HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Regressionstests

Im Kontext von Projekt- und Qualitätsmanagement

Marc Herschel, 05.12.2020

Inhaltsverzeichnis

Definition: Regressionstest

„Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.“

IEEE Standard Glossary of Software Engineering Terminology, S. 61

„Regression testing is the rerunning of test cases that a program has previously executed correctly and comparing the current result to the previous result in order to detect unintended effects of a software change.“

General Principles of Software Validation (FDA), S. 25

Regressionen und Regressionstests

Ziel: Sicherstellen, dass Modifikationen innerhalb eines Softwaresystems keine neuen Fehler, sprich „Regressionen“ als unerwünschten Nebeneffekt verursachen.

Modifikationen sind in diesem Falle z.B.

- eine Erweiterung/Änderung der Funktionalität des Softwaresystems.
- das Anwenden von Refaktorisierungs-Maßnahmen.
- das Aktualisieren / die Neueinführung von externen Abhängigkeiten wie z.B. Libraries oder Frameworks.

Regressionen und Regressionstests

Notwendigkeit: Erlaubt es somit also eine Modifikation des Softwaresystems durchzuführen und gleichzeitig gewissenhaft die korrekte Funktionalität zu bewahren.

Einmalige Ausführung der Regressionstests nicht ausreichend!
→ Durchführung konstant nach jeder Änderung notwendig!

Problem: Ausführung aller Testfälle in komplexen Systemen sehr aufwändig! → Teilmenge der Testfälle ausführen.

- z.B. nur Testfälle, die mit der Änderung assoziiert sein könnten.
  Problem: Diese Teilmengen muss auch bestimmt werden.

Eigenschaften von Regressionstests

Gehören zu den dynamischen Softwaretests (Ausführung zur Laufzeit)

Gehören zu den Black-Box-Tests (Keine Sicht auf Quellcode)

Es handelt sich **nicht** um fachliche Tests → Ziel ist fachliche Konsistenz nach Änderungen (Sprich, alles funktioniert noch wie zuvor und Änderungen an einer Kompilationseinheit wirken sich nicht auf andere aus)

Hoher Wiederholungsfaktor der Tests → Ausführung nach jeder Änderung, die an dem System *„committed”* werden soll notwendig!

→ **daher Testautomatisierung bzgl. Regressionstests anstreben**

Abstrakte Funktionsweise

Operiert mit hilfe eines Soll-Ist-Verhaltensabgleich!

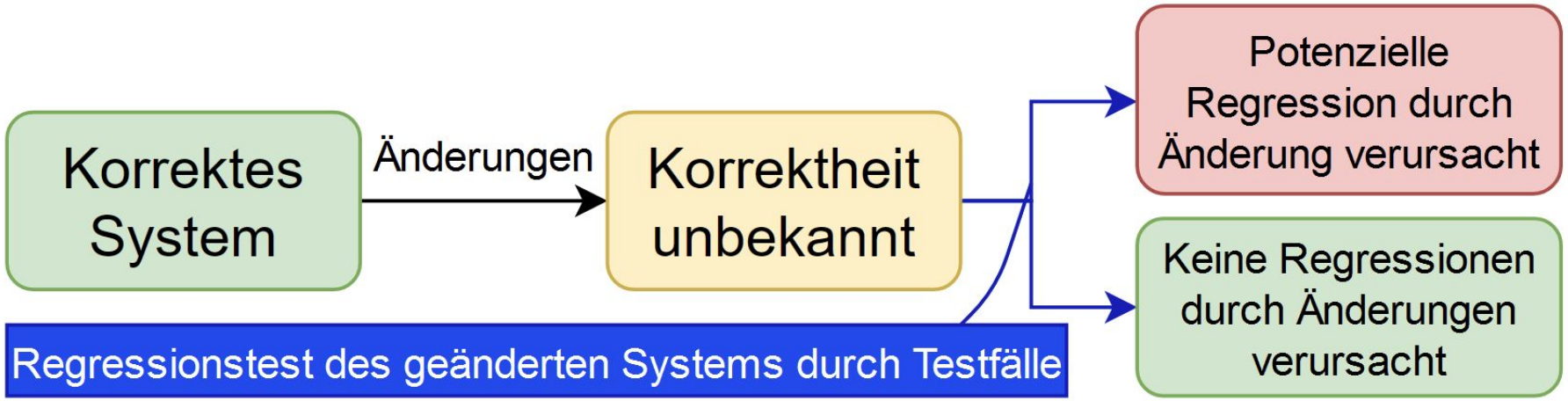
- Ausgaben der letzten stabilen Version der Software unter Ausführung der Testfälle sind als **Soll-Verhalten** definiert.
- Ausgaben der modifizierten Version der Software unter Ausführung der Testfälle ist als **Ist-Verhalten** definiert.

