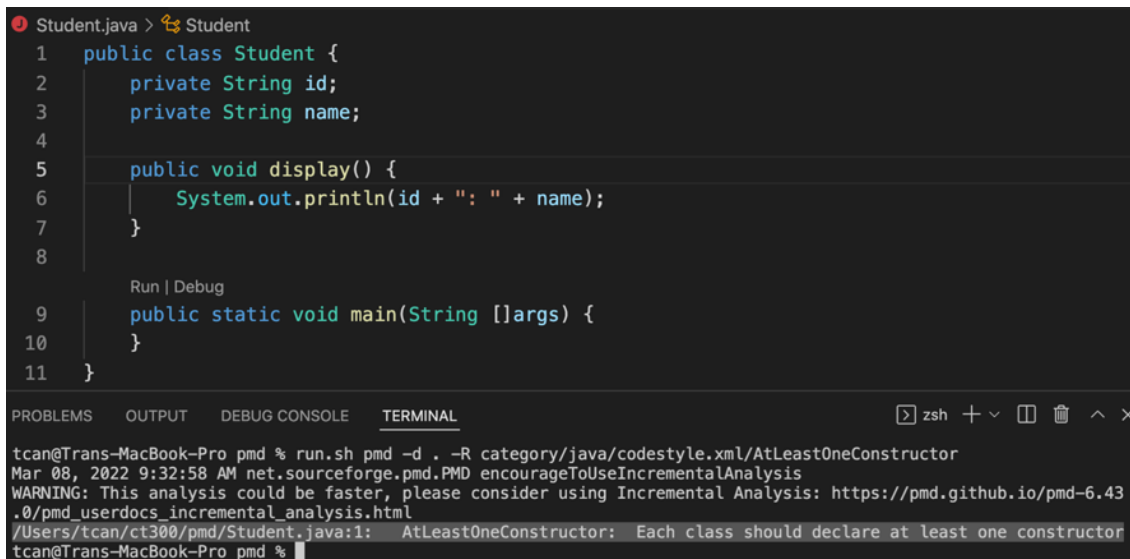
- Windows:
`pmd.bat -d <source code path> -R <rule set file path>`
- Linux/MacOS:
`run.sh pmd -d <source code path> -R <rule set file path>`

Nếu chỉ muốn kiểm tra một rule nào đó thì sử dụng `<rule set file path>`:

`category/java/codestyle/<tên rule>`

Ví dụ, để kiểm tra mỗi lớp phải có ít nhất 1 hàm xây dựng, ta sử dụng rule set file path như sau: `-R category/java/codestyle.xml/AtLeastOneConstructor`.

Ví dụ, kiểm tra lớp `Student` với rule `AtLeastOneConstructor` (mỗi lớp phải có ít nhất 1 hàm xây dựng) sẽ sinh ra 1 thông báo là lớp này thiếu hàm xây dựng.

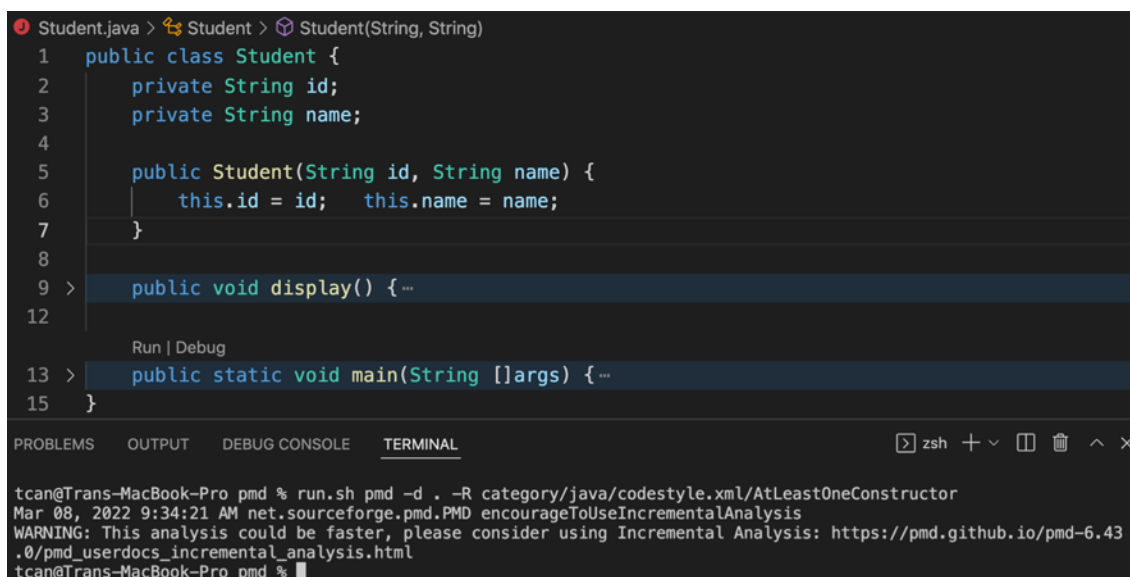


```
Student.java > Student
1 public class Student {
2     private String id;
3     private String name;
4
5     public void display() {
6         System.out.println(id + " : " + name);
7     }
8
9     public static void main(String []args) {
10
11 }

Run | Debug

tcant@Trans-MacBook-Pro pmd % run.sh pmd -d . -R category/java/codestyle.xml/AtLeastOneConstructor
Mar 08, 2022 9:32:58 AM net.sourceforge.pmd.PMD encourageToUseIncrementalAnalysis
WARNING: This analysis could be faster, please consider using Incremental Analysis: https://pmd.github.io/pmd-6.43
./pmd_userdocs_incremental_analysis.html
/Users/tcant/ct300/pmd/Student.java:1: AtLeastOneConstructor: Each class should declare at least one constructor
tcant@Trans-MacBook-Pro pmd %
```

