



# Plugin EclEmma 2.2.1

## Apoio ao Teste Estrutural

Auri Marcelo Rizzo Vincenzi<sup>1</sup>, Márcio Eduardo Delamaro<sup>2</sup> e  
José Carlos Maldonado<sup>2</sup>

<sup>1</sup>Instituto de Informática  
Universidade Federal de Goiás

<sup>2</sup>Instituto de Ciências Matemáticas e de Computação  
Universidade de São Paulo



# Organização

## Ferramenta EclEmma

- Discussão

- Pre-requisitos

- Instalação

- A Ferramenta

- O Exemplo

## Instrumentação Off-line

- Instrumentação *off-line* com Maven e JaCoCo

- Executando a Aplicação Instrumentada

## Gerando o Relatório de Cobertura

## Outras Ferramentas Similares

## Exercício







## Como Implementar??? (2)

- ▶ Quais os passos essa ferramenta deveria executar?



## Como Implementar??? (2)

- ▶ Quais os passos essa ferramenta deveria executar?
- ▶ Você é capaz de visualizar alguma otimização nesse processo?  
O que pode ser melhorado?



# Pre-requisitos

- ▶ Conhecimento da Linguagem Java.
- ▶ Conhecimento de Critérios de Teste Estruturais (Fluxo de Controle).
- ▶ Kit de Desenvolvimento Java – Versão 1.5 ou superior.
- ▶ Ambiente de desenvolvimento Eclipse 3.5 ou superior.
- ▶ Ferramenta EclEmma – Versão 2.2.1 ou superior.







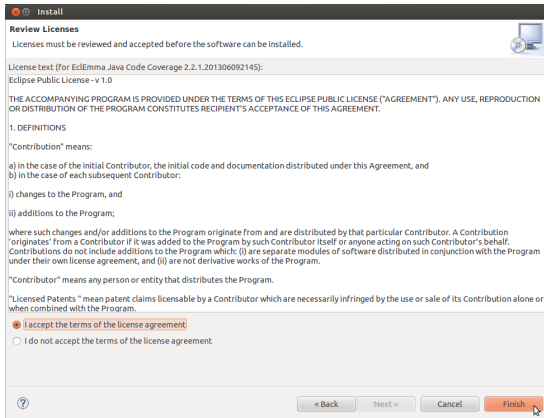








# Instalação da EclEmma (7)



## Instalação da EclEmma (3)

- ▶ EclEmma pode ser executada de várias formas distintas
  - ▶ Execução direta com a instrumentação ocorrendo **on-the-fly** via Eclipse.
  - ▶ Execução via script Ant (<http://ant.apache.org/>).
  - ▶ Execução via script Maven (<http://maven.apache.org/>)
- ▶ Este material ilustra a utilização via execução direta e via Maven.



# A Ferramenta EclEmma

- ▶ Ferramenta de código aberto para o teste de programas Java baseada na biblioteca JaCoCo (<http://www.eclemma.org/jacoco/>).
- ▶ Passos básicos para uso da ferramenta:
  - ▶ Instrumentar as classes a serem testadas.
  - ▶ Executar as classes instrumentadas com os casos de testes.
    - ▶ Importa casos de testes do JUnit.
    - ▶ Aceita também casos de testes em outros formatos. Basta executar as classes instrumentadas. Cada execução corresponde a um novo caso de teste.
  - ▶ Gerar relatórios de cobertura.
  - ▶ Com base nos relatórios decidir por continuar ou não com os testes.
- ▶ O processo de instrumentação pode ocorrer **on-the-fly** ou **off-line**.

# Programa Exemplo (1)

- ▶ Programa Identifier
- ▶ Código: Identifier.java e IdentifierMain.java
- ▶ Classe principal: Identifier

## Programa Exemplo (2)

- ▶ Para executar os testes do JUnit no Identifier e avaliar qual a cobertura de código obtida, basta seguir os passos abaixo:
  - ▶ Abra ou crie um projeto para o Identifier no Eclipse (preferencialmente um Projeto Maven)
  - ▶ Clique com o botão direito sobre a classe de teste IdentifierTestCase.java, por exemplo.
  - ▶ Entre no menu **Coverage As->JUnit Test**
    - ▶ Nesse momento, as classes do código fonte são compiladas e, antes de serem carregadas para execução, é feita a instrumentação das mesmas diretamente no bytecode gerado.
    - ▶ Com isso, os casos de teste do JUnit são executados nas classes instrumentadas e a cobertura pode ser monitorada.
    - ▶ O resultado da execução é apresentado a seguir.