Unterschiede dieses Verhaltensabgleichs gelten vorerst erstmal als Regressionen und sorgen durch Inkonsistenz für einen Alarm!

Darauffolgend: Klassifizierung ob wirklich ein Fehler oder erwünschtes Verhalten durch geänderte Anforderung durch Stakeholder.

Abstrakte Funktionsweise

Anwendung von Regressionstests nach Änderungen des Softwaresystems:



Quelle: Eigene Illustration

Im abstrakten Sinne also ein stetiger Soll-Ist-Vergleich des Laufzeitverhaltens der getesteten Software mit der letzten stabilen Version unter verschiedenen Eingaben nach Änderungen durch Testfälle!

- Keine Regressionen verursacht → Übernehme Änderungen im System
- Regression verursacht → Ursachenuntersuchung / Fehlerbehebung / Erneutes Laufen der Regressionstests / Änderung der Testfälle

Automatisierte Testsysteme

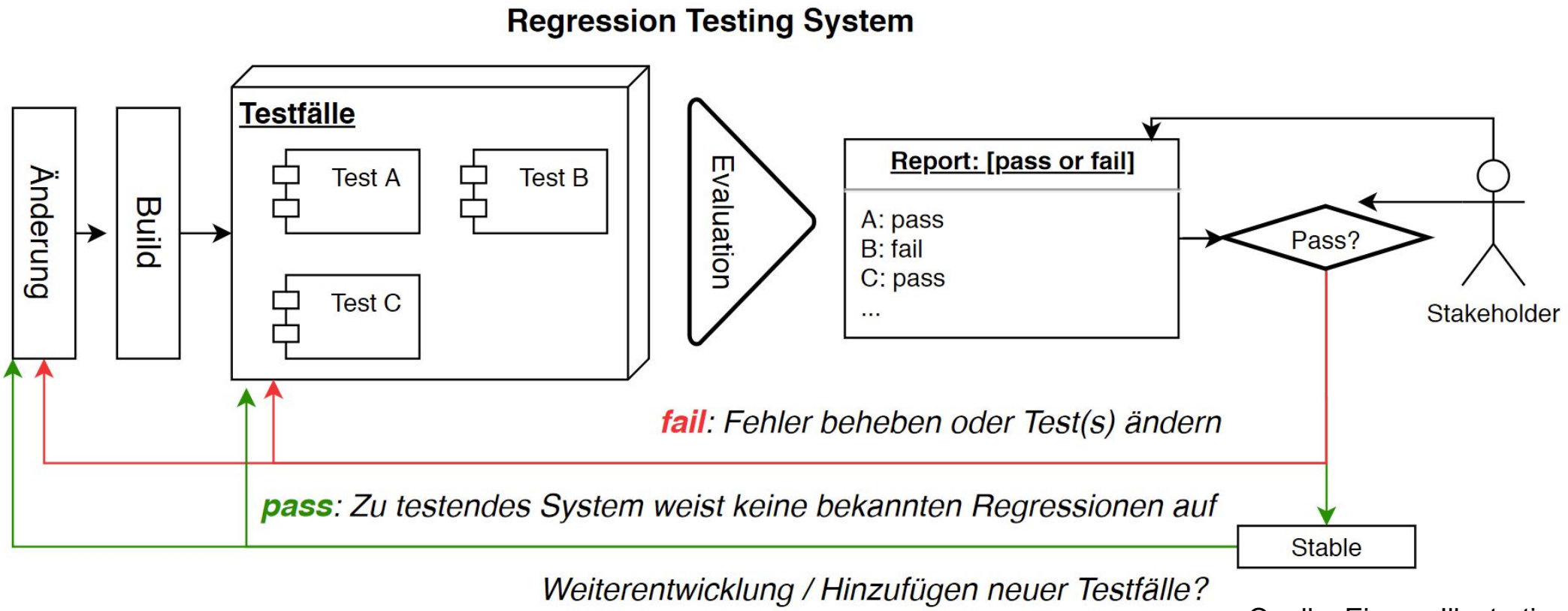
Manuelle Durchführung ist unwirtschaftlich, da Wiederholungscharakter!

