

CodeCover

Michel Meyer und Manuel Schwarz

22. Januar 2013

Inhalt

- 1 Einleitung
- 2 Funktionsweise
 - Tests
 - Technische Integration
- 3 CodeCover allgemein
 - Einsatzzweck
 - Allgemeine Informationen
 - Installation (Eclipse)
- 4 CodeCover Demo
 - Kommandozeile
 - Eclipse
- 5 Fazit



Motivation

- Softwarequalität erhöhen/verbessern
- Fehler schneller finden
- Code-Refactoring vereinfachen
- einfach zu bedienendes Tool
- evtl. IDE-Einbettung



Testarten

CodeCover deckt folgende Tests ab:

- Bedingungsüberdeckung
- Zweigüberdeckung
- Schleifenüberdeckung
- Anweisungsüberdeckung
- Ternärer Operator Überdeckung
- Synchronisationsüberdeckung

Benutzung

- Kommandozeile (Report erstellen)
- Eclipse (verschiedene Views + Report)
- Testfälle einfach erstellen und zusammenfassen
- Nutzen mit Ant
- Unterstützung von JUnit



White- bzw. Glass-Box Testing

- dynamisches Testverfahren
- strukturorientiert
- genauer: kontrollflussorientiert
- Prüfung direkt am Code im Gegensatz zum Blackbox Testing
- Ziel: Möglichst hohe Überdeckung



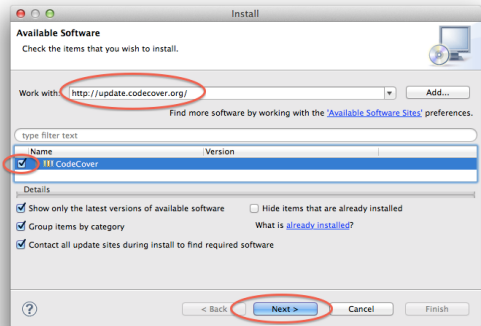
Download und Toolinfos

- Quelle: kostenlos unter codecover.org
- letzte Version (Stand: März 2011): CodeCover 1.0.1.2 (knapp 3 MB)
- Lizenz: Eclipse Public Licence (EPL)
- Plattformen: Kommandozeile (Linux, Windows, Mac OS) sowie Eclipse- und Ant-Integration
- Programmiersprachen: Java und COBOL

Installation (Eclipse)

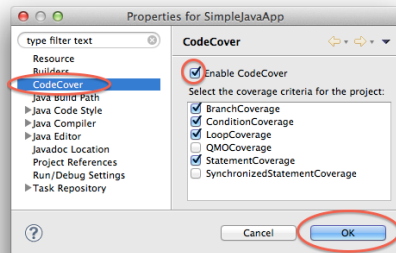
Installation

- Eclipse starten
- “Help” →
“Install New
Software...”
- URL:
`http://update.codecover.org/`
eingeben
- CodeCover
auswählen



CodeCover aktivieren

- Rechtsklick auf gewünschtes Projekt
- CodeCover auswählen und aktivieren
- die gewünschten Kriterien auswählen

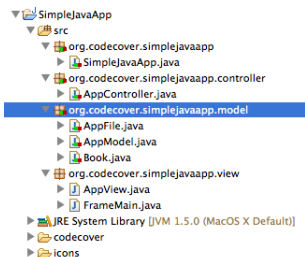




Installation (Eclipse)

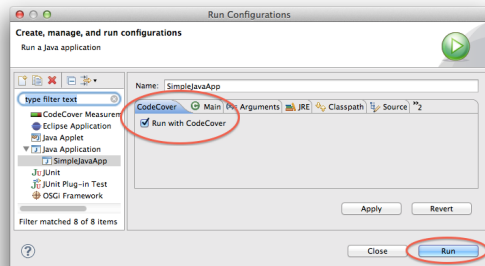
Zu prüfende Klassen auswählen

- zu testende Klassen auswählen
- Rechtsklick → “Use For Coverage Measurement”



