Mutationstesten

Jens Kosiol

Sommersemester 2024

Wann ist eine Testsuite (von Unittests) gut?

Ziele von (Unit-)Tests

Unmittelbare Ziele:

- Existierende Fehler im Code aufdecken
- Regressionstesten ermöglichen (neu eingefügte Fehler aufdecken)

Wünschenswerte Eigenschaften:

- Schnelle Laufzeit
- Wartbarkeit
- Modularität
- …

Adäquatheitskriterien für Tests

Adäquatheitskriterien für Tests sollen dabei helfen, gute Testsuites zu entwickeln:

- Vorgehensmodell für den Entwurf neuer Testsuites
- Messbar machen der Qualität vorhandener Testsuites (Qualitätsmetriken)
- Aufdecken fehlender Testfälle in einer vorhandenen Testsuite
- (Aufdecken unnötiger Testfälle)

Bekannte Adäquatheitskriterien

- Code-Coveragekriterien, z. B. Statement-, Branch-, Path-Coverage
  - Leicht verständlich
  - Statement- und Branch-Coverage sind gut messbar
  - Gut adressierbar
  - Werden industriell eingesetzt
- Data-Coverage
  - Deckt die möglichen Nutzungsvarianten des Codes systematisch ab
  - Prinzipiell auch als Greybox-Verfahren möglich (Kenntnis der Signaturen)
  - Schwieriger umzusetzen und zu messen

Solche Coverage-Kriterien messen, dass Code ausgeführt wurde, nicht, dass Code getestet wurde!!!

Grundidee Mutationstesten

- Motivation: Qualität einer Testsuite überprüfen, indem man prüft, ob sie in der Lage ist, Fehler im Code zu finden
- Grundsätzliches Vorgehen: Ändere den Ausgangscode auf zufällige Art und Weise ab (Mutation) und überprüfe, ob die vorhandene Testsuite in der Lage ist, diese Änderung zu erkennen.

Beispiel

```java
public class Adder {

  public int add(int a, int b) {
    return a+b;
  }
}
```

Originalklasse

```java
public class Adder {

  public int add(int a, int b) {
    return a*b;
  }
}
```

Mutant

```java
class AdderTest {
  private Adder adder = new Adder();

  @Test
  void addingZeroIsLeftNeutral() {
    assertEquals(5, adder.add(0, 5));
  }
}
```

Test

Ablauf Mutationstesten (naiv)

1. Von dem Originalcode werden Mutanten erstellt
2. Die Testsuite wird gegen jeden der Mutanten laufen gelassen
   – Ein Mutant, der von einem Test entdeckt wird, heißt tot (dead), die anderen heißen lebendig (live)

Berechnung Mutation Score:

$$\frac{\#\ \text{getöteter Mutanten}}{\#\ \text{generierte Mutanten}}$$

Art der Mutanten

Competent Programmer Hypothesis: Ein Programmierer schreibt normalerweise Programme, die syntaktisch nahe an einem korrekten Programm liegen.

Coupling Effect: Eine Testsuite, die minimale Änderungen am Programm erkennen kann, ist so gut, dass sie auch komplexere Fehler erkennt.

Daher: Mutationstesten wird häufig mit sogenannten first-order Mutanten durchgeführt (eine Mutation am Ausgangscode).

Mutationsoperatoren

- Ein Mutationsoperator führt eine kleine Veränderung am Code durch wie z. B. den Austausch eines binären Operators gegen einen anderen.
- Mutationsoperatoren werden spezifisch für eine Programmiersprache entworfen.
- PIT (https://pitest.org/) ist das bekannteste Framework für Mutationstesten in Java.
  - Überblick über die Mutationsoperatoren von PIT: https://pitest.org/quickstart/mutators/

Einsatzzwecke Mutationstesten

- Mutationsanalyse (mutation analysis): Messen des mutation scores einer Testsuite, um deren Qualität zu evaluieren
  - Eine hohe Anzahl möglichst diverser und relevanter Mutanten muss entworfen und die Testsuite dagegen laufen gelassen werden (in der Hoffnung auf einen hohen mutation score).
- Mutationstesten (mutation testing): Aufdecken von Lücken in einer Testsuite durch Mutanten
  - Hier ist es von Vorteil, Mutanten zu generieren, die eine möglichst hohe Chance haben zu überleben (ohne zum Ausgangsprogramm equivalent zu sein).