Thêm hàm xây dựng cho lớp này và kiểm tra lại, kết quả như sau:



```
Student.java > Student > Student(String, String)
1 public class Student {
2     private String id;
3     private String name;
4
5     public Student(String id, String name) {
6         this.id = id; this.name = name;
7     }
8
9     public void display() { ...
12
13     public static void main(String []args) { ...
15 }

Run | Debug

tcant@Trans-MacBook-Pro pmd % run.sh pmd -d . -R category/java/codestyle.xml/AtLeastOneConstructor
Mar 08, 2022 9:34:21 AM net.sourceforge.pmd.PMD encourageToUseIncrementalAnalysis
WARNING: This analysis could be faster, please consider using Incremental Analysis: https://pmd.github.io/pmd-6.43
./pmd_userdocs_incremental_analysis.html
tcant@Trans-MacBook-Pro pmd %
```

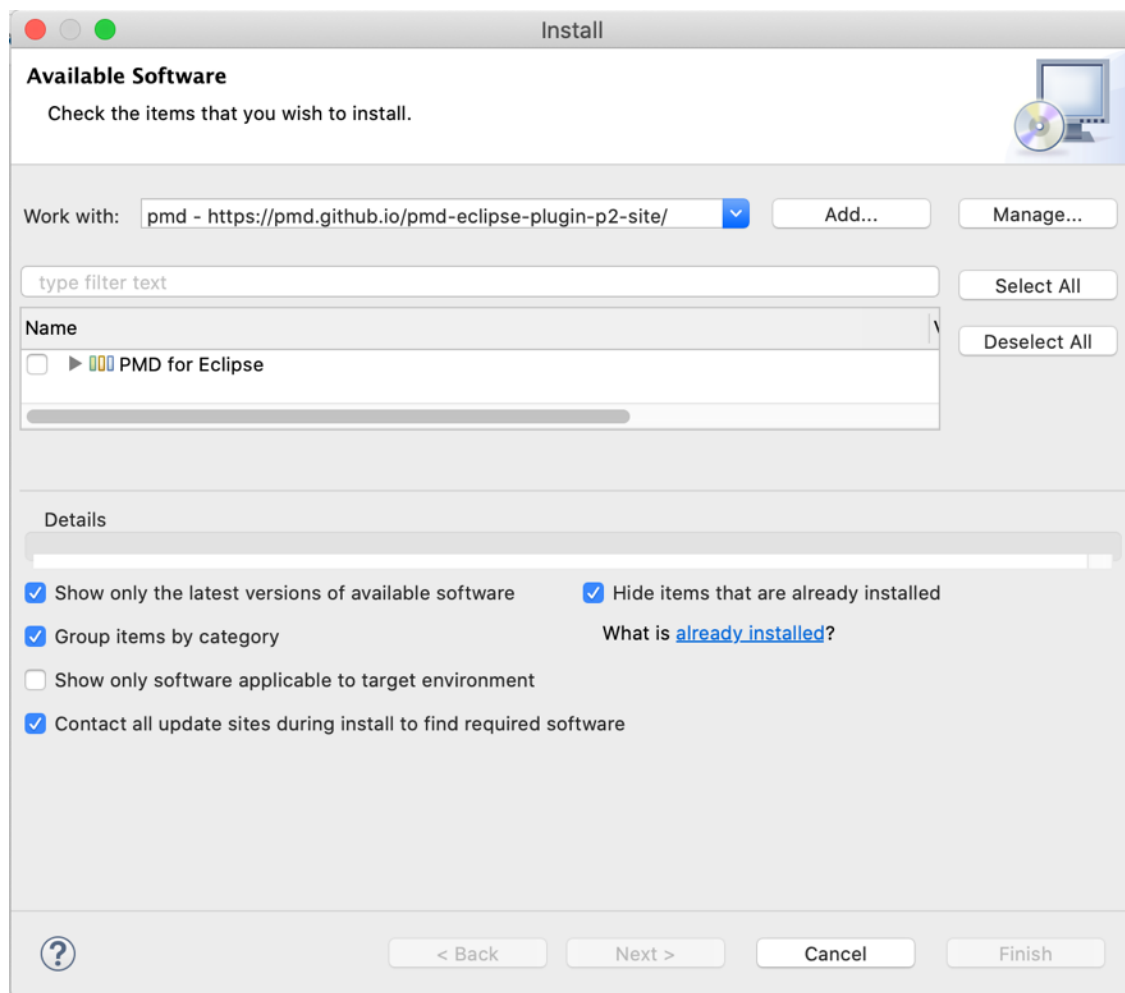
Danh sách các rule được định nghĩa sẵn trong PMD có thể được tham khảo trong link sau:

https://pmd.github.io/latest/pmd_rules_java.html

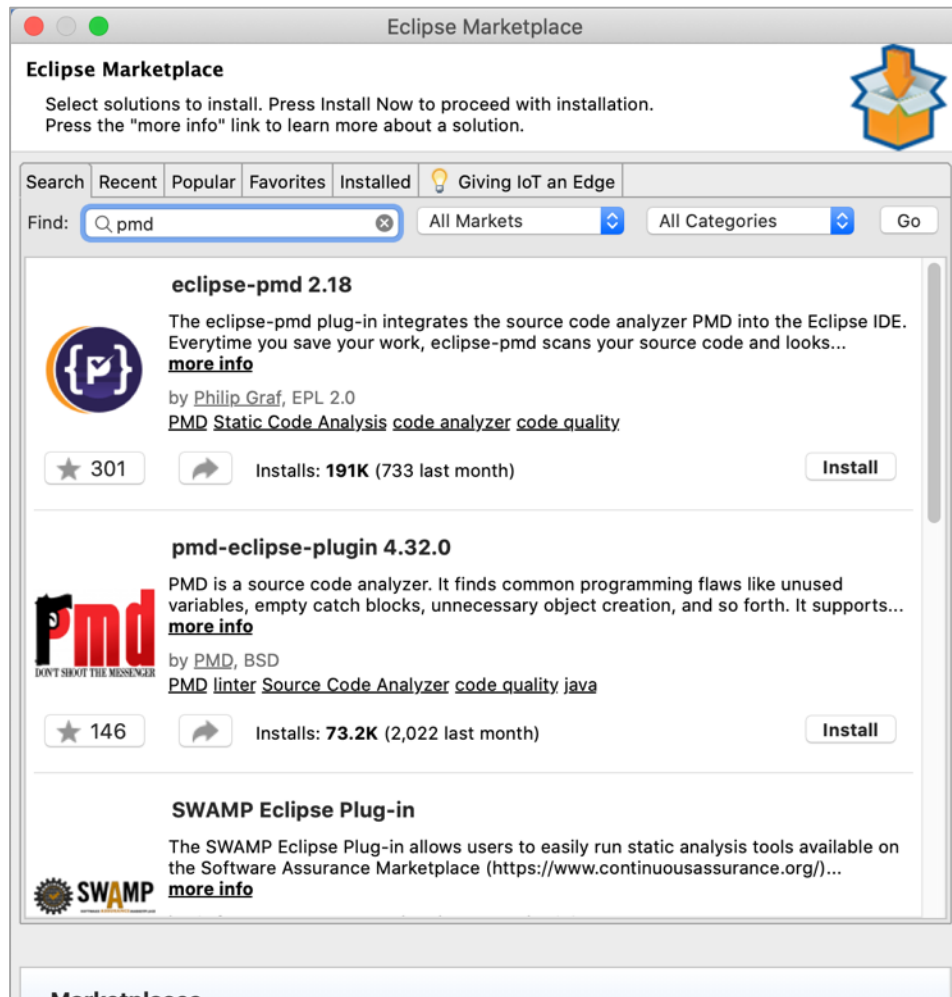
1.2.3 Sử dụng PMD Eclipse plugin

Có 2 cách cài đặt PMD plugin trong Eclipse:

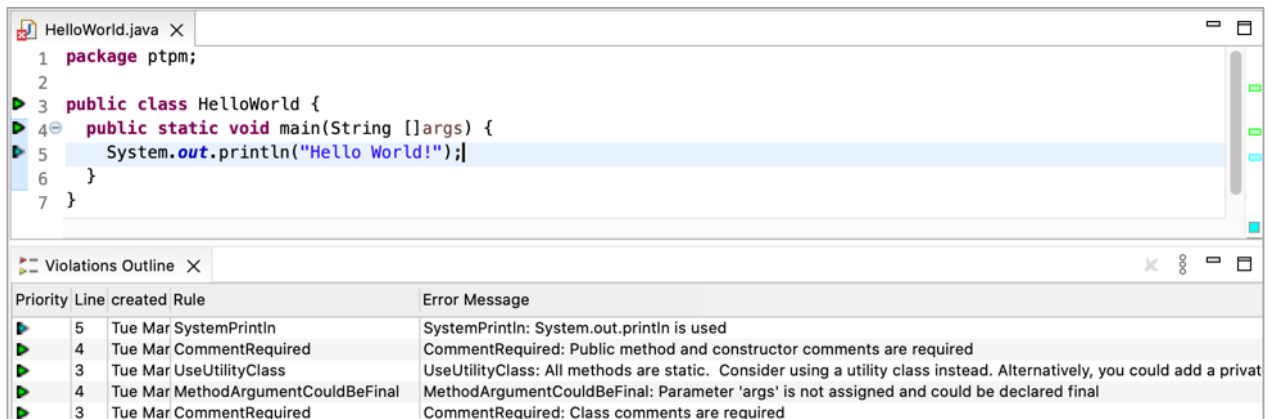
- Dùng chức năng **Install New Software**: đường dẫn để cài đặt PMD là <https://pmd.github.io/pmd-eclipse-plugin-p2-site/>



- Dùng Eclipse Marketplace:



Sau khi cài đặt PMD, ta có thể phân tích mã nguồn bằng cách right click lên tập tin cần kiểm tra và chọn PMD > Check Code



Một số lỗi trong đoạn code trên, ý nghĩa và cách khắc phục:

1. `SystemPrintln`: trong code có sử dụng `System.out.println()`
=> Nên dùng `Logger` để có thể cấu hình được.
2. `CommentRequire`: phải có comment cho lớp, phương thức.
=> Thêm comment cho lớp và các phương thức.
3. `UseUtilityClass`: lớp `HelloWorld` chỉ bao gồm các phương thức tĩnh, do đó nên được khai báo như là một lớp tiện ích.
=> Khai báo class là `final` hoặc thêm hàm xây dựng `private`.
4. `MethodArgumentCouldBeFinal`: đối số `args` của hàm `main()` không thay đổi trong hàm.
=> Nên khai báo đối số là `final`.

2 Unit testing (<https://www.vogella.com/tutorials/JUnit/article.html>)

2.1 Unit testing là gì?

Unit test (kiểm thử đơn vị) là một loại kiểm thử phần mềm trong đó các đơn vị hay thành phần riêng lẻ của phần mềm như hàm, phương thức, lớp,... sẽ được kiểm thử để kiểm tra các thành phần này có hoạt động đúng theo yêu cầu hay không. Mục đích của nó nhằm đảm bảo mỗi đoạn code (mỗi phần của chương trình) đều hoạt động hoàn hảo. Trong chu trình phát triển phần mềm thì kiểm thử đơn vị là lần kiểm tra đầu tiên trước khi tiến hành kiểm tra tích hợp (integration testing). Kiểm thử đơn vị là phương pháp kiểm thử whitebox, thường được thực hiện bởi người phát triển phần mềm (developer).

