Visualisierung von Code-Smells in Code-Cities

Bachelorarbeit

Falko Galperin

Matrikelnummer: 4508458

27. September 2021

Fachbereich 3 — Mathematik und Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Prof. Dr. Jan Peleska

ZUSAMMENFASSUNG

In dieser Bachelorarbeit wird ein Ansatz zur Visualisierung von Code-Smells in Code-Cities und dessen Implementierung in der Visualisierungssoftware SEE beschrieben. Dabei soll die Forschungsfrage geklärt werden, ob sich Code-Cities besser hinsichtlich einiger Aspekte als traditionelle tabellarische Darstellungsformen eignen, um Code-Smells übersichtlich zu visualisieren. Code-Smells werden dabei von einem externen Server abgerufen, in SEE verarbeitet und in dessen Code-Cities, aber auch IDE-ähnlichen „Code-Windows“ visualisiert.

Im zweiten Teil der Arbeit wird eine Evaluation als Vergleichsstudie zwischen dem AXIVION-Dashboard (eine tabellarische Analysesoftware für Code-Smells) und SEE beschrieben und durchgeführt, welche die vorangegangene Implementierung testen und die Forschungsfrage klären soll. Die Teilnehmer bekamen in einer Online-Studie mehrere Aufgaben gestellt, die sie mit dem Dashboard und SEE lösen sollten. $n = 20$ Teilnehmer nahmen an dieser Studie teil. Deren Ergebnisse wurden im Hinblick auf die benötigte Zeit, den Anteil korrekter Antworten, die Usability unter Verwendung des *System Usability Scale* und den empfundenen Aufwand sowie die empfundene Komplexität mittels des *After-Scenario Questionnaire* auch unter Betrachtung bestehender Erfahrung ausgewertet.

Die Auswertung ergab bei der benötigten Zeit und dem empfundenen Aufwand ein signifikant ($p < 0,05$) besseres Ergebnis für SEE, beim Anteil korrekter Antworten und der empfundenen Usability hatte jedoch das Dashboard ein signifikant besseres Ergebnis. Dies legt die Vermutung nahe, dass sich Code-Cities vermutlich eher für eine schnelle Übersicht und tabellarische Formen hingegen besser für Detailanalysen eignen. Abschließend werden konkrete Optimierungsvorschläge genannt, um z. B. die Usability von SEE in der Zukunft zu verbessern.

ERKLÄRUNG

Ich versichere, die Bachelorarbeit — sofern dies nicht explizit anders gekennzeichnet wurde — ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 27. September 2021

Falko Galperin

DANKSAGUNG

Zuallererst möchte ich mich bei Rainer Koschke bedanken, der mich durch das schnelle und ausführliche Beantworten meiner zahlreichen Fragen bei der Implementierung, beim Schreiben der Bachelorarbeit und auch bei der Durchführung der Evaluation unterstützt hat, z. B. indem dort das Angebot einer Gratispizza für alle Teilnehmenden des Bachelorprojekts als Anreiz gesetzt wurde.

Marcel Steinbeck danke ich unter anderem für einen Performance-rettenden Tipp bei der Implementierung. Ebenso bedanke ich mich bei Christian Müller für das Testen und Verbreiten meiner Evaluation und natürlich auch bei allen anderen Teilnehmenden der Evaluation, die letztendlich für eine sehr zufriedenstellende Teilnehmerzahl gesorgt haben. Weiterhin möchte ich mich bei Thore Frenzel bedanken, der mich unter anderem beim Testen der Evaluation unterstützt hat. Abschließend möchte ich mich bei allen anderen Korrekturlesern bedanken, die trotz des Umfangs dieser Bachelorarbeit nicht vor der Aufgabe zurückgeschreckt sind.

GENDER-HINWEIS

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

INHALTSVERZEICHNIS

1 EINLEITUNG

In dieser Bachelorarbeit geht es um die Implementierung und Evaluation einer Visualisierung von Code-Smells in Code-Cities. Im ersten Kapitel beginnen wir[1] damit, einige Grundlagen, den Aufbau, sowie die Motivation hinter dieser Arbeit zu erläutern. Weiterhin legen wir die zentral zu untersuchende Forschungsfrage fest.

1.1 FORMAT

In diesem Dokument werden viele technische Begriffe eingeführt und verwendet, die zum Verständnis wichtig sind. Diese sind gesammelt im Glossar (Anhang A) nachschlagbar, ebenso wie die Akronyme (Anhang B). Jedes Mal, wenn ein nachschlagbarer Begriff das erste Mal verwendet wird, wird er in *dieser roten Farbe* abgedruckt und (falls dies nicht schon im Text selbst geschieht) in einer Randnotiz erklärt — in der Digitalversion dieser Bachelorarbeit lässt er sich außerdem anklicken und leitet zum entsprechenden Teil des Glossars. Alle darauffolgenden Verwendungen des Begriffes sind zwar immer noch anklickbar, werden aber nicht mehr farblich hervorgehoben. Insgesamt werden hier folgende verschiedene Farben verwendet:

Die Erklärung als Randnotiz sieht z. B. so aus.

