Fehlerfreie Software mit statischer Code-Analyse und dynamischen Tests

Um eine gute Softwarequalität zu gewährleisten, ist eine Kombination von statischer Analyse und hinreichendem Testen zur Laufzeit (dynamische Tests) in Verbindung mit einer Messung der Testabdeckung erforderlich. Unser Praxisbeispiel zeigt, warum beide Verfahren zur Absicherung der Softwarequalität genutzt werden müssen und warum die Konzentration auf nur eine Test- oder Analyseform gefährliche Konsequenzen haben kann.

Als ein Beispiel aus der Praxis, dass diese Aussage eindrucksvoll belegt, mag das Projekt eines Haushaltsgeräteherstellers dienen.

Für die Steuereinheit einer Waschmaschinenproduktreihe wurde eine Software entwickelt. Diese war in der Programmiersprache C geschrieben, mit dem Ziel, auf einem Microcontroller mit ARM-Technologie abzulaufen. Für alle Maschinen der Produktreihe sollte die identische Steuereinheit mit identischer Software Verwendung finden, wobei je nach Maschinentyp bestimme Funktionen ein- bzw. abgeschaltet werden sollten. Die Maschinen der etwas höheren Preisklasse sollten z.B. mit einem Sensor ausgestattet werden, um den Verschmutzungsgrad der Wäsche zu messen und dem Programm so zu ermöglichen, die notwendigen Zeiten des Hauptwaschgangs automatisch und bedarfsgerecht anzupassen. Die Waschzeiten der einfacheren Maschinen ohne Sensor sollten dagegen konstant sein.

Zur Sicherstellung einer hinreichenden Softwarequalität wurden Testfälle unter der Vorgabe einer MC/DC Testabdeckung von 100% erstellt. Auf eine statische Analyse, glaubte man, verzichten zu können.

Um die daraus später entstandenen Probleme zu beschreiben, ist ein Blick auf den Quellcode nötig. Der Originalcode kann jedoch aus rechtlichen Gründen nicht veröffentlicht werden und wäre zum einfachen Verständnis auch sicherlich zu komplex. Der in Abb. 1 wiedergegebene Code ist daher stark vereinfacht und auf die Problembeschreibung reduziert.

```
size_t durationMainWashCycle(size_t prog, size_t load, size_t staining) {
	if (((prog == 3) || (prog == 5) || (prog == 7)) && (load < 5)) {
		return staining * 5;
	}
	else if (((prog == 4) || (prog == 6)) && (load < 5)) {
		return staining * 8;
	}
	else if (((prog == 4) || (prog == 6)) && (load < 3)) {
		return staining * 7;
	}
	else {
		return staining * 9;
	}
}
```

***Abb. 1** Funktion zur Berechnung der Waschdauer des Hauptwaschgangs*

Die in Abb. 1 wiedergegebene Funktion errechnet die Waschdauer in Abhängigkeit vom gewählten Waschprogramm, der Beladungsmenge und dem Verschmutzungsgrad der Wäsche. Bezogen auf diese Funktion wurden während der Modultests die in untenstehender Tabelle aufgeführten Testfälle zum Ablauf gebracht, wobei der Verschmutzungsgrad „staining“ in das Testergebnis als Faktor einfließt. Inwieweit die Produktversion „product_version“ das Ergebnis beeinflusst, wird später erläutert.

Testfall Nr.	product_version	prog	load	staining	Ergebnis	erw. Ergebnis
1	11	3	4	3	15	15
2	11	5	4	3	15	15
3	11	7	4	3	15	15
4	11	3	6	3	27	27
5	12	4	2	1	**8**	**7**
6	12	6	2	1	**8**	**7**
7	13	4	4	1	8	8
8	13	6	4	1	8	8

Abb. 2 *Durchgeführte Testfälle bezogen auf die Funktion „`durationMainWashCycle()`“*

Das Ergebnis der Messung der Testabdeckung zeigt Abb. 3.

