QUALITÄTSMESSUNGEN IM FRONTEND

NIKLAS WILHELM
(BACHELORSTUDIUM WIRTSCHAFTSINFORMATIK)

Betreuer: Prof. Dr. Silke Lechner-Greite, Prof. Dr. Gerd Beneken

Motivation

Analysen der in der Softwareentwicklung anfallenden Kosten haben ergeben, dass der weitaus größte Anteil der Aufwendungen durch die Software-Wartung entsteht. Der Hauptgrund für die hohen Wartungskosten liegt in der schlechten Qualität der Software. In dem Software-Produkt enthaltene Fehler machen eine Anpassung oft überhaupt erst erforderlich. Auch andere Qualitätsprobleme wie komplexe Programmierkonstrukte, welche die Verständlichkeit erschweren, stellen Kostentreiber dar. Ein Großteil der Wartungskosten wird nämlich dadurch verursacht, dass die Entwickler den Quelltext erst verstehen müssen, bevor sie ihn anpassen können. Um die Qualität der Software zu verbessern und die Wartungskosten zu minimieren, sollten Werkzeuge zur Überprüfung des Software-Produktes eingesetzt werden. Sie können Probleme zu einem Zeitpunkt erkennen, zu dem die Korrektur noch mit geringen Kosten verbunden ist. Auf dem Markt befindet sich eine große Anzahl an Werkzeugen, die sich u.a. in ihrem Einsatzzweck und in ihren unterstützten Technologien unterscheiden. Das wird bei dem Vergleich der beiden Werkzeuge FindBugs und Dependency Cruiser deutlich. Während FindBugs der Erkennung von Fehlern in Java-Projekten dient, prüft Dependency Cruiser die Architektur von TypeScript-Projekten. Unternehmen müssen deshalb auf dem Markt befindliche Werkzeuge identifizieren und vergleichen. Nur dann ist es möglich, Werkzeuge auszuwählen, die sich gegenseitig ergänzen und die im Unternehmen genutzten Technologien unterstützen.

Zielsetzung und Methodik

Das Hauptziel dieser Bachelorarbeit ist der Vergleich und die Bewertung von Qualitätswerkzeugen, welche die Programmiersprache TypeScript und die Frameworks Angular, React und Vue.js unterstützen. Es wurden für die folgenden Qualitätsprobleme Werkzeuge gesammelt und untersucht:

- Code-Smells
- Bug-Patterns
- Architekturverletzungen
- Unzureichende Testabdeckung

Der Vergleich und die Bewertung der Werkzeuge erfolgte je nach Qualitätsproblem anhand unterschiedlicher Kriterien. Bei den ersten beiden Qualitätsproblemen, nämlich Code-Smells und Bug-Patterns, stand die Analyse von insgesamt 26 ausgewählten Problemen im Mittelpunkt. Hierbei wurde untersucht, ob und in welcher Weise die jeweiligen Werkzeuge die Probleme identifizieren können. Beim Qualitätsproblem der "Architekturverletzungen" lag der Fokus auf Architekturregeln, die von den Werkzeugen unterstützt werden. Die Analyse wurde hierbei anhand von sieben ausgewählten Architekturregeln vorgenommen. Bei der Untersuchung der unzureichenden Testabdeckung wurde hingegen auf einen breiten Vergleich verschiedener Werkzeuge verzichtet. Stattdessen lag das Augenmerk ausschließlich auf dem Werkzeug Istanbul. Die Analyse fokussierte darauf, ob das Werkzeug in der Lage ist, die korrekte Zweigüberdeckung für unterschiedliche Kontrollstrukturen zu berechnen.