# Programa Exemplo (3)

The screenshot displays an IDE window for a Java project named 'Identifier'. The main editor shows the source code for 'Identifier.java' with line numbers 1 through 10. The code defines a public class 'Identifier' with a method 'validateIdentifier(String s)' that returns a boolean. The method includes logic to check if the string length is greater than 0, and if so, it checks the first character and updates the 'valid\_id' flag and an index 'i'.

```

1 public class Identifier {
2     public boolean validateIdentifier(String s) {
3         char achar;
4         boolean valid_id = false;
5         if (s.length() > 0) {
6             achar = s.charAt(0);
7             valid_id = valid_s(achar);
8             if (s.length() > 1) {
9                 achar = s.charAt(1);
10                int i = 1;

```

On the left, the Package Explorer shows a JUnit test runner for 'IdentifierTest.java' with a success message: 'Finished after 0,024 seconds. Runs: 6/6 (1 ignored) Errors: 0 Failures: 0'.

On the right, the Outline view shows the class structure:

- Identifier
  - validateIdentifier(String): boolean
  - valid\_s(char): boolean
  - valid\_f(char): boolean

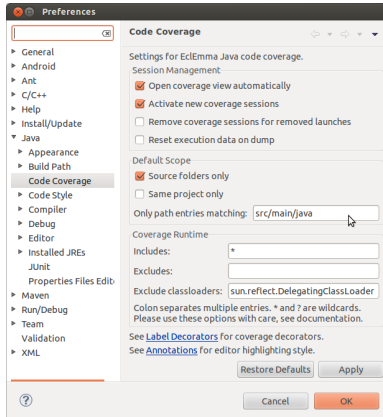
At the bottom, the Coverage view displays a table for 'IdentifierTest Case (09/02/2014 21:56:34)'. The table lists elements and their coverage statistics:

Element	Coverage	Covered Instruct	Missed Instrui	Total Instructions
Identifier	35,4%	158	288	446
src/test/java	19,8%	64	260	324
(default package)	19,8%	64	260	324
IdentifierTest2.java	0,0%	0	108	108
ParameterizedTestCase.java	0,0%	0	74	74
MyTestRunner.java	0,0%	0	65	65
IdentifierTest.java	86,5%	64	10	74
TodosTestes.java	0,0%	0	3	3
src/main/java	77,0%	94	28	122
(default package)	77,0%	94	28	122
IdentifierMain.java	0,0%	0	28	28
Identifier.java	100,0%	94	0	94

## Programa Exemplo (4)

- ▶ Observe que as classes de teste também foram incluídas na análise.
- ▶ Com isso, o número de requisitos aumenta e o nível de cobertura diminui.
- ▶ Em geral não se deseja medir a cobertura do código de teste, apenas da aplicação.
- ▶ Para isso, é possível configurar quais classes devem ser consideradas.
- ▶ Entre em **Window->Preferences** e escolha **Java->Code Coverage**.
- ▶ Na tela que irá abrir, preencha o campo **Only path entries matching:** e preencha com `src/main/java`.

# Programa Exemplo (5)



## Programa Exemplo (6)

- ▶ Clique no duplo “X” na janela de relatório (**Coverage**) para remover todas as sessões de execução.
- ▶ Execute novamente o conjunto de teste do JUnit utilizando a opção **Coverage As->JUnit Test**.

The screenshot shows an IDE window with the following components:

- Code Editor:** Displays the source code for `Identifier.java`. The code includes a `validateIdentifier` method that checks string length and iterates through characters to validate them.
- Package Explorer:** Shows the project structure with `IdentifierTest.java` selected under `src/test/java`.
- Console:** Shows the output of the JUnit test: "Finished after 0.024 seconds", "Runs: 6/6 (1 ignored)", "Errors: 0", "Failures: 0", and "IdentifierTestCase [Runner: JUnit-4] (0,001 s)".
- Coverage View:** A table showing coverage data for the test run on 09/02/2014 at 21:56:34.
 