Hits/True	False	Line	Source
8		24	`size_t durationMainWashCycle(size_t prog, size_t load, size_t staining) {`
3	5	25	`  if (((prog == 3) || (prog == 5) || (prog == 7)) && (load < 5)) {`
1		25	`    1: ((T) || (_) || (_)) && (T)`
1		25	`    2: ((F) || (T) || (_)) && (T)`
1		25	`    3: ((F) || (F) || (T)) && (T)`
	1	25	`    4: ((T) || (_) || (_)) && (F)`
	0	25	`    5: ((F) || (T) || (_)) && (F)`
	0	25	`    6: ((F) || (F) || (T)) && (F)`
	4	25	`    7: ((F) || (F) || (F)) && (_)`
+		25	`    MC/DC (cond 1): 1 + 7`
+		25	`    MC/DC (cond 2): 2 + 7`
+		25	`    MC/DC (cond 3): 3 + 7`
+		25	`    MC/DC (cond 4): 1 + 4,  2 - 5,  3 - 6`
3		26	`    return staining * 5;`
		27	`  }`
4	1	28	`  else if (((prog == 4) || (prog == 6)) && (load < 5)) {`
2		28	`    1: ((T) || (_)) && (T)`
2		28	`    2: ((F) || (T)) && (T)`
	0	28	`    3: ((T) || (_)) && (F)`
	0	28	`    4: ((F) || (T)) && (F)`
	1	28	`    5: ((F) || (F)) && (_)`
+		28	`    MC/DC (cond 1): 1 + 5`
+		28	`    MC/DC (cond 2): 2 + 5`
-		28	`    MC/DC (cond 3): 1 - 3,  2 - 4`
4		29	`    return staining * 8;`
		30	`  }`
0	1	31	`  else if (((prog == 4) || (prog == 6)) && (load < 3)) {`
0		31	`    1: ((T) || (_)) && (T)`
0		31	`    2: ((F) || (T)) && (T)`
	0	31	`    3: ((T) || (_)) && (F)`
	0	31	`    4: ((F) || (T)) && (F)`
	1	31	`    5: ((F) || (F)) && (_)`
-		31	`    MC/DC (cond 1): 1 - 5`
-		31	`    MC/DC (cond 2): 2 - 5`
-		31	`    MC/DC (cond 3): 1 - 3,  2 - 4`
0		32	`    return staining * 7;`
		33	`  }`
		34	`  else {`
1		35	`    return staining * 9;`
		36	`  }`
		37	`}`

Abb.3 *Messung der Testabdeckung (MC/DC)*

Die Testabdeckung der betrachteten Funktion lag allerdings nur bei 71%.

Aus dem Report der Testabdeckung in Verbindung mit den Testergebnissen wurde sofort ein Problem sichtbar: Der Programmpfad mit der if-Bedingung mit Beginn in Zeile 31

```
else if (((prog == 4) || (prog == 6)) && (load < 3)) {
              return staining * 7;
```

wurde nicht durchlaufen, obwohl die entsprechenden Testfälle (Nr. 5 und 6) ausgeführt wurden. Das Testergebnis für diese Testfälle wich zudem von den erwarteten Ergebnissen ab. Aus dem Testabdeckungsbericht wird deutlich, dass stattdessen die if-Bedingung ab Zeile 28 durchlaufen wurde.
Ein Tausch der beiden else if-Bedingungen im Code behob diesen Fehler. Ein erneuter Testlauf ergab nunmehr eine Testabdeckung von 95% wobei die Testergebnisse den Erwartungen entsprachen.

Aus dem Report der Testabdeckung war ersichtlich, dass für eine Testabdeckung von 100% noch ein zusätzlicher Testfall benötigt wurde:

Testfall Nr.	product_version	prog	load	staining	Ergebnis	erw. Ergebnis
9	14	6	6	1	9	9

Ein weiteres Nachtesten unter Berücksichtigung des fehlenden Testfalls ergab dann die geforderte Testabdeckung von 100%.

Nach dem erfolgreichen Abschluss der Integrationstests unter Behebung einiger anderer kleinerer Probleme gingen die Waschmaschinen schließlich in Produktion und wurden ausgeliefert. Nach geraumer Zeit gab es Reklamationen von unzufriedenen Kunden. Bei einigen Maschinen war das Waschergebnis unzureichend, da der Hauptwaschgang bereits nach kurzer Zeit beendet war. Es wurden auch Fälle berichtet, bei denen die Maschinen die Hauptwäsche auf mehrere Stunden ausgedehnt hatten. Zuverlässig nachstellen ließ sich das Verhalten nicht.

Zunächst wurde ein Defekt des Verschmutzungssensors vermutet und ein Austausch im Rahmen der Gewährleistung vorgenommen. Es zeigte sich allerdings relativ schnell, dass damit das Problem nicht behoben war.