→ **Daher**: Verwendung/Entwicklung passender Werkzeuge.

Aufgabe eines solchen Werkzeugs:

- Testeingaben/Testergebnisse für Testfälle aufzeichnen (Soll-/Ist-Verhalten bzw. Referenztestfälle genannt)
- Aufzeichnung nach Modifikation der Software wiederholen
- Anschließender Vergleich der Ergebnisse von Version **N** mit den der Version **N+1** *(Differenzen = mögl. Regressionen)*
- Manueller Eingriff nur bei Abweichungen im Testbericht!

Automatisierte Testsysteme



Quelle: Eigene Illustration

Beispielhafter Ablauf eines automatisierten Testsystems für Regressionstests bei der Weiterentwicklung von Software.

Testberichte

Notwendig um Aufmerksamkeit auf potenzielle Regressionen durch fehlgeschlagene Tests zu ziehen

Falls Regressionstests fehlschlagen, kann das verantwortliche Team diesen Testbericht sichten

Testberichte sollten detaillierte Informationen bzgl. d. Regressionen enthalten, um die Fehlersuche zu vereinfachen!

- Name des Testfalls
- Art des Problems (Unerwartetes Ergebnis, Absturz, Timeout, ...)
- Falls vorhanden: Lokalisierungsinformationen (Stacktraces, …)
- Anzahl aller Tests / Fehlgeschlagener Tests / Fehlerrate

Im Idealfall ist es so möglich eine Regression schnell zu orten!

Sichtung d. Testberichte: Am besten so schnell wie möglich!

Regressionstest-Techniken

Retest All

- Ausführen aller vorhandenen Testfälle
- Genaue Testabdeckung, allerdings aufwändig durchzuführen

Regression Test Selection

- Ausführung einer Untermenge aller vorhandenen Testfälle
- Bestimmung der Untermenge z.B. durch Deduplizierung / Messung der Testabdeckung und Anwendung auf Modifikation

Test Case Prioritization

- Optimierung: Testfälle, die oft Fehlschlagen vor Testfällen die selten Fehlschlagen ausführen → Fehlerquote erhöhen

Hybrid Approach

- Mischung von Regression Test Selection / Test Case Prioritization

Fallstudie: Regressionstests in Graja

http://graja.hs-hannover.de

- Autobewerter ***Graja*** stellt ein komplexes Softwaresystem dar
- Zustand vor der Arbeit: **~60k** Zeilen Java-Quellcode (Metrik: *SLOC*) und **31** Gradle-Untermodule
- Modultests mittels JUnit 4 für nur **3** dieser **31** Module
- Integrationstests bezüglich der Untermodule: **keine**

Problem: *Wie kann eine fehlerfreie Weiterentwicklung Grajas durch Softwaretests, in diesem Falle Regressionstests, garantiert werden?*

Ziel: *Konzept für einen Regressionstestmechanismus unter der Verwendung von Graja-Musterlösungen entwickeln und umsetzen.*