Ausführen

- Run → Run Configurations...
- Run with CodeCover auswählen
- Ausführen






Installation (Eclipse)

Farbkodierung

- grün: komplette Abdeckung
- gelb: Teilabdeckung
- rot: wird nicht evaluiert

```
AppModel.java  FrameMain.java  HelloHighlighting.java  Main.java  
1  package main;
2
3  public class Main
4  {
5      public static void main(String[] args)
6      {
7          boolean a, b, c, d;
8
9          a = true;
10         b = true;
11         c = false;
12         d = false;
13
14         if (a || b)
15         {
16             System.out.println("a == true");
17         }
18
19         while ((a && b) || (c && d))
20         {
21             System.out.println("(a && b) || (c && d)");
22             a = false;
23         }
24
25         for (int t = 0; t < 100; t++)
26         {
27             if (t > 10)
28             {
29                 System.out.println("t > 10");
30             }
31
32             if (t == 100)
33             {
34                 System.out.println("t == 100");
35             }
36         }
37     }
38
39 }
```



einfaches Beispiel

DEMO



Instrument

```
Manuel@manschwa ~/Desktop/ws
% sudo ./codecover.sh instrument --root-directory TestProject/src --destination TestPro
ject/instrumentedSrc --container TestProject/test-session-container.xml --language java -
-charset UTF-8
|=====| [100.0%]
```

Abbildung: Source Code instrumentieren, d.h. zur Analyse vorbereiten.



Compile

```
Manuel@manschwa ~/Desktop/ws
% sudo javac -cp "TestProject/instrumentedSrc" TestProject/instrumentedSrc/main/Main.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Abbildung: Aufbereitete Dateien kompilieren...



Run

```
Manuel@manschwa ~/Desktop/ws/TestProject/instrumentedSrc
% sudo java main.Main                                     !4984
a == true
(a && b) || (c && d)
t > 10
t > 10
t > 10
```

Abbildung: ... und ausführen. Dabei wird eine clf-Datei erstellt, die die Ergebnisse beinhaltet und einen Testfall darstellt.



Analyse

```
Manuel@manschwa ~/Desktop/ws  
% sudo ./codecover.sh analyze --container TestProject/test-session-container.xml --coverage-log "TestProject/instrumentedSrc/coverage-log-2013-01-07-04-39-56-817.clf" --name TestSession1 --comment "The first test session"
```

Abbildung: Die Ergebnisse werden analysiert und in eine xml-Datei eingefügt. Dabei kann ein *merge* mehrerer clf-Dateien, also Testfälle, erfolgen.

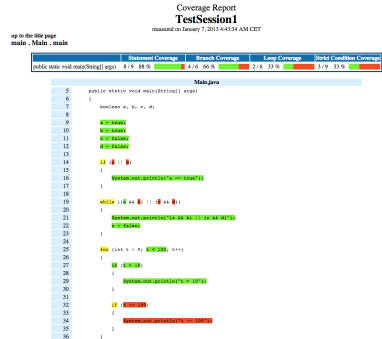
Report