- **Rotbraun** für die Einführung eines Glossarbegriffs,
- **Türkis** für interne Links (z. B. auf andere Abschnitte),
- **Magenta** für externe Links (z. B. auf Webseiten),
- **Grün** für zitierte Literatur und
- **Violett** für Verweise auf angehängte Dateien (siehe Anhang C).

1.2 MOTIVATION

Visualisierung im Allgemeinen hilft oft dabei, Eigenschaften komplexer Systeme zu verstehen, indem diese vereinfacht mithilfe eines betrachtbaren Modells dargestellt werden. Besonders im Bereich der Softwareentwicklung, bei der die Systeme nicht nur abstrakt sind und daher keine direkte Darstellungsform haben, sondern auch schnell unüberschaubar werden, kann eine Visualisierung der Struktur einer

1 Ich verwende in dieser Arbeit stellenweise „wir“, um den Leser mit einzubeziehen, nicht weil mehrere Personen bei der Ausarbeitung der Bachelorarbeit beteiligt waren.

solchen Software enorm hilfreich sein. Eine an der Universität Bremen entwickelte Visualisierung für Softwaresysteme nennt sich *Software Engineering Experience* (SEE) und wird im folgenden Abschnitt kurz vorgestellt.

1.2.1 *SEE*

Abbildung 1.1: Das Logo von SEE.

SEE ist eine in *Unity* implementierte interaktive Visualisierung von Software, welche die *Code-City*-Metapher verwendet und einen kollaborativen Mehrbenutzermodus über verschiedene Plattformen[2] hinweg ermöglicht. Diese wurde an der Universität Bremen unter anderem im Kontext vom Bachelorprojekt *See, Exchange, Arrange* (SEA) entwickelt.

Unity: *Eine Spiele-Engine, mit der man unter Verwendung der Programmiersprache C# 2D- und 3D-Spiele entwickeln kann.*

SEA: *Ein Bachelorprojekt der Universität Bremen, in welchem das Softwareprojekt SEE weiterentwickelt wurde.*

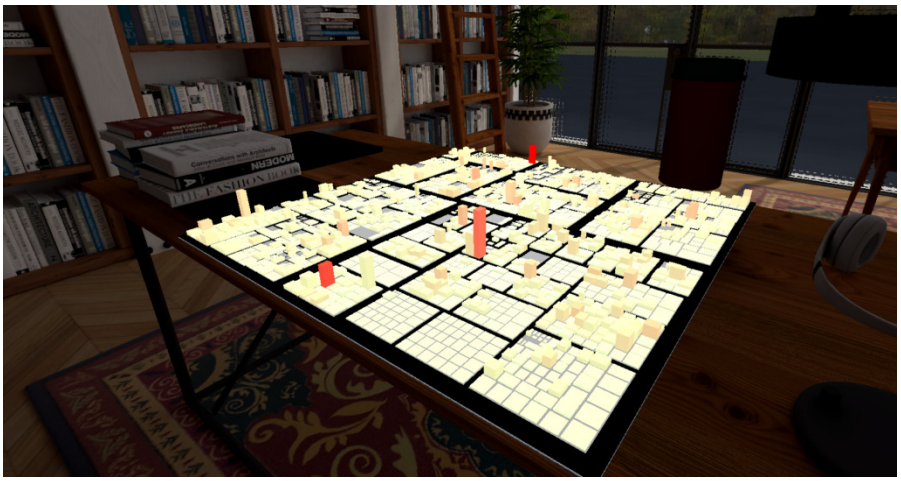


Abbildung 1.2: Der Code eines Softwareprojekts, dargestellt als Code-City in SEE.

In der Code-City-Metapher werden Softwarekomponenten durch Gebäude in einer Stadt repräsentiert, wobei die Eigenschaften dieser Gebäude verschiedene Metriken der Software ausdrücken können — z. B. könnte die Höhe eines Gebäudes der Anzahl der Codezeilen entsprechen. SEE verwendet den *Abhängigkeitsgraphen* als Datenmodell für die Visualisierung, dieser wird durch *Graph eXchange Language* (GXL)-Dateien repräsentiert. Die Elemente (Gebäude) in der Stadt nennen wir auch *Knoten* im Abhängigkeitsgraphen, die entsprechend der Programmstruktur hierarchisch angeordnet sind — eine Methode wäre z. B. einer Klasse untergeordnet.