Regressionstests im Kontext von Graja

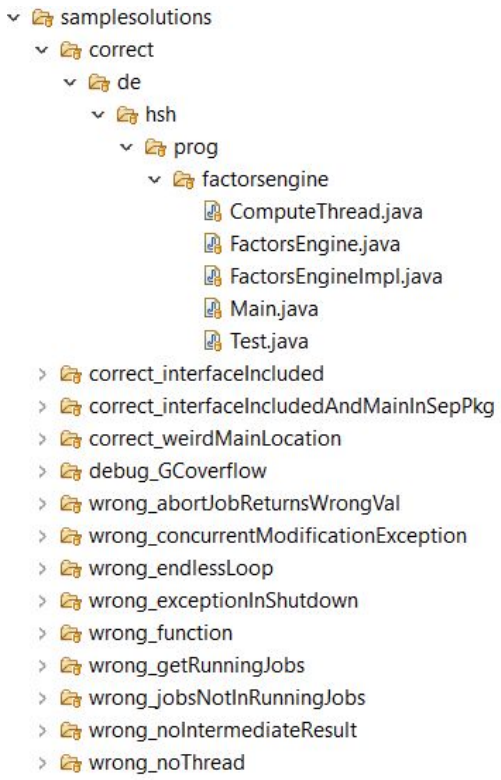
Im Falle von Graja: Das Bewertungsergebnis einer beliebigen Graja-Aufgabe sollte nach Änderungen innerhalb eines Graja-Untermoduls, bei gleichbleibenden Aufgaben-Verhalten reproduziert werden.

Ausnahme: Änderung soll Testfall explizit durch vorher unbekanntes Verhalten *„brechen“* → neuer Testfall, der Verhalten der Änderung abbildet notwendig.

Problem: *Woher für Graja-Regressionstests die Testfälle nehmen?*

Die bereits bestehenden Aufgaben in Kombination mit den bereits bestehenden Musterlösungen als Testfälle verwenden!

Realisierung der Testfälle in Graja



Quelle: Eigene Illustration

Verwendungszweck: z.B. manuelles Testen des *Grader-Codes* einer Aufgabe durch *Graja-GUI*.

Musterlösungen simulieren studentische Einreichungen und werden in Verbindung mit der jeweiligen Aufgabe durch Graja bewertet.

Können sowohl positiv als auch negativ ausfallen, als auch *„verbotene“* Operationen, die durch Graja abgefangen werden durchführen.

Für **57** Aufgaben existieren bereits **437** Musterlösungen!

Daher guter Kandidat für Testfälle! Große Vielfalt an Aufgaben und Musterlösungen können direkt übernommen werden und decken den gesamten Einsatzbereich ab!

Der Graja-Regressionstestmechanismus

Verhaltensaufzeichnung gegeben durch *Bewertungsschema*, *generierte Kommentarbäume* und *Bewertungsabbrüche*.

Verhaltensabgleich implementiert durch inhaltlichen/strukturellen Vergleich der *Kommentarbäume* und des *Bewertungsschemas*.

Ausgabe der Differenzen durch menschenlesbaren Testbericht.

Probleme:

- **Nichtdeterminismus** (z.B. durch Threading, Zufallsgeneratoren) → Mehrere Ausführungsstränge als gültig werten
- **Rauschen** (z.B. Zeitstempel, Dateipfade in Aufzeichnungen) → Rauschunterdrückung, „fuzzy compare“

Schnittstelle über das Build-Tool Gradle gegeben → Testautomatisierung durch z.B. CI/CD Pipelines und Webhooks möglich.

↑ - Location Context - 1 violation(s)

Context Classifier: GRADE_NODE - 1 violation(s)

Violation(s) occured in Level L3+ (Nodes of ProFormA-Grading Aspects and Combine-Groups)

Violation(s) occured within a node of the grading schema.

Path in grading schema:

```
root -> functionality -> junit@de.hsh.prog.bruch.grader.Grader#gettersShouldR
```

Attachments

No attachments exist.

↑ - Violation: EQUALITY_CLASH_GRADES

Both success ratings or grade sets do not match when being compared with each other.