Töten von Mutanten

Was ist nötig, damit ein Mutant von einer Testsuite getötet wird?

RIP:

- Reach: Ein Test muss den mutierten Code erreichen.
- Infect: Die gewählten Testdaten müssen dazu führen, dass in Ausgangscode und Mutant unterschiedliche Zustände vorliegen.
- Propagate: Der falsche Zustand muss sich in den Output fortsetzen und vom Test überprüft werden.

Schwierigkeiten von Mutationstesten

Mutation score kann ein zuverlässigerer Indikator dafür sein, dass eine Testsuite in der Lage ist, Fehler aufzudecken als Coverage-Kriterien (siehe z. B. [CPTH17] für den Fall von Systemtests), ist aber praktisch schwer einzusetzen:

- Unheimlich zeitaufwändig:
  - Mutanten müssen generiert und kompiliert werden
  - Jeder Mutant muss gegen die Testsuite getestet werden
- Wird ein Mutant von der Testsuite nicht als solcher erkannt, muss überprüft werden, ob er äquivalent zum Ausgangsprogramm ist (unentscheidbares Problem!).
- Aufgrund vieler möglicher elementarer Mutationen entsteht für Programme realistischer Größe ein riesiger Raum möglicher Mutanten (wovon von einer halbwegs gut ausgebauten Testsuite die meisten getötet werden).

Beispiel für äquivalenten Mutanten

```
if (x > 0) {
    foo(x);
}
```

Originalklasse

```
if (x > 0) {
    foo(abs(x));
}
```

Mutant

Mutation score adaptiert

Berechnung Mutation Score:

$$\frac{\#\ \text{getöteter Mutanten}}{\#\ \text{generierte Mutanten} - \#\text{äquivalente Mutanten}}$$

Optimierungen von Mutationstesten

- Mutation von Bytecode statt Sourcecode
- Verwendung von Mutationsschemata statt einzelner Mutationen
- Konzentration auf Mutationen, die mit hoher Wahrscheinlichkeit zu nicht-äquivalentem Code führen
- Konzentration auf Mutationen, die mit höherer Wahrscheinlichkeit zu Mutanten führen, die überleben
- Analyse, auf welche Mutanten überhaupt Tests angewendet werden müssen
- Analysen, welche Tests überhaupt für welchen Mutanten ausgeführt werden müssen
- …

Ausblick Aufgabe

- Ist es mit LLMs vielleicht möglich, „realistischere“ Mutanten zu generieren, also solcher, die eher einem typischen Bug eines Entwicklers entsprechen?
- Ist es über LLMs möglich, einfacher Mutanten zu generieren, die
  - nicht äquivalent zum Ausgangsprogramm sind und
  - nicht von den vorhandenen Tests getötet werden?
- Ist es mit LLMs möglich, aus einem nicht getöteten Mutanten einen Test zu gewinnen, der diesen Mutanten tötet?

Literatur

- [Fraser10] Gordon Fraser: Folien „Mutation Testing“, Vorlesungsfolien 2010 (https://www.st.cs.uni-saarland.de/edu/testingdebugging10/slides/10-MutationTesting.pdf)
- [CPTH17] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, Mark Harman: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. ICSE 2017: 597–608
- [BWB+21] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, Erik Meijer: What It Would Take to Use Mutation Testing in Industry – A Study at Facebook. ICSE (SEIP) 2021: 268–277