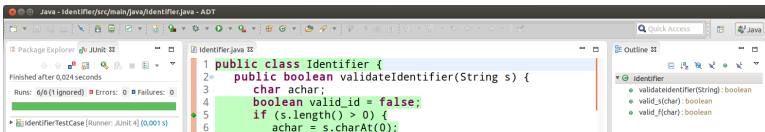
Element	Coverage	Covered Instruct	Missed Instruct	Total Instructions
Identifier	35,4 %	158	288	446
src/test/java	19,8 %	64	260	324
(default package)	19,8 %	64	260	324
IdentifierTest2.java	0,0 %	0	108	108
ParameterizedTestCase.java	0,0 %	0	74	74
MyTestRunner.java	0,0 %	0	65	65





## Programa Exemplo (8)

- ▶ EclEmma apresenta relatórios de cobertura para os seguintes critérios:
  - ▶ Contagem de Instruções (bytecode)
  - ▶ Contagem de Desvios
  - ▶ Contagem de Linhas (código-fonte)
  - ▶ Contagem de Métodos
  - ▶ Contagem de Tipos
  - ▶ Contagem de Complexidade
- ▶ Uma definição de cada um desses critérios pode ser encontrada em <http://www.eclemma.org/jacoco/trunk/doc/counters.html>



# Programa Exemplo (9)

The screenshot shows an IDE window titled "Java - Identifier/src/main/java/Identifier.java - ADT". The main editor displays the following Java code:

```

1 public class Identifier {
2     public boolean validateIdentifier(String s) {
3         char achar;
4         boolean valid_id = false;
5         if (s.length() > 0) {
6             achar = s.charAt(0);
7             valid_id = valid_s(achar);
8             if (s.length() > 1) {
9                 achar = s.charAt(1);
10                int i = 1;
11                while (i < s.length()) {
12                    achar = s.charAt(i);
13                    if (!valid_f(achar))

```

Below the code editor, a "Coverage" window is open, displaying a table for the test case "IdentifierTestCase (09/02/2014 22:08:26)".

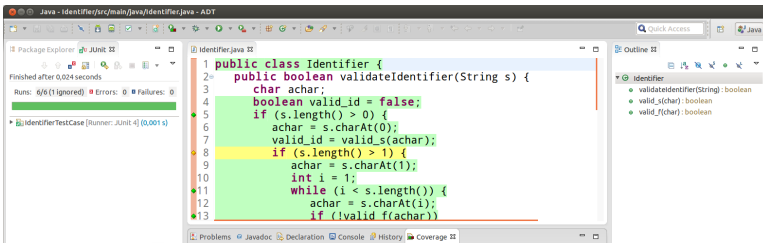
Element	Coverage	Covered Instruct	Missed Instruct	Total Instruct
Identifier	77,0%	94	28	
src/main/java	77,0%	94	28	
(default package)	77,0%	94	28	
IdentifierMain.java	0,0%	0	28	
Identifier.java	100,0%	94	0	

A context menu is open over the Coverage window, listing various options:

- Show Projects
- Show Package Roots
- Show Packages
- Show Types
- Instruction Counters
- Branch Counters
- Line Counters
- Method Counters
- Type Counters
- Complexity
- Hide Unused Elements

## Programa Exemplo (10)

- ▶ Para gerar o relatório de cobertura, basta clicar com o botão direito na área do relatório da interface gráfica e escolher a opção `ExportSession...`
- ▶ Em seguida, basta navegar até o local onde se deseja gerar o relatório, escolher o formato do mesmo e clicar em `Finish`.
- ▶ A sequência de telas para isso é mostrada a seguir, bem como um exemplo do relatório gerado.