Detected by: **Content Equality Test (CONTENT)**

Expected Grades:

```
CORRECT
```

Observed Grades:

```
WRONG
```

Expected Success:

```
0,000
```

Observed Success:

```
1,000
```

↑ - Violation: EQUALITY_CLASH_STRING

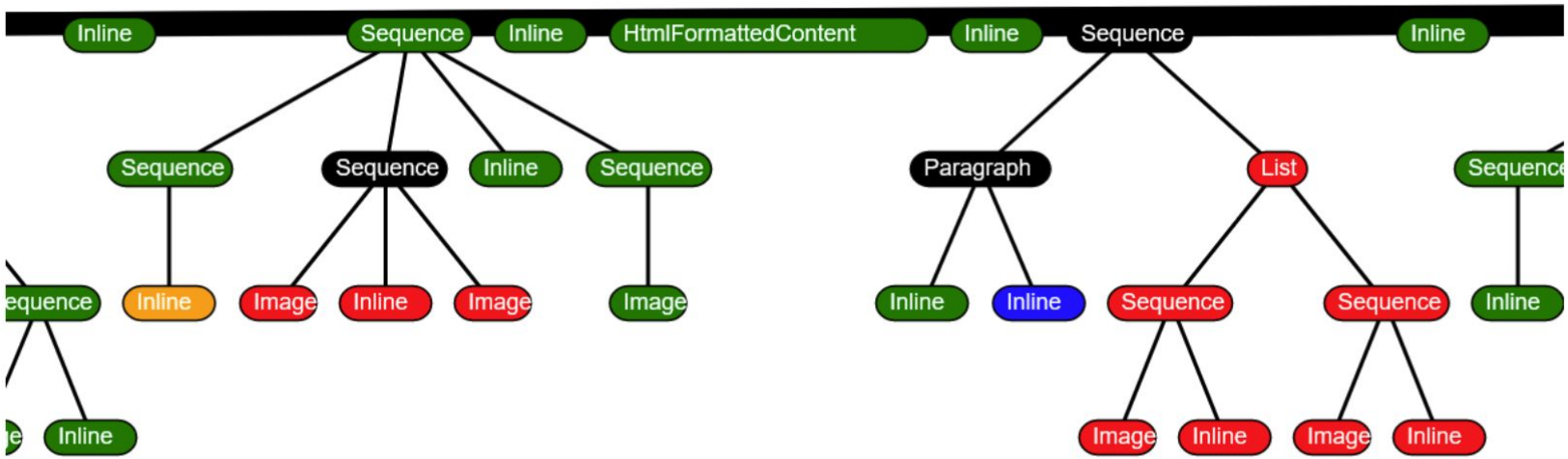
Both string values of a node do not match when being compared with each other.

Detected by: **Content Equality Test (CONTENT)**

```
=== Expected ===

Correct. Score: 2.00/2.00. Assignment de.hsh.prog.referenztypenswap.


=== Observed ===

Error. Score: 2.00/2.00. Assignment de.hsh.prog.referenztypenswap.
```

Origin of observed comment:

```
de.hsh.graja.util.comment.AbstractElementContent.<init>(AbstractElementContent.java:17)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:50)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:39)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:62)
de.hsh.graja.util.comment.Inline.format(Inline.java:124)
de.hsh.graja.util.comment.Paragraph.formatInlineItem(Paragraph.java:82)
de.hsh.graja.core.Core.gradeAssignment(Core.java:238)
de.hsh.graja.core.Core.gradeRequest(Core.java:104)
de.hsh.graja.regression.BehaviorRecordingCore.gradeRequest(BehaviorRecordingCore.java:41)
de.hsh.graja.core.apcli.BackendCommandProvider.gradeBackendRequest(BackendCommandProvider.java:84)
de.hsh.graja.core.apcli.BackendCommandProvider.executeCommand(BackendCommandProvider.java:50)
de.hsh.graja.util.cli.Command.execute(Command.java:342)
de.hsh.graja.util.cli.Main.main(Main.java:61)
de.hsh.graja.Cli.main(Cli.java:62)
```



Quelle: Eigene Illustration

Abschließende Bewertung von Regressionstests

Notwendigkeit, da viele Standards die Wiederholung von Tests fordern!

Verhindert **Effektiv** das Entstehen von ***Pattsituationen*** bezüglich der Funktionalität im Bezug zur Modifikation in der Entwicklung hochkomplexer Softwareprojekte!

Ohne Werkzeuge und Testautomatisierung nicht wirtschaftlich umsetzbar!

Schwierig bei Nichtdeterminismus und großer Menge an Testfällen (unwirtschaftlich, langsam → Reduzierung der Testfälle / selektive Auswahl notwendig).

Abschließende Bewertung von Regressionstests

Falls korrekt umgesetzt → sofortige Auswirkung bzgl. d. Qualität!