Abhängigkeitsgraph: *Ein in SEE visualisierter Graph aus typisierten Knoten und Kanten, die zusätzliche Attribute (z. B. Metriken) enthalten können. Die Typen können dabei beliebig sein, wodurch SEE gut generalisierbar ist.*

GXL: *Ein Dateiformat für Graphen, wird z. B. in SEE zur Repräsentation von Projektgraphen verwendet.*

Klon: *Zwei unterschiedliche Stellen im Quellcode, die ähnlichen Code enthalten.*

Ein Beispiel dafür, wie so eine Code-City in SEE aussehen kann, kann in Abbildung 1.2 gefunden werden. Dort wird zusätzlich durch den Farbverlauf von weiß nach rot die *Klon*rate eines Knotens dargestellt — man sieht etwa, dass die rötlichen Gebäude in diesem Fall besonders viele Klone haben, wobei der mittlere auch noch wegen seiner Höhe ungewöhnlich viele Codezeilen hat. Dies ist im Vergleich zu tradi-

2 Neben Desktop- und Touch-Umgebungen noch Virtual Reality (z. B. *Valve Index*) und Augmented Reality (z. B. *Microsoft HoloLens*).

tionellen *Integrated Development Environments* (IDE) deutlich schneller erkennbar, da man dort erst einmal die entsprechenden Dateien öffnen müsste, abgesehen davon, dass neben der Höhe durch die Breite, Färbung und Positionierung im Vergleich zu anderen Gebäuden weitere Informationen erkannt werden können.

IDE: *Software, die zur Programmentwicklung verwendet wird, wie etwa* Eclipse *oder* JetBrains IntelliJ.

1.2.2 *Code-Windows*

Neben den Code-Cities lässt sich in SEE auch in den sogenannten *Code-Windows* der Quellcode selbst anzeigen (siehe Abbildung 1.3).

Code-Window: *Eine Darstellung von Quellcode in SEE, die der Darstellung in IDEs sehr ähnelt.*

Abbildung 1.3: Ein Code-Window.

Wie auch in IDEs üblich, verwenden die Code-Windows *Syntax-Highlighting*, um den Quelltext lesbarer zu gestalten. Indem man auf einen Knoten in der Code-City klickt, öffnet sich das zu diesem Element gehörige Code-Window und zeigt dessen Quelltext an — in Abbildung 1.3 wäre das z. B. die `CoseLayout`-Klasse. Am rechten und unteren Rand befinden sich Scrollbars, mit denen man durch die Datei scrollen kann. Im oberen Bereich in grau befinden sich Tabs, mit denen man zwischen mehreren offenen Dateien hin- und hernavigieren kann, wobei in der Abbildung nur ein einziger Tab offen ist.

Syntax-Highlighting: *Das farbliche Hervorheben von Stellen im Quellcode basierend auf der Syntax der Programmiersprache.*

1.2.3 *Code-Smells*

Code-Smells sind definiert als „bestimmte Strukturen im Code [...], die es nahelegen [...], dass ein Refactoring angebracht ist“ [Fow20]., etwa Codestellen mit schlechter Strukturierung.[3] IDEs, wie die in Abbildung 1.4 gezeigte JetBrains *IntelliJ IDEA*, markieren bestimmte Code-Smells oft im Code-Editor mithilfe eines integrierten statischen Analysetools, oder bieten — wie in der Abbildung — eine Möglichkeit,