2.2 Unit testing tự động với JUnit

2.2.1 Giới thiệu JUnit

JUnit là một framework mã nguồn mở, miễn phí, đơn giản dùng để thực hiện unit test tự động cho ngôn ngữ lập trình Java. Phiên bản thông dụng nhất là JUnit 4 và phiên bản mới nhất là JUnit 5 cho phép thực hiện cả unit testing và integration testing. Trong tài liệu này, chúng ta sẽ sử dụng JUnit 4 để thực hiện Unit test.

Một số tính năng chính của JUnit:

- Cung cấp các annotation để định nghĩa các phương thức kiểm thử.
- Cung cấp các Assertion để kiểm tra kết quả mong đợi.
- Cung cấp các Test runner để thực thi các test scripts.
- Thực hiện tự động các test cases.

- Cho phép nhóm các test cases thành các test suite.
- Hiện thị kết quả trực quan.

Một số khái niệm cơ bản trong JUnit:

- **Test case:** là 1 “trường hợp” kiểm thử. Mỗi chức năng (đơn vị chương trình) thường có nhiều test cases, mỗi test case sẽ kiểm thử 1 trường hợp nào đó. Ví dụ: để kiểm thử hàm giải phương trình bậc 1 thì có thể có các test cases như:
 - o Trường hợp cả a và b đều bằng 0 (vô số nghiệm).
 - o Trường hợp a bằng 0 và b khác không (vô nghiệm).
 - o Trường hợp a và b khác không (phương trình có 1 nghiệm).
- **Setup:** là hàm được thực hiện trước khi chạy các test case, thường dùng để chuẩn bị dữ liệu để chạy các test case.
- **Teardown:** là hàm được thực hiện sau khi các test case chạy xong, thường dùng để dọn dẹp dữ liệu, giải phóng tài nguyên, ...
- **Assert:** dùng để kiểm tra tính đúng đắn của unit. Mỗi test case sẽ có một hoặc nhiều câu lệnh Assert.
- **Mock:** là một đối tượng ảo, mô phỏng các tính chất và hành vi của đối tượng thực được truyền vào bên trong khối mã đang vận hành nhằm kiểm tra tính đúng đắn của các hoạt động bên trong. Đây là cơ chế cho phép ta test 1 unit phụ thuộc vào unit khác khi unit đó chưa thực hiện xong. Giả sử chương trình của chúng ta được chia làm 2 module: A và B. Module A đã code xong, B thì chưa. Để test module A, ta dùng mock để làm giả module B, không cần phải đợi tới khi module B code xong mới test được.
- **Test suite:** là một tập các test case và nó cũng có thể bao gồm nhiều test suite khác.

2.2.2 Kiểm thử tự động với JUnit trong Eclipse

JUnit được tích hợp sẵn trong Eclipse nên ta có thể sử dụng mà không cần cài đặt gì thêm. Để minh họa kiểm thử tự động với JUnit, ta tạo 1 lớp với 2 phương thức (unit) như sau:

- `int add(int n1, int n2)`: tính tổng 2 số nguyên.
- `int divide(int n1, int n2)`: tính phép chia nguyên hai số n1 và n2. Nếu n2 bằng 0 thì trả về giá trị ngoại lệ.

```
MathUtil.java X
1 package ptpm;
2
3 public final class MathUtil {
4     private MathUtil() { }
5
6     public static int add(int n1, int n2) {
7         return n1 + n2;
8     }
9
10    public static int divide(int n1, int n2) {
11        if (n2 == 0) {
12            throw new IllegalArgumentException("Cannot divide by zero");
13        }
14
15        return n1 / n2;
16    }
17 }
```

Để kiểm thử 2 phương thức trên, đầu tiên ta tạo 1 lớp với tên là `MathUtilTest`. Đây là một naming convention của JUnit. Class này nên được đặt trong thư mục test, cùng package với class cần test để dễ quản lý.

Tiếp theo, tạo các test case để test các trường hợp có thể có của một phương thức. Mỗi test case nên tạo một phương thức để kiểm tra:

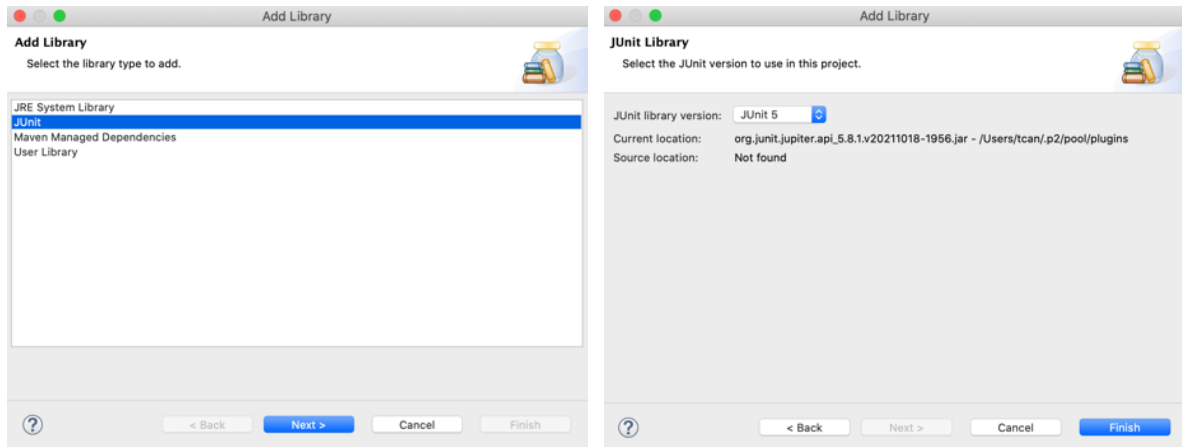
- Phương thức `divide()` có thể có 1 số trường hợp test sau: trường hợp kết quả phép chia ra một số nguyên, trường hợp kết quả phép chia ra số lẻ, trường hợp `n2` bằng 0.
- Phương thức `add()`: chỉ đơn giản kiểm tra kết quả cộng 2 số.

```
6 public class MathUtilTest {
7
8     @Test
9     public void divide_SixDividedByTwo() {
10         final int expected = 3;
11         final int actual = MathUtil.divide(6, 2);
12
13         Assert.assertEquals(expected, actual);
14     }
15
16     @Test
17     public void divide_OneDividedByTwo() {
18         final int expected = 0;
19         final int actual = MathUtil.divide(1, 2);
20
21         Assert.assertEquals(expected, actual);
22     }
23
24     @Test(expected = IllegalArgumentException.class)
25     public void divide_OneDividedByZero() {
26         MathUtil.divide(1, 0);
27     }
28
29     @Test
30     public void add_SixAddedByTwo() {
31         final int expected = 8;
32         final int actual = MathUtil.add(6, 2);
33
34         Assert.assertEquals(expected, actual);
35     }
36 }
37
```

Lưu ý:

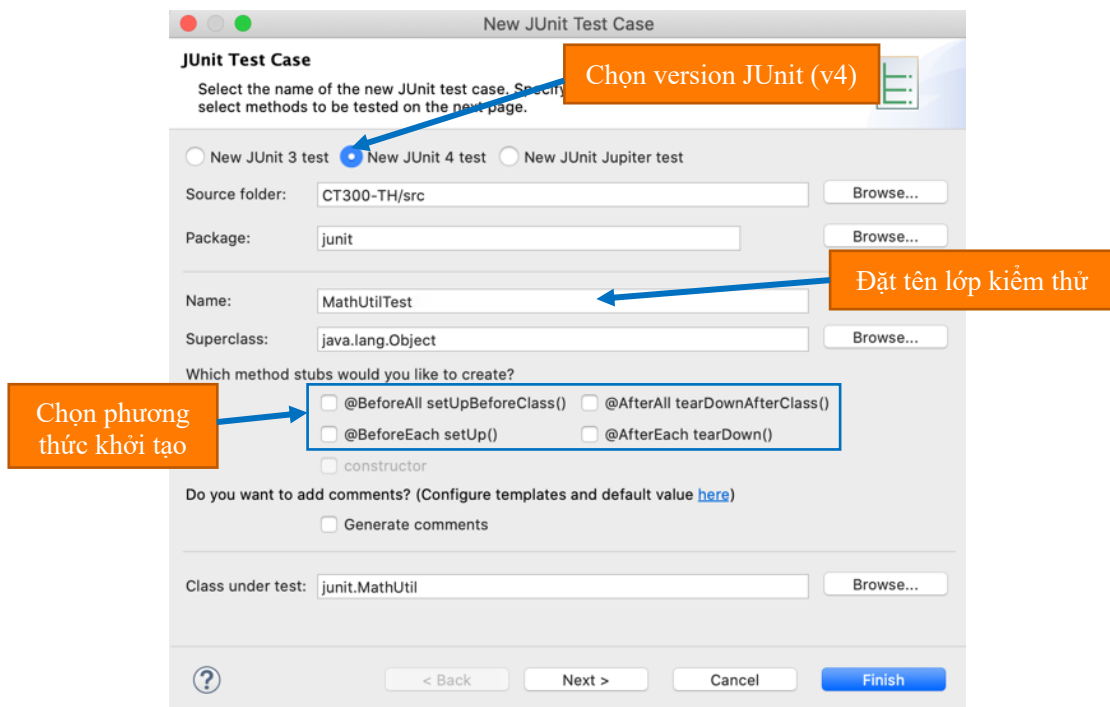
- Mỗi phương thức cài đặt một test case phải được đặt 1 annotation `@Test`.
- Ngoài ra, ta còn phải khai báo sử dụng thư viện JUnit cho project bằng cách right click lên project và chọn Build Path > Add Libraries ...