Nach genauerer Betrachtung der Reklamationsfälle fiel dann auf, dass diese auf einen bestimmten Maschinentyp der Produktreihe beschränkt waren. Man zog daraufhin auch die Möglichkeit eines Softwarefehlers in Betracht und beauftragte einen Dienstleister mit einer statischen Quellcodeanalyse.

```
size_t getStainingLevel(size_t product_version) {
       size_t y;
       if (product_version < 12) { //products w/o staining sensor
              y = 3;
       }
       else if (product_version > 12) {  //products with staining sensor
              y = readStainingSensor();
       }
       return y;
}
```

Abb. 4 *Ermittlung des Verschmutzungsgrads in Abhängigkeit von der Produktversion*

Die Vorgabe, dass alle Modelle ab dem Modell Nr. 12 den Verschmutzungssensor abfragen und alle Modelle darunter mit einem konstanten Verschmutzungswert arbeiten, wurde fehlerhaft umgesetzt. Für das Modell Nr. 12 bleibt die Variable „y“ in der Funktion „`getStainingLevel()`“ uninitialisiert und

übergibt damit einen unbestimmten Wert für den Verschmutzungsgrad. Da während der Modultests der Wert nicht auffällig war, blieb dieser schwerwiegende Fehler unentdeckt.

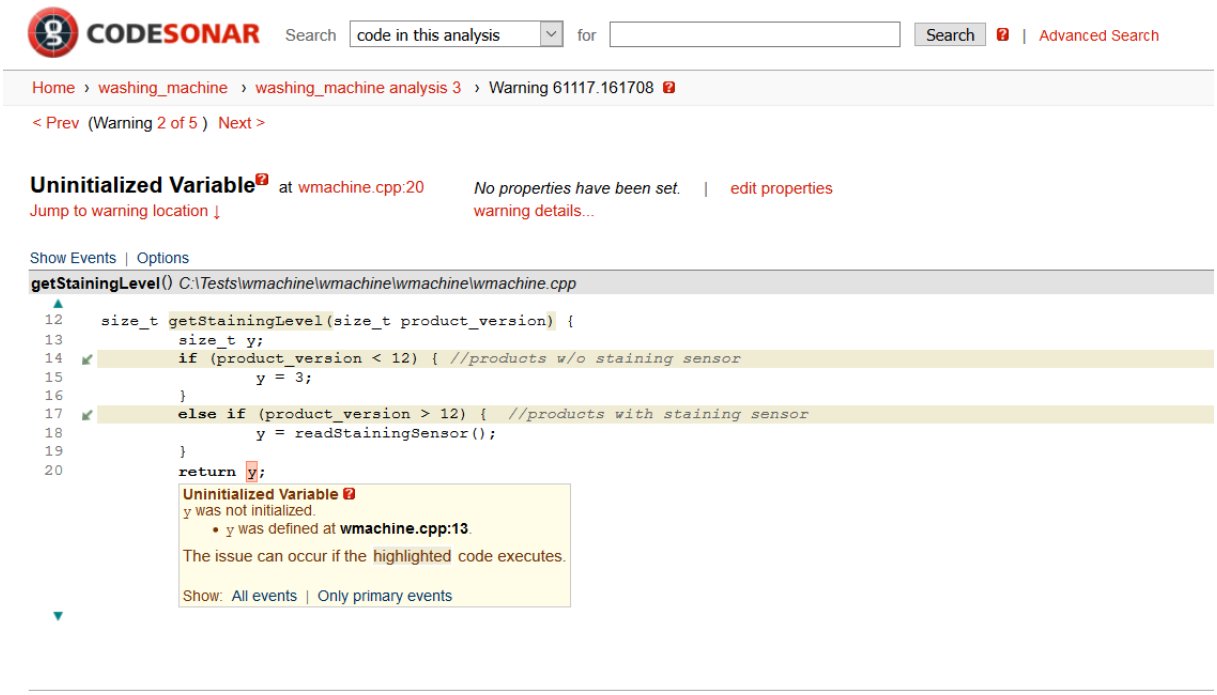

Abb. 5 *Nicht initialisierte Variable verursacht unbestimmtes Verhalten*

Das Ergebnis der statischen Quellcodeanalyse des hier zur Veranschaulichung vereinfachten Codes zeigt zudem (Abb. 6), dass der zuvor beschriebene Fehler des nicht erreichbaren Codeabschnitts mit großer Wahrscheinlichkeit bereits zur Implementierungszeit hätte frühzeitig aufgedeckt werden können.