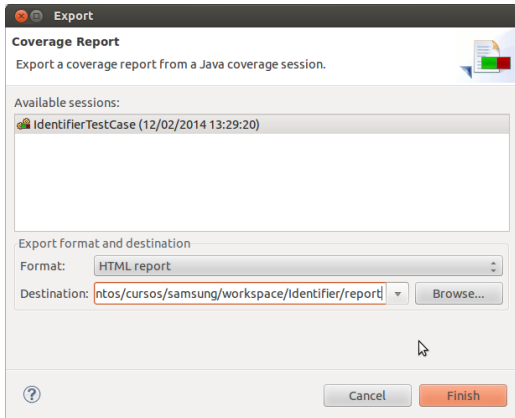


```
1 public class Identifier {
2= public boolean validateIdentifier(String s) {
3     char achar;
4     boolean valid_id = false;
5     if (s.length() > 0) {
6         achar = s.charAt(0);
7         valid_id = valid_s(achar);
8         if (s.length() > 1) {
9             achar = s.charAt(1);
10            int i = 1;
11            while (i < s.length()) {
12                achar = s.charAt(i);
13                if (!valid_f(achar))
```

The screenshot shows an IDE window titled "Java - Identifier/src/main/java/Identifier.java - ADT". The main editor displays the code for the `Identifier` class. The `validateIdentifier` method is highlighted in green. The `valid_s` and `valid_f` methods are also visible. The Package Explorer on the left shows the project structure, and the Outline on the right shows the class and method structure. The bottom status bar includes "Problems", "Javadoc", "Declaration", "Console", "History", and "Coverage".



# Programa Exemplo (12)



# Programa Exemplo (13)

IdentifierTestCase (12/02/2014 13:29:20)

## IdentifierTestCase (12/02/2014 13:29:20)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Mis
Identifier		77%		79%	8	25	8	30	
Total	28 of 122	77%	8 of 38	79%	8	25	8	30	

IdentifierTestCase (12/02/2014 13:29:20)





## Objetivo

- ▶ O processo de instrumentação *off-line*, serve para que uma versão instrumentada da aplicação para ser executada fora do ambiente de desenvolvimento mas, mesmo assim, ser possível monitorar a cobertura de código.
- ▶ Sua principal utilidade é na condução de testes funcionais de sistemas, permitindo uma posterior análise da qualidade do teste funcional que foi executado.
- ▶ Além disso, é possível entregar uma versão instrumentada da aplicação para um usuário para descobrir qual o perfil de execução desse usuário particular.
- ▶ Essas informações são importantes para priorizar a execução de testes e também para avaliar a qualidade dos critérios de teste funcionais empregados.



## Instrumentação *off-line* com Maven e JaCoCo (1)

- ▶ JaCoCo é a API empregada pela EclEmma para o monitoramento de cobertura de código *on-the-fly*.
- ▶ A mesma API pode ser empregada para a geração de uma versão instrumentada *off-line* da aplicação em teste.
- ▶ A documentação de como usar a JaCoCo em modo *off-line* está disponível em: <http://www.eclemma.org/jacoco/trunk/doc/index.html>
- ▶ Para isso, o primeiro passo, é criar um novo arquivo de configuração do Maven (`pom-jacoco.xml`, por exemplo) com o conteúdo apresentado a seguir e também disponível em <http://www.eclemma.org/jacoco/trunk/doc/examples/build/pom-offline.xml>:

## Instrumentação *off-line* com Maven e JaCoCo (2)

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0_ http://maven.apache.org/xsd/mav
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>Identifier</groupId>
5   <artifactId>Identifier</artifactId>
6   <version>1.0</version>
7   <name>Identifier</name>
8   <description>Validador de identificador</description>
9   <build>
10     <plugins>
11       <plugin>
12         <groupId>org.apache.maven.plugins</groupId>
13         <artifactId>maven-compiler-plugin</artifactId>
14         <version>3.1</version>
15         <configuration>
16           <source>1.5</source>
17           <target>1.5</target>
18         </configuration>
19       </plugin>
20       <plugin>
21         <groupId>org.jacoco</groupId>
22         <artifactId>jacoco-maven-plugin</artifactId>
23         <version>0.6.4.201312101107</version>
24         <executions>
25           <execution>
26             <id>default-instrument</id>
27             <goals>
28               <goal>instrument</goal>
29             </goals>
30           </execution>

```

## Instrumentação *off-line* com Maven e JaCoCo (3)

```
31     <execution>
32         <id>default-restore-instrumented-classes</id>
33         <goals>
34             <goal>restore-instrumented-classes</goal>
35         </goals>
36     </execution>
37     <execution>
38         <id>default-report</id>
39         <phase>prepare-package</phase>
40         <goals>
41             <goal>report</goal>
42         </goals>
43     </execution>
44     <execution>
45         <id>default-check</id>
46         <goals>
47             <goal>check</goal>
48         </goals>
```

# Instrumentação *off-line* com Maven e JaCoCo (4)