Ergebnis

Für die Qualitätsprobleme „Code-Smells“ und „Bug-Patterns“ wurden die zwei Dashboard Toolkits SonarQube und Teamscale sowie die drei Sensoren SonarLint, ESLint und IntelliJ-Linter untersucht. Die Analyse von insgesamt 26 Code-Smells und Bug-Patterns führte zu den folgenden Resultaten:

- 21 Probleme erkannt ➔ SonarQube
- 20 Probleme erkannt ➔ SonarLint und IntelliJ-Linter
- 19 Probleme erkannt ➔ ESLint
- 15 Probleme erkannt ➔ Teamscale

SonarQube, SonarLint, IntelliJ-Linter und ESLint konnten eine ähnlich große Anzahl an Problemen erkennen, das Dashboard Toolkit Teamscale ist hingegen relativ weit abgeschlagen. Der Grund dafür ist, dass es im Vergleich zu den anderen Werkzeugen deutlich weniger Regeln für TypeScript zur Verfügung stellt. Andere im Frontend verwendete Sprachen wie HTML (Hypertext Markup Language) und CSS (Cascading Style Sheets) werden von Teamscale überhaupt nicht unterstützt. Das gilt auch für ESLint, da es sich auf JavaScript und TypeScript spezialisiert.

Für das Qualitätsproblem „Architekturverletzungen“ wurde die Produktgruppe Structure101g, das Dashboard Toolkit Teamscale und die zwei Sensoren Dependency Cruiser und TSArch untersucht. Die Abbildung 1 stellt die Analyseergebnisse der sieben ausgewählten Architekturregeln dar.

ROSENHEIMER INFORMATIKPREIS WIF-BACHELOR

Architekturregeln / Werkzeuge	Structure101g	Teamscale	Dependency Cruiser	TSArch
Einhaltung der Schichtenarchitektur	✓	✓	✓	✓
Verbot zyklischer Abhängigkeiten	✓	×	✓	✓
Pflichtimport eines Moduls	×	×	✓	×
Importverbot von Modulen mit bestimmten Dateinamen	✓	✓	✓	✓
Vorgabe für Dateinamen	×	×	×	✓
Verbot alleinstehender Module	×	×	✓	×
Importverbot eines externen Moduls	×	✓	✓	×

Abbildung 1: Analyseergebnisse der ausgewählten Architekturregeln
Quelle: Eigene Darstellung

Ein grüner Haken bedeutet, dass das Werkzeug die Definition der Regel unterstützt. Ein roter Haken bedeutet das Gegenteil.

Der Sensor Dependency Cruiser unterstützt sechs der sieben Architekturregeln. Die einzige Architekturregel, die sich mit dem Werkzeug nicht definieren lässt, ist eine Vorgabe für Dateinamen. Darunter ist zu verstehen, dass alle Dateiennamen in einem bestimmten Verzeichnis dem gleichen Schema entsprechen müssen. Eine Vorgabe für Dateinamen lässt sich lediglich mit TSArch realisieren. Das Werkzeug weist abgesehen davon allerdings keine Vorteile gegenüber dem Dependency Cruiser auf. TSArch ist wesentlich langsamer und kann im Gegensatz zu dem Dependency Cruiser auch keine Vue.js-Dateien analysieren.

Bei dem Qualitätsproblem „Unzureichende Testabdeckung“ ergab die Analyse, dass Istanbul für die meisten, aber nicht für alle Kontrollstrukturen die korrekte Zweigüberdeckung berechnet. Ein einfaches If-Else Konstrukt, ein ternärer Operator oder ein Switch-Case stellen keine Probleme dar. Ein verschachteltes If-Else Konstrukt oder eine Schleife hingegen schon. Istanbul identifiziert für die beiden problematischen Kontrollstrukturen nicht immer alle Zweige. Das hat zur Folge, dass eine vollständige Zweigüberdeckung nicht wie in der Theorie vorgesehen eine vollständige Anweisungsüberdeckung sicherstellt.

Quellen:

- [W. Löwe and T. Panas, “Rapid construction of software comprehension tools,” International Journal of Software Engineering and Knowledge Engineering, vol. 15, no. 06, pp. 995–1025, 2005.
- P. Liggesmeyer, Software-Qualität. Heidelberg: Spektrum Akademischer Verlag, 2009.
- T. A. Standish, “An essay on software reuse,” IEEE Transactions on Software Engineering, no. 5, pp. 494–497, 1984.
- D. W. Hoffmann, Software-Qualität: Springer-Verlag, 2013.