3 Arten von Code-Smells werden in Abschnitt 2.1 konkretisiert.

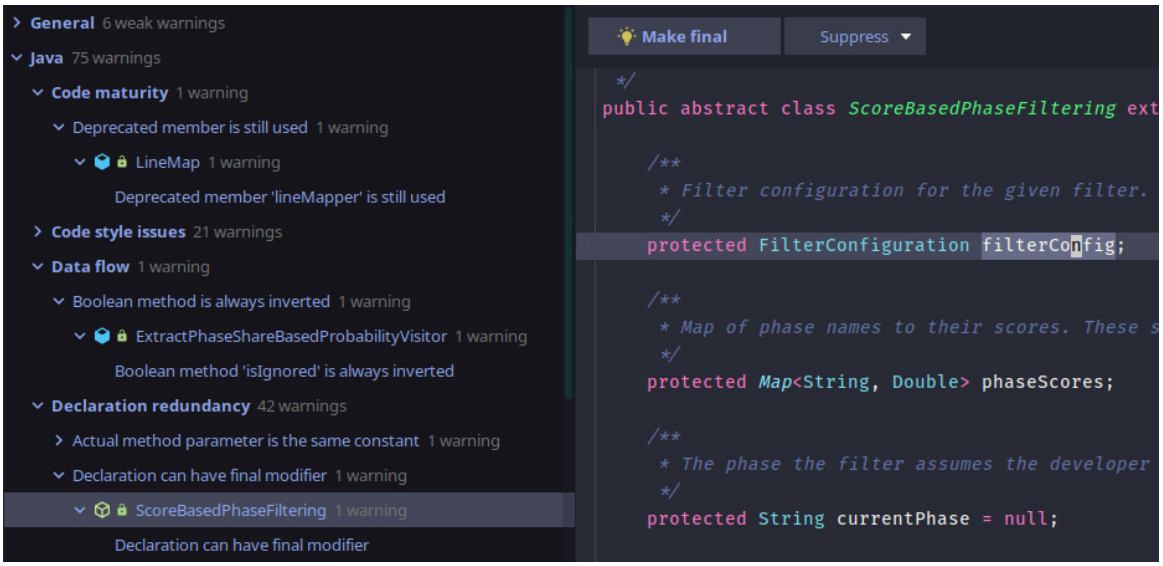


Abbildung 1.4: Auflistung von Code-Smells durch die JetBrains *IntelliJ IDEA*-IDE.

alle entdeckten Code-Smells in der gewählten Komponente aufzulisten. In der Abbildung lassen sich verschiedene Arten von Code-Smells erkennen, etwa, dass eine Komponente mit dem Namen `lineMapper` als `@Deprecated` markiert wurde und dennoch an manchen Stellen verwendet wird. Ausgewählt wurde hier das unterste Listenelement mit der Beschreibung „*Declaration can have final modifier*“ — hier kann augenscheinlich das Attribut `filterConfig` als `final` markiert werden. Hierfür wird dann rechts daneben die entsprechende Stelle im Code angezeigt und hervorgehoben. Oben in der Leiste gibt es einerseits die Option *Suppress* (um die Warnung zu unterdrücken) und andererseits die Option *Make final*, welche das beschriebene Problem löst.[4]

1.3 FORSCHUNGSFRAGE UND AUFBAU

Die zentrale Forschungsfrage, die in dieser Arbeit untersucht werden soll, lautet: *Eignen sich Code-Cities besser als traditionelle tabellarische Darstellungsformen, um Code-Smells übersichtlich zu visualisieren?*

Um diese Frage zu beantworten, werde ich zuerst in Kapitel 2 erläutern, wie die Visualisierung der Code-Smells konzeptuell umgesetzt werden kann, bevor ich dann die konkreten Details dieser Implementierung in Kapitel 3 beschreibe. Damit die Forschungsfrage beantwortet werden kann, führe ich in Kapitel 4 eine Evaluation mit einer später eingeführten Vergleichssoftware durch, bevor ich schließlich in Kapitel 5 einen Ausblick auf weitere Ausbaumöglichkeiten gebe und mit dem Fazit dieser Arbeit abschließe.

4 In diesem Fall würde die Zeile dann zu `protected final FilterConfiguration filterConfig;` abgeändert werden. So einen Vorschlag gibt es nicht bei allen Code-Smells.

2

ZIELE

In diesem Teil werden die Ziele dieser Bachelorarbeit — nämlich eine Visualisierung für Code-Smells in SEE zu implementieren — genauer erläutert und spezifiziert. Hierbei geht es allerdings nur um die Ziele der Implementierung. Die Ziele der Evaluation werden in Kapitel 4 gesondert behandelt.

Es sollen Code-Smells an den zugehörigen Gebäuden visuell hervorgehoben werden, sodass Nutzer von SEE sofort erkennen können, welche Teile der Software Überarbeitungsbedarf haben. Als weiteres Teilziel dieser Arbeit sollen Code-Smells nicht nur in den Gebäuden selbst visuell manifestiert werden, sondern auch in den *Code-Windows*, ähnlich wie man es aus traditionellen IDEs kennt, durch eine farbliche Hervorhebung.

Dabei ist es noch wichtig anzumerken, dass die Erkennung der Code-Smells selbst nicht Teil der Arbeit ist. Stattdessen wird die *Application Programming Interface* (API) des *Axivion-Dashboards* verwendet: SEE wird somit nicht selbst die Code-Smell-Detektion durchführen, sondern lediglich beim Programmstart bzw. in den Code-Windows „on demand“ die Daten hierfür von der API abrufen. Über das Dashboard sind außerdem noch weitere Metriken[1] abrufbar. Ein weiteres Ziel soll es sein, diese Metriken beim Programmstart abzurufen und den entsprechenden Elementen in der Code-City zuzuordnen, sodass diese anschließend in SEE verwendet und optional persistiert werden können. Der wesentliche Bestandteil der Bachelorarbeit (neben der Evaluation) liegt also in der Einbindung der Dashboard-Daten in SEE und in der visuellen Aufarbeitung davon für die Code-Cities und die Code-Windows. Diese Ziele werden nun noch einmal etwas genauer ausgeführt.