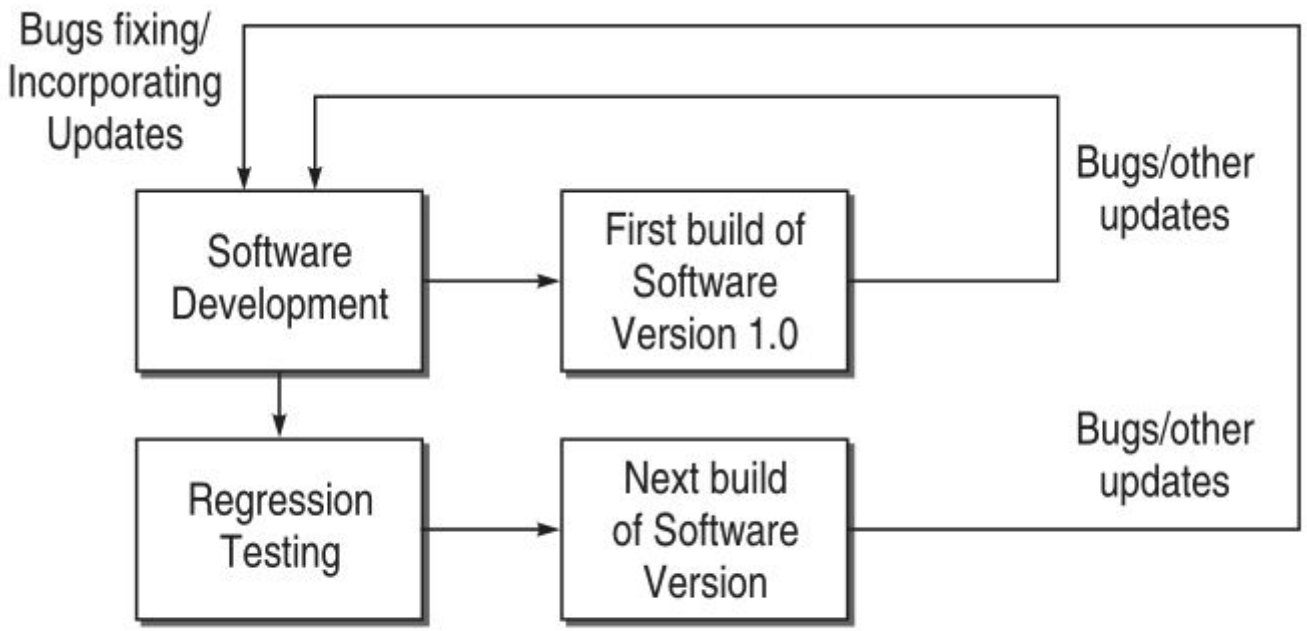


Figure 8.1 Regression testing produces Quality Software

Quelle: Software Testing - Principles and Practices, S. 257

1. Validiert geänderte Teile der Software
2. Validiert nicht verbundene Teile der Software, die mögl. durch Änderungen betroffen sind
3. Garantiert gleiche Funktion wie vor Änderungen
4. Reduziert das Risiko schwerwiegender Fehler

Quellen

5 Dinge, die Sie über Regressionstests wissen sollten - J. Rößler @ 18.11.2020

https://www.informatik-aktuell.de/entwicklung/methoden/5-dinge-die-sie-ueber-regressionstests-wissen-sollten.html

Regressionstest: Ihre Software-Versicherung - C. Johner @ 18.11.2020

https://www.johner-institut.de/blog/iec-62304-medizinische-software/regressionstest/

General Principles of Software Validation; Final Guidance for Industry and FDA Staff - HHS

IEEE Standard Glossary of Software Engineering Terminology - IEEE Foundation

Software-Qualität: Testen, Analysieren und Verifizieren - P. Liggesmeyer

Software Testing - Principles and Practices - N. Chauhan

Automated Defect Prevention - D. Huizinga; A. Kolawa

Understanding Regression Testing Techniques - G. Duggal, B. Suri

Musterlösungen in Graja als Mechanismus für Regressionstests - M. Herschel

Vielen Dank für Ihre Aufmerksamkeit!

Fragen können nun gestellt und beantwortet werden.