Sau đó chọn JUnit và chọn version 4 (JUnit4) như hình minh họa bên dưới.

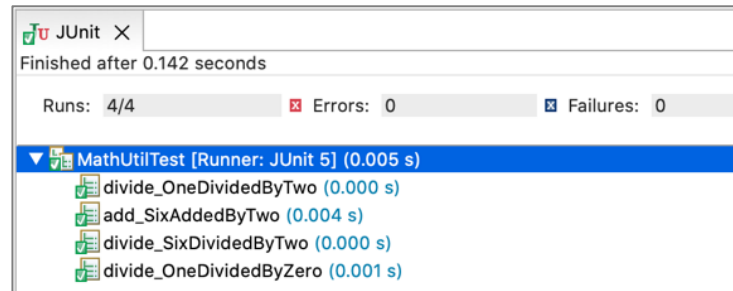


- Ta cũng có thể thêm 1 lớp kiểm thử cho một lớp bằng cách right click lên lớp cần test và chọn New > JUnit Test Case... (thực hiện cách này thì Eclipse sẽ tự động thêm thư viện JUnit vào project nếu như ta chưa thêm trước đó).

Sau đó cấu hình cho lớp kiểm thử như sau:



Như vậy, để kiểm thử 2 phương thức trên thì ta có tổng cộng 4 test cases. Để chạy các test cases này, ta right-click lên lớp MathUtilTest và chọn Run As > Unit Test. Kết quả thực hiện các test cases trên như sau:



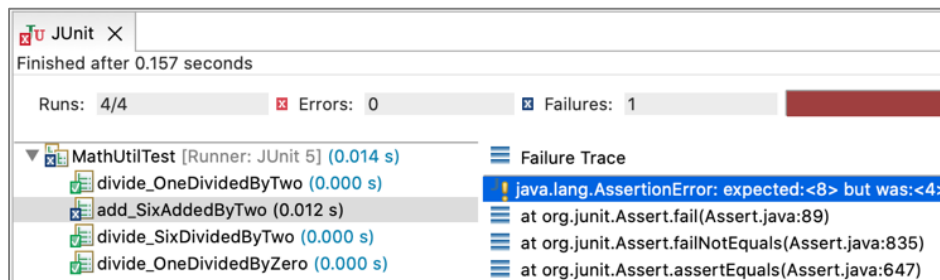
Trong đó:

- Run: số test case đã thực hiện (4 test case).
- Errors: số test case bị lỗi khi thực hiện (0 test case) => cần sửa test case.
- Failure: số test case bị thất bại (kết quả mong đợi không như kết quả thực tế) => cần kiểm tra lại mã phương thức được test.

Để kiểm thử trường hợp các phương thức bị sai, ta sửa phương thức add () lại như sau:

```
public static int add(int n1, int n2) {
    return n1 - n2;
}
```

Và kết quả kiểm thử là:



Ngoài ra, ta cũng có thể kiểm tra độ bao phủ của các test cases với plugin **EclEmma** đã được tích hợp sẵn trong Eclipse bằng cách right click lên tập tin test chứa các test cases và chọn Coverage As > JUnit Test.