API: *Eine Software-Schnittstelle, die bereitgestellt wird, um Zugriff von anderer Software zu ermöglichen.*

Axivion-Dashboard: *Ein kommerzielles Produkt als Teil der Axivion-Suite, das Informationen zu Code-Smells von Softwareprojekten in einem Web-Interface zusammenträgt.*

2.1 ABRUFEN DER DATEN

Die Implementierung von SEE muss um eine Komponente erweitert werden, welche für die Eingabe von bestimmten Daten (URL, API-Schlüssel, Projektname) die zugehörigen Code-Smells von der *Representational State Transfer* (REST)-API des Axivion-Dashboards zurückgibt. Dementsprechend wird hier folgendes benötigt:

- Datenmodelle für die Code-Smells, damit diese in einer strukturierten Form dem Aufrufer zurückgegeben werden können.

REST: *Eine im Web oft verwendete Art von API, in der meist über HTTP-Anfragen an bestimmte Endpunkte Informationen abgerufen oder modifiziert werden können.*

1 Zum Beispiel die Anzahl der Codezeilen oder die Anzahl der `TODO`s.

Erosion: *Synonym zu Code-Smells.*

- Methoden, die diese Datenmodelle erzeugen und mit den Daten des AXIVION-Dashboard füllen, indem sie diese aus dem Netz abrufen[2]. Hierbei werden folgende Arten von Code-Smells zusammen mit dessen *Erosions*-Icons unterschieden[3]:
  - ***Architektur-Verletzungen***: eine Inkonsistenz zwischen Architekturmodell und Quellcode.
  - **Klone**: zwei unterschiedliche Stellen im Quellcode, die ähnlichen Code enthalten.
  - ***Zyklen***: mehrere unterschiedliche Stellen im Quellcode, die im Aufrufgraphen gegenseitig voneinander abhängen.
  - ***Toter Code***: eine Definition im Quellcode, die an keiner anderen Stelle im Programm genutzt wird.
  - ***Metrik-Verletzungen***: ein Metrikwert ist außerhalb eines definierten Bereiches für eine Codemetrik.
  - ***Stil-Verletzungen***: andere Arten von Problemen, hauptsächlich die Verletzung von Code-Konventionen.
- Falls das nicht bereits in den obigen Datenmodellen enthalten ist: Eine Methode, die für einen gegebenen Code-Smell eine Beschreibung bzw. Erklärung lädt.

2.2 CODE-WINDOWS

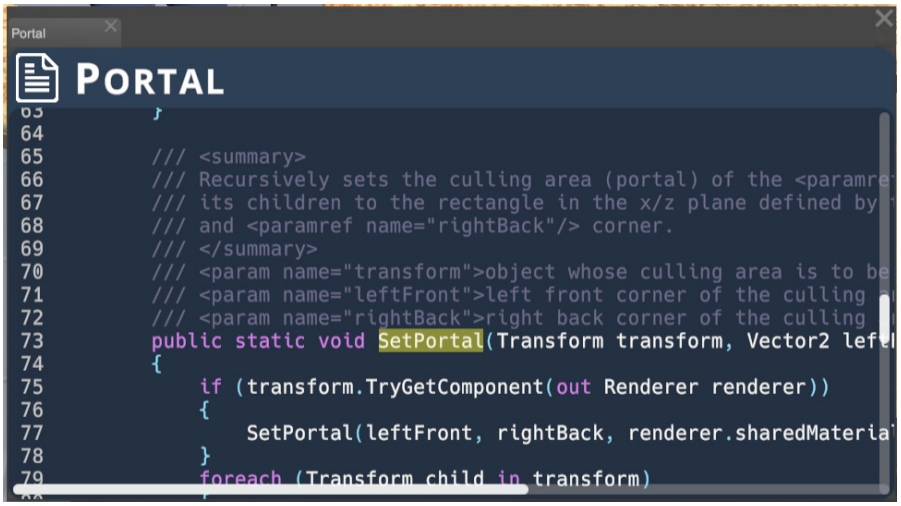


Abbildung 2.1: Mockup für die Markierung eines Code-Smells in SEE.

2 Hierfür wird eine Bibliothek für das Abrufen aus dem Netz und für das Wandeln der erhaltenen Daten zu C#-Klassen verwendet (siehe Kapitel 3).

3 Die Beschreibungen wurden hierbei dem Handbuch des AXIVION-Dashboards entnommen und ins Deutsche übersetzt.

Die Code-Smells sollen in den Code-Windows so angezeigt werden, wie man es aus IDEs wie *Eclipse*[4] oder JETBRAINS *IntelliJ IDEA*[5] kennt.