```

49     <configuration>
50         <rules>
51             <!-- implementation is needed only for Maven 2 -->
52             <rule implementation="org.jacoco.maven.RuleConfiguration">
53                 <element>BUNDLE</element>
54                 <limits>
55                     <!-- implementation is needed only for Maven 2 -->
56                     <limit implementation="org.jacoco.report.check.Limit">
57                         <counter>COMPLEXITY</counter>
58                         <value>COVEREDRATIO</value>
59                         <minimum>0.60</minimum>
60                     </limit>
61                 </limits>
62             </rule>
63         </rules>
64     </configuration>
65 </execution>
66 </executions>
67 </plugin>
68 <plugin>
69     <groupId>org.apache.maven.plugins</groupId>
70     <artifactId>maven-surefire-plugin</artifactId>
71     <version>2.12.2</version>
72     <configuration>
73         <systemPropertyVariables>
74             <jacoco-agent.destfile>target/jacoco.exec</jacoco-agent.destfile>
75         </systemPropertyVariables>
76     </configuration>
77 </plugin>

```

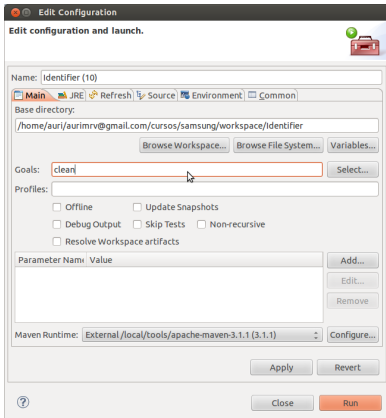




## Instrumentação *off-line* com Maven e JaCoCo (7)

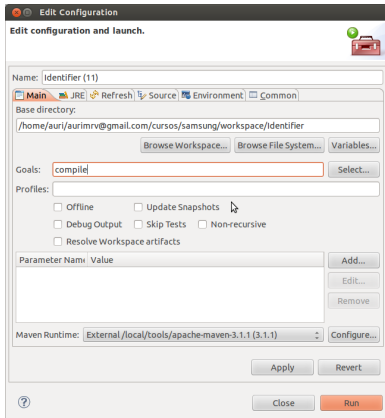
- ▶ Criado o novo arquivo de configuração, para gerar o arquivo jar com as classes instrumentadas para a execução *off-line* execute os seguintes comandos, na ordem estabelecida.
  1. mvn clean
  2. mvn compile
  3. mvn test-compile
  4. mvn -f pom-jacoco.xml jacoco:instrument
  5. mvn package -Dmaven.test.failure.ignore=true
- ▶ Ao final deste processo, será gerado um arquivo .jar dentro do diretório target e nele estão as classes instrumentadas da aplicação.
- ▶ A sequencia de telas a seguir ilustra todo o processo para a geração do jar instrumentado.

# Instrumentação off-line com Maven e JaCoCo (8)

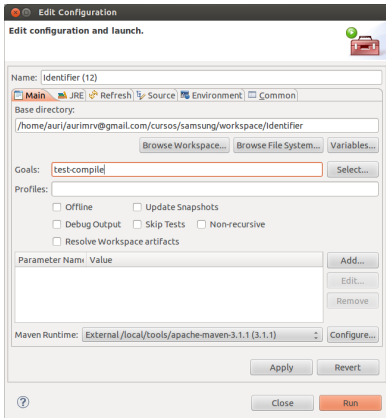




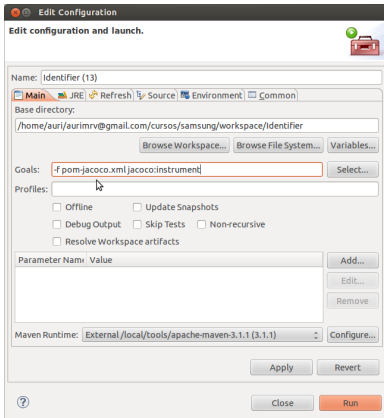
# Instrumentação off-line com Maven e JaCoCo (9)



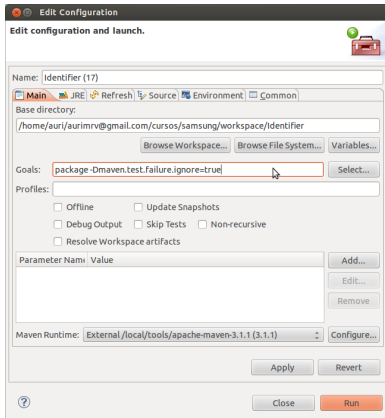
# Instrumentação *off-line* com Maven e JaCoCo (10)



# Instrumentação off-line com Maven e JaCoCo (11)



# Instrumentação off-line com Maven e JaCoCo (12)



## Executando a Aplicação Instrumentada (1)

- ▶ Como dentro do código instrumentado existem instruções que dependem da JaCoCo, para executá-la, é necessário incluir no CLASSPATH o caminho para o arquivo `org.jacoco.agent-0.6.4.201312101107-runtime.jar`, localizado dentro do repositório local do Maven `.m2/repository/org/jacoco/org.jacoco.agent/0.6.4.201312101107`.
- ▶ O comando abaixo ilustra como executar a aplicação assumindo que o jar com as classes instrumentadas e a o jar da JaCoCo estejam no mesmo diretório.