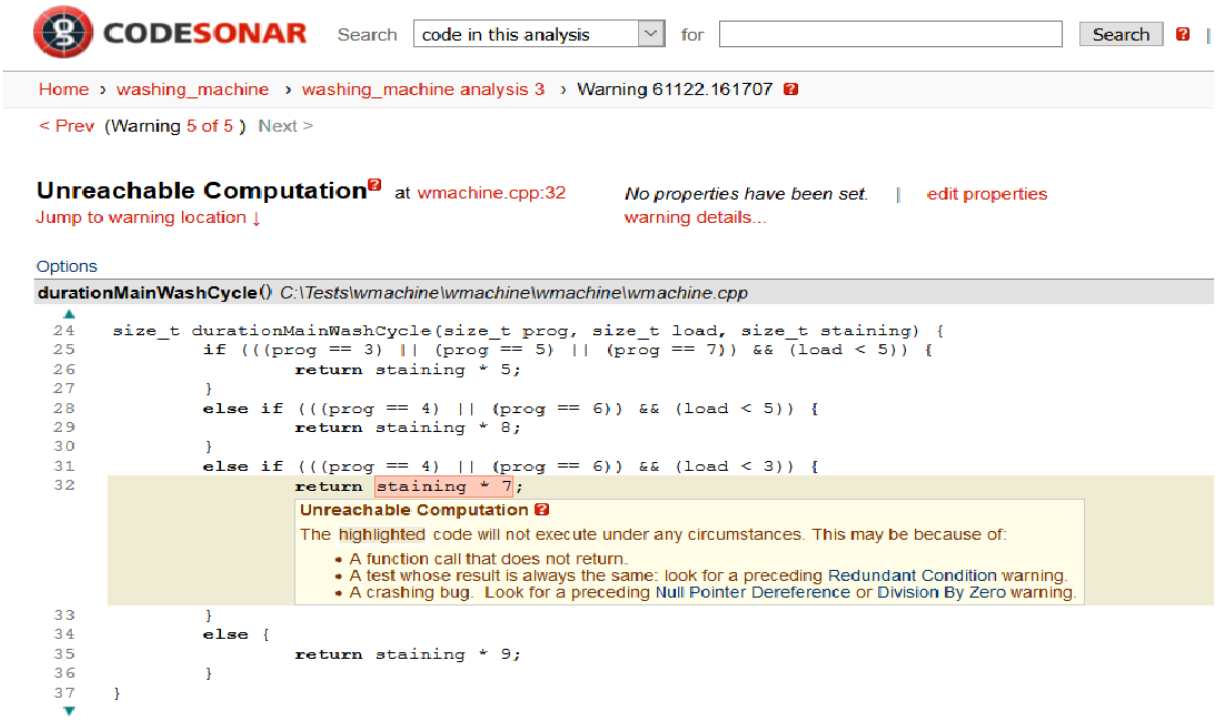

Abb. 6 *Nicht erreichbarer Code*

Die Fehler konnten nur durch die Kombination von statischer und dynamischer Analyse aufgedeckt werden. Leider wurde die statische Codeanalyse durch den Waschmaschinenhersteller erst nachträglich eingesetzt, statt bereits früh in der Entwicklungsphase, in der die Fehlerkorrektur noch relativ kostengünstig ist.

Durch rechtzeitige statische Analyse in Verbindung mit dynamischen Tests hätte eine kostspielige Rückrufaktion und die damit verbundenen Imageschäden vermieden werden können.

Der Elektrogerätehersteller setzt nunmehr sowohl die statische Analyse als auch das dynamische Testen bei Messung der Testabdeckung für alle seine Softwareprojekte ein.

Weitere Informationen:

Für die statische Codeanalyse wurde das Werkzeug CodeSonar[i] von GrammaTech eingesetzt. Die Messung der Testabdeckung (Code Coverage) erfolgte mit dem Tool Testwell CTC++[ii] von Verifysoft.

Die französische Übersetzung dieses Artikels wurde im Standardwerk „Pratique des Tests Logiciels“ von Jean-François Pradat-Peyre und Jacques Printz im Verlag Dunod[iii] veröffentlicht.

Autoren:

Royd Lüdtke ist Direktor für Statische Codeanalysetools bei der Verifysoft Technology GmbH

Roland Person ist Technischer Kundenberater bei der Verifysoft Technology GmbH und beschäftigt sich vor allem mit der dynamischen Codeanalyse und der Testabdeckung.

[i] https://www.verifysoft.com/de_grammatech_codesonar.html
[ii] https://www.verifysoft.com/de_ctcpp.html
[iii] ISBN 978-2-10-081995-9