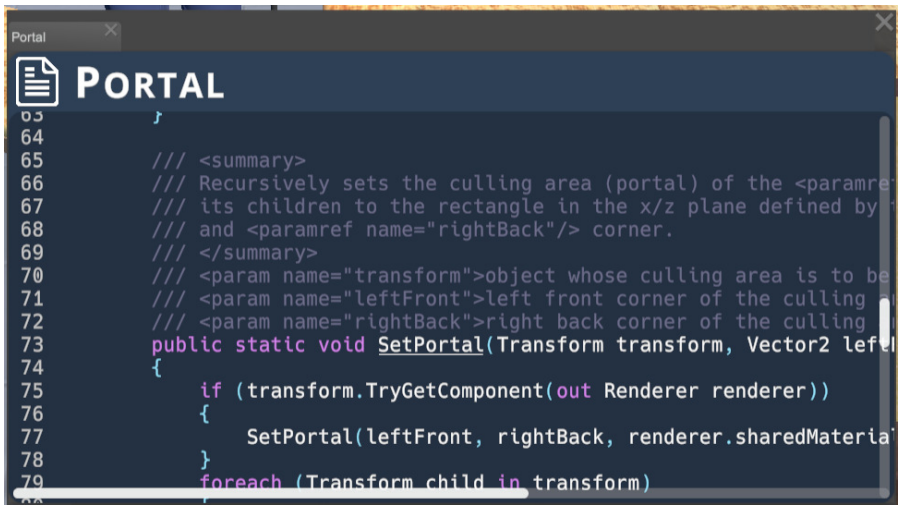


Abbildung 2.2: Mockup für die Unterstreichung eines Code-Smells in SEE.

Das heißt konkret, die entsprechenden Stellen sollen farblich markiert oder unterstrichen werden. Eine Veranschaulichung für die erstere Variante lässt sich in Abbildung 2.1 finden, eine für die letztere Variante in Abbildung 2.2. Hier wurde jeweils beispielhaft der Methodenname `SetPortal` markiert — das könnte z. B. passieren, wenn in der statischen Analyse aufgefallen ist, dass dieser Methodenkörper eine zu geringe Kommentardichte aufweist, denn dann würde es sich um eine Metrik-Verletzung handeln.

Der Nutzer sollte ebenfalls beim Hovern der Maus über diese Stelle Informationen über den Code-Smell eingeblendet bekommen. Das beinhaltet insbesondere eine Erklärung, wieso es sich an dieser Stelle um einen Code-Smell handelt; diese Erklärung wird von dem Axivion-Dashboard bereitgestellt und ist Teil der Daten, die durch die in Abschnitt 2.1 beschriebene Komponente abgerufen werden können. Weitere dort eingeblendete Informationen könnten z. B. bei einer Metrik-Verletzung die oberen und unteren Grenzen im Vergleich zum tatsächlichen Metrikwert sein. In den Code-Cities werden diese Informationen nicht einblendbar sein, da sich dies schwieriger integrieren lässt[6], dementsprechend muss man für diese Information zwangsläufig die Code-Windows verwenden.

Die Daten, die hierfür nötig sind, sollen „on-demand“ abgerufen werden, d.h. die Anfrage an das Axivion-Dashboard soll stattfinden, sobald das Code-Window geöffnet wird. Während der Ladezeit soll das Code-Window so wie vorher funktionieren (z. B. mit Syntax-Highlighting

4 https://www.eclipse.org/ (letzter Abruf: 23.08.2021)
5 https://www.jetbrains.com/idea/ (letzter Abruf: 23.08.2021)
6 Konzeptionell ließe sich das ähnlich wie in den Code-Windows so umsetzen, dass diese Detailinformationen nur beim Hovern angezeigt werden. Dies liegt aber außerhalb der Zielsetzung dieser Bachelorarbeit.

und ohne das Interface einzufrieren), außerdem soll es eine adäquate Indikation für den Ladevorgang geben, etwa in Form eines Ladebalkens.

2.3 CODE-CITIES

Hier müssen einerseits die in Abschnitt 2.1 beschriebenen Daten über die Code-Smells in die Code-Cities integriert werden, andererseits sollen diese Daten dann auch visualisiert werden.

Abbildung 2.3: Die alten Erosions-Icons in SEE.

Bei der Visualisierung soll es hauptsächlich darum gehen, die Anzahl an Code-Smells in einer bestimmten Datei schnell erkennbar zu machen, ähnlich wie sich bspw. die Färbung eines Gebäudes proportional zur Anzahl an Klonen verhält (siehe Abbildung 1.2). Hierfür sollen die bereits in SEE existierenden Erosions-Icons verwendet werden (siehe Abbildung 2.3). Die Erosions-Icons sind Symbole, welche Rauchschwaden ähneln, die oberhalb von Knoten mit Code-Smells erscheinen — diese müssen folgendermaßen angepasst werden:

- Die Anzahl an Code-Smells pro Typ soll sowohl aus den GXL-Dateien der Code-City, als auch aus den Daten des Axivion-Dashboards (siehe Abschnitt 2.1) entnommen werden können.
- Die Icons sollen sowohl an *Blattknoten* als auch an *inneren Knoten* des Abhängigkeitsgraphen erscheinen.
  - An den inneren Knoten sollen die aggregierten Code-Smell-Metriken erscheinen. Bei einer aggregierten Metrik wird die entsprechend zugeordnete Metrik aus den Kindknoten gelesen und summiert.
  - An den Blattknoten sollen die ausgelesenen Metriken erscheinen. Dies soll für Blattknoten in Form von Dateien

Blattknoten: *Ein Knoten in einem Baum, welcher keine weiteren Kindknoten besitzt.*

Innerer Knoten: *Ein Knoten in einem Baum, welcher Kindknoten besitzt.*

und Elementen auf einer niedrigeren Stufe als Dateien, z. B. Methoden oder Attribute, möglich sein.