```
$ java -cp Identifier-1.0.jar:\  
> org.jacoco.agent-0.6.4.201312101107-runtime.jar \  
> IdentifierMain abcd1  
Valido
```





## Gerando o Relatório de Cobertura (1)

- ▶ Para gerar o relatório de cobertura dos testes, basta copiar o arquivo `jacoco.exec` para dentro da pasta `target` do projeto do Eclipse e, em seguida, executar os *Goals*:
  1. `mvn -f pom-jacoco.xml jacoco:restore-instrumented-classes`
  2. `mvn -f pom-jacoco.xml jacoco:report`
- ▶ Após a execução dos mesmos será criado o diretório `target/site` com o relatório de cobertura dos testes realizados.





## Ferramenta EclEmma

Discussão

Pre-requisitos

Instalação

A Ferramenta

O Exemplo

## Instrumentação Off-line

Instrumentação *off-line* com Maven e JaCoCo

Executando a Aplicação Instrumentada

## Gerando o Relatório de Cobertura

## Outras Ferramentas Similares

## Exercício

## Outras Ferramentas Similares

- ▶ EclEmma não é a única ferramenta que apoia a aplicação de critérios de teste baseados em fluxo de controle.
- ▶ Considerando as principais linguagens de programação, existem outras ferramentas similares, tais como:
  - ▶ Emma (<http://emma.sourceforge.net/>) ou EclEmma (<http://www.eclEmma.org/>) para Java.
  - ▶ CodeCover (<http://codecover.org/>) para Java.
  - ▶ Cobertura (<http://cobertura.github.io/cobertura/>)
  - ▶ TCAT (<http://www.soft.com/TestWorks>) para C/C++ e Java.
  - ▶ JavaCov (<http://www.alvicom.hu/>) para Java.
  - ▶ Dentre outras...
  - ▶ Uma lista extensa pode ser encontrada em <http://www.testingfaqs.org/> e <http://www.opensourcetesting.org/>.



## Ferramenta EclEmma

Discussão

Pre-requisitos

Instalação

A Ferramenta

O Exemplo

## Instrumentação Off-line

Instrumentação *off-line* com Maven e JaCoCo

Executando a Aplicação Instrumentada

## Gerando o Relatório de Cobertura

## Outras Ferramentas Similares

## Exercício

## Exercício

- ▶ Avaliar a cobertura dos casos de teste do programa `CommentPrinter`.
- ▶ Utilize a ferramenta EclEmma para gerar casos de testes de modo que 100% dos comandos e decisões de todo código do programa seja obtido.
- ▶ Lembre-se que os casos de testes podem ser feitos no JUnit.
- ▶ Entregar os casos de testes desenvolvidos que garantam a cobertura máxima dos critérios apoiados pela ferramenta.
- ▶ Qual a complexidade ciclomática média das classes que compõem o programa `CommentPrinter`?