| Element | Coverage | Covered Instructions | Missed Instructions | Total Instructions |
|-------------------|----------|----------------------|---------------------|--------------------|
| ptpm | 76.7 % | 46 | 14 | 60 |
| src | 76.7 % | 46 | 14 | 60 |
| ptpm | 76.7 % | 46 | 14 | 60 |
| HelloWorld.java | 0.0 % | 0 | 4 | 4 |
| MathUtil.java | 100.0 % | 15 | 0 | 15 |
| MathUtil | 100.0 % | 15 | 0 | 15 |
| add(int, int) | 100.0 % | 4 | 0 | 4 |
| divide(int, int) | 100.0 % | 11 | 0 | 11 |
| MathUtilTest.java | 75.6 % | 31 | 10 | 41 |


```
MathUtilTest.java MathUtil.java X
1 package ptpm;
2
3 public final class MathUtil {
4     private MathUtil() { }
5
6     public static int add(int n1, int n2) {
7         return n1 + n2;
8     }
9
10    public static int divide(int n1, int n2) {
11        if (n2 == 0) {
12            throw new IllegalArgumentException("Cannot divide by zero");
13        }
14
15        return n1 / n2;
16    }
17 }
```

2.2.3 Đọc thêm

- Các phương thức khởi tạo:
 - o @BeforeAll: thực hiện phương thức trước tất cả các test cases.
 - o @BeforeEach: thực hiện phương thức trước mỗi test case.
 - o @AfterEach: thực hiện phương thức sau mỗi test case.
 - o @AfterAll: thực hiện phương thức sau tất cả các test cases.
- Thực hiện có điều kiện các test case: @Enabled/DisableForJreRange, @Enabled/DisabledOnJre, @Enabled/DisableOnOs,...
- Tham số hóa (parameterized) các test case để test cho nhiều giá trị dữ liệu test.

3 Bài tập

3.1 Kiểm thử tự động với JUnit - MyMath

Tạo một lớp MyMath như sau:

```
public class MyMath {
    public int multiply(int x, int y) {
        if (x > 999) {
            throw new IllegalArgumentException("X should be less than 1000");
        }
        return x * y;
    }
}
```

Và một lớp unit test cho lớp trên như sau (SV tự thêm vào các chỉ thị import cần thiết):

```
class MyMathTest {
    @Test(expected=IllegalArgumentException.class)
    void testExceptionIsThrown() {
        MyMath tester = new MyMath();
        tester.multiply(1000, 5);
    }
}
```

```
@Test
void testMultiply() {
    MyMath tester = new MyMath();
    assertEquals(50, tester.multiply(10, 5), "10 x 5 must be 50");
}
}
```

Hãy thực hiện các yêu cầu sau:

- Thực thi các test cases và cho biết có số test case thành công và số test case thất bại?
- Sửa lại code để sửa các lỗi được phát hiện.
- [Nâng cao]** Sử dụng hàm khởi tạo `@BeforeEach` (setup) để đơn giản hóa test code.
Gợi ý: gom các phần chung của các test cases, đặt vào hàm khởi tạo `@BeforeEach`.

3.2 Kiểm tra mã nguồn và kiểm thử lớp Triangle

Tạo 1 lớp Triangle có các thuộc tính là độ dài 3 cạnh và 1 phương thức xây dựng và 1 hàm `isTriangle()` để kiểm tra xem đó có phải là 1 tam giác hay không:

- Nếu không phải là tam giác thì trả về -1.
Gợi ý: ba đoạn thẳng a, b, c có thể tạo thành 1 tam giác nếu $(a + b > c) \&\& (a + c > b) \&\& (b + c > a)$.
- Trả về 0 nếu là tam giác thường, 1 nếu là tam giác cân và 2 nếu là tam giác đều.

Sau đó, thực hiện các yêu cầu sau:

- Tạo các test cases để kiểm tra phương thức `isTriangle()` và kiểm tra độ bao phủ của các test cases với plugin EcJemma.
- Kiểm tra lớp Triangle với CheckStyle và PMD và sửa lại mã nguồn để không còn bị các lỗi coding convention.

3.3 [Làm thêm] Kiểm tra mã nguồn và kiểm thử lớp ShoppingCart

Thực hiện unit test cho chương trình ShoppingCart (giỏ hàng) được mô tả như sau:

- Khởi tạo, giỏ hàng có 0 sản phẩm.
- Khi có 1 sản phẩm được thêm vào:
 - Số lượng sản phẩm của giỏ hàng phải được tăng lên 1.
 - Giá trị của giỏ hàng phải được tăng thêm giá trị của sản phẩm mới được thêm vào.
- Khi có 1 sản phẩm được bỏ ra khỏi giỏ hàng: số sản phẩm và giá trị giỏ hàng phải được giảm tương ứng.
- Khi xóa 1 sản phẩm không có trong giỏ hàng, một ngoại lệ `ProductNotFoundException` phải được quăng ra.

Link download chương trình ShoppingCart: <https://bit.ly/3wkniHT>