- Die Skalierung eines Erosions-Icons wird immer in Abhängigkeit von der Code-Smell-Anzahl relativ zur maximalen Anzahl innerhalb aller Knoten mittels linearer Interpolation[7] berechnet, d.h. mit s_{min} und s_{max} als gewünschter minimaler bzw. maximaler Skalierung, mit v_{max} als maximaler Code-Smell-Anzahl und mit v als jeweiliger Code-Smell-Anzahl würde man für die jeweilige Skalierung s erhalten:

$$s = s_{min} \cdot \left(1 - \frac{v}{v_{max}}\right) + s_{max} \cdot \frac{v}{v_{max}}$$

  Bisher wurde für v_{max} immer die maximale Anzahl an Code-Smells innerhalb der gesamten Code-City verwendet. Dies macht aber bei Erosions-Icons an Inneren Knoten wenig Sinn[8], daher soll die Implementierung der Skalierung so geändert werden, dass v_{max} der maximalen Code-Smell-Anzahl *pro* Ebene innerhalb des Abhängigkeitsgraphen entspricht, anstatt global berechnet zu werden.

- Zuletzt soll es eine Möglichkeit geben, sich die exakten Zahlen der Code-Smell-Arten anzeigen zu lassen, etwa indem man über den entsprechenden Knoten hovert.

2.4 KONFIGURATION

Die Konfiguration der Funktionen dieses Abschnittes soll im *Unity-Inspektor* stattfinden. Insbesondere ist geplant, die Einstellungen für die Code-City-Anzeige dabei im Inspektor für das Code-City-*GameObject* zu platzieren. Dort soll folgendes einstellbar sein:

- Notwendige Daten, wie API-Schlüssel, URL, Projektname.
- Für jede Art von Code-Smell (siehe Abschnitt 2.1) soll einstellbar sein, ob diese in den Code-Windows dargestellt werden sollen.
  - Wenn ja, soll außerdem einstellbar sein, welche Farbe für diese Art verwendet werden soll.
- Für die Code-Cities:
  - Ob Metriken aus dem Axivion-Dashboard geladen werden sollen.

Ebene: *Eine Ebene in einem Graphen enthält alle Knoten, die einen bestimmten Abstand zum Wurzelknoten haben. Die oberste Ebene enthält dabei alleine den Wurzelknoten, die nachfolgende Ebene alle direkten Kinder des Wurzelknotens, usw.*

Unity-Inspektor: *Ein Teil des Editors der Spiele-Engine* Unity, *welcher einem Nutzer die Konfiguration von Spiele-Objekten ermöglicht. Wird in* SEE *z. B. für die Konfiguration der Code-Cities verwendet.*

GameObject: *Ein Objekt in Unity, das als Zusammenhalt von einzelnen Komponenten fungiert und z. B. eine Position, ein Aussehen und ein Verhalten haben kann.*

7 Es gibt auch eine Variante, die statt linearer Interpolation einen sogenannten *Z-Score* verwendet, dies wird aber im Rahmen dieser Bachelorarbeit nicht näher behandelt.

8 Dort würde schließlich z. B. die Code-Smell-Anzahl des Wurzelknotens der Summe aller enthaltenen Anzahlen entsprechen und daher per Definition deutlich größer als alle anderen Erosions-Icons dargestellt werden.

    * Wenn ja, sollte es die Möglichkeit geben, nur Code-Smells ab einer gewissen Version des Projekts anzuzeigen[9].
    * Wenn ja, soll außerdem einstellbar sein, ob existierende Metriken in der GXL-Datei durch die neuen Daten aus dem Dashboard überschrieben werden sollen.
  - Ob Code-Smells jeweils an Inneren Knoten und Blattknoten dargestellt werden sollen.