```
Manuel@manschwa ~/Desktop/ws
% sudo ./codecover.sh report --container TestProject/test-session-container.xml --destination TestProject/report/TestProjectReport.html --session "TestSession1" --template report-templates/HTML_Report_hierarchic.xml
|=====| [100.0%]
```

Abbildung: Schließlich wird ein Report auf Basis aller Ergebnisse in Form von html-Dateien erzeugt.

Report

Darstellung der HTML-Datei
mit den
Code Coverage-Ergebnissen
der Testfälle.





komplexes Beispiel

DEMO

SimpleJavaApp



Live Notification (1)

Live Notification wird genutzt um verschiedene Testfälle zu generieren.

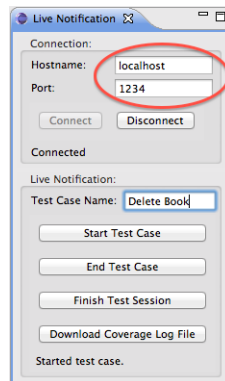
Notwendige VM Argumente:

VM Argumente

- Dcom.sun.management.jmxremote
- Dcom.sun.management.jmxremote.port=1234
- Dcom.sun.management.jmxremote.ssl=false
- Dcom.sun.management.jmxremote.authenticate=false

Live Notification (2)

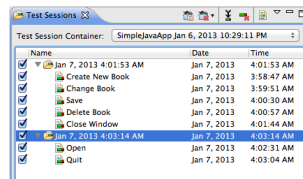
- “Live Notification”-View wählen
- “localhost” als hostname und “1234” als port eintragen
- Applikation starten
- “connect” klicken
- Testfälle generieren





Testfälle

- Erzeugte Testfälle werden gespeichert
- Auswahl treffen
- View wählen und anzeigen



Coverage View

- Zeigt den Grad der jeweiligen Überdeckung an
- hier: Anweisungsüberdeckung

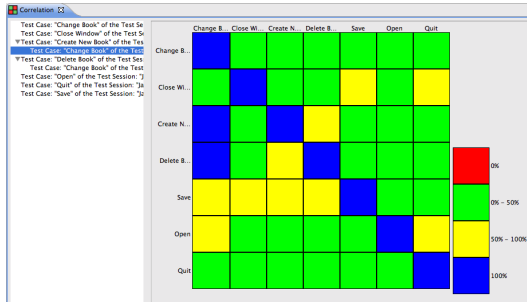
Coverage

☐ Show methods with Statement Coverage <= 90.5 %

| Name | Statement | Branch | Loop | Term |
|----------------|-----------|--------|--------|--------|
| SimpleJavaApp | 57.5 % | 45.7 % | 44.4 % | 45.5 % |
| org | 57.5 % | 45.7 % | 44.4 % | 45.5 % |
| codecover | 57.5 % | 45.7 % | 44.4 % | 45.5 % |
| simplejavaapp | 57.5 % | 45.7 % | 44.4 % | 45.5 % |
| SimpleJavaApp | 0.0 % | 0.0 % | - | 0.0 % |
| controller | 51.5 % | 40.8 % | 33.3 % | 50.0 % |
| model | 76.5 % | 44.8 % | 53.3 % | 43.8 % |
| AppFile | 96.7 % | 66.7 % | 53.3 % | 75.0 % |
| AppModel | 57.1 % | 40.9 % | - | 37.5 % |
| AppModel | 0.0 % | - | - | - |
| AppModelModif | - | - | - | - |
| addAppModelM | 66.7 % | 50.0 % | - | 50.0 % |
| addBookToFile | 66.7 % | 50.0 % | - | 50.0 % |
| closeFile | 66.7 % | 50.0 % | - | 25.0 % |
| getAppFile | 50.0 % | 50.0 % | - | 50.0 % |
| getBooksInFile | 0.0 % | 0.0 % | - | 0.0 % |
| getBooksInFile | 100.0 % | - | - | - |
| getInstance | 0.0 % | 0.0 % | - | 0.0 % |
| getPathOfFile | 100.0 % | - | - | - |
| isFileModified | 100.0 % | - | - | - |
| loadFile | 75.0 % | 50.0 % | - | 50.0 % |
| newFile | 0.0 % | - | - | - |
| putAppFile | 100.0 % | - | - | - |
| removeAppMod | 66.7 % | 50.0 % | - | 50.0 % |
| removeBookFro | 66.7 % | 50.0 % | - | 50.0 % |
| saveFile | 66.7 % | 50.0 % | - | 50.0 % |
| Book | 67.4 % | 50.0 % | - | 50.0 % |
| view | 42.9 % | 60.0 % | 33.3 % | 50.0 % |

Correlation View

- Zeigt an, wie stark unterschiedliche Testfälle miteinander korrelieren





Zusammenfassung und Fazit

- gute Eclipse-Integration
- einfaches Generieren und Zusammenfassen von Testfällen
- verschiedene nützliche Views in Eclipse
- keine Weiterentwicklung seit fast 2 Jahren
- nur Java und COBOL werden unterstützt
- evtl. Alternativen suchen