2.5 ZWISCHENFAZIT

Hier sind noch einmal die Ziele des Implementierungsteils dieser Arbeit in Stichpunkten zusammengefasst:

- Die Integration des Axivion-Dashboards in SEE als Programmierinterface, sodass für ein konfiguriertes Projekt zugehörige Daten bezüglich Code-Smells abgerufen werden können.
- Die Darstellung der abgerufenen Daten in den bereits beschriebenen Code-Windows in SEE, indem dort die entsprechenden Teile mit Code-Smells abhängig vom Typ farbig markiert werden.
  - Weiterhin soll beim Anklicken eines markierten Teils eine Erklärung zu dem zugehörigen Code-Smell erscheinen. Diese wird den Daten des ersten Punkts entnommen.
- Die Darstellung der abgerufenen Daten in den bereits beschriebenen Code-Cities in SEE, indem dort die Komponenten mit Code-Smells einen visuellen Indikator in Form der Erosions-Icons bekommen.
- Der Import der Metriken, die im Axivion-Dashboard verfügbar sind, in die Code-Cities, sodass diese Metriken dort für visuelle Indikatoren wie die Färbung eines Knotens verwendet werden können. Diese abgerufenen Metriken sollen dann für die jeweilige Code-City auch persistierbar sein.

9 Dies ist z. B. nützlich, um neu hinzugekommene Code-Smells zu untersuchen.

3

IMPLEMENTIERUNG

Nachdem ich eben in Kapitel 2 die Zielsetzung beschrieben und den Aufbau der Implementierung entworfen habe, will ich nun auf die genaue Umsetzung eingehen. Hierbei wird es um erwähnenswerte konkrete Details, aufgetretene Probleme und daraus resultierende Änderungen gehen. Wie im vorherigen Kapitel angedeutet, besteht die Implementierung im wesentlichen aus drei Teilen: Dem Abrufen der Daten, der Integration in die Code-Windows und der Integration in die Code-Cities — da die Umsetzung in dieser Reihenfolge stattgefunden hat, werden die Abschnitte hier auch so angeordnet.

Der gesamte Quellcode von SEE, inklusive der hier beschriebenen Änderungen, kann unter `https://github.com/uni-bremen-agst/SEE` gefunden werden[1]. Die eben beschriebenen drei Abschnitte wurden in Pull-Requests #352, #360 und #369 umgesetzt, dort können die entsprechenden Änderungen gefunden werden.

3.1 ABRUFEN DER DATEN

Das Ziel dieses Teils war es, ein Programmierinterface in SEE zu implementieren, über welches man auf eine einfache Weise die relevanten Daten des Axivion-Dashboards abrufen kann.

3.1.1 *Zugriff auf die API*

Auf die API des Axivion-Dashboards kann man (wie bei REST üblich) zugreifen, indem man HTTP-Anfragen an eine bestimmte URL stellt. Man authentisiert sich mithilfe eines speziellen HTTP-Headers, in welchem man einen vorher im Webinterface des Dashboards generierten Token mitliefert. Bei einer erfolgreichen Anfrage erhält man entweder die geforderten Daten[2] im *JavaScript Object Notation* (JSON)-Format oder eine Fehlermeldung. Um in Unity bzw. C# nun also HTTP-Anfragen an die URL des Dashboards zu erstellen, habe ich `UnityWebRequest`[3] verwendet, denn diese Lösung war bereits in Unity integriert und ließ sich einfach und zufriedenstellend nutzen, insbesondere zusammen mit

JSON: *Ein menschenlesbares Datenformat, das z. B. gerne in REST-APIs verwendet wird.*

1 Um Zugriff auf das Repository zu bekommen, bitte Rainer Koschke (`koschke@uni-bremen.de`) kontaktieren.

2 Theoretisch lassen sich über die API auch Daten im Dashboard manipulieren, dies wurde im Rahmen dieser Bachelorarbeit allerdings nicht getan.

3 `https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html` (letzter Abruf: 26.08.2021)

`UniTask` und der asynchronen Programmierung in C#, zwei Konzepte, die ich im nächsten Absatz kurz anreißen möchte.

In C# lässt sich das **`async`**-Schlüsselwort verwenden, um asynchrone Methoden zu definieren. Asynchrone Methoden sind wie normale C#-Methoden eine sequenzielle Abfolge von Anweisungen, haben jedoch den Unterschied, dass sie bei einer Anweisung mit dem **`await`**-Schlüsselwort, vereinfacht gesagt, nicht die Ausführung des gesamten Programms blockieren, sondern „im Hintergrund“[4] auf den Abschluss der Anweisung warten [Mic20]. Das ist z. B. für Netzwerkaufrufe ein besonders wichtiges Konzept, da bei einem synchronen Methodenaufruf die GUI blockieren[5] würde, bis die Netzwerkabfrage fertig bearbeitet wurde. Bei einem asynchronen Methodenaufruf hingegen wird die Kontrolle während der Netzwerkabfrage an den Aufrufer zurückgegeben, sodass Unity normal weiterlaufen kann und der Nutzer nicht für die Dauer der Abfrage warten muss. Die Konsequenz davon, **`async`**-Methoden zu verwenden ist jedoch, dass der Rückgabetyp `T` der Methode in ein `Task<T>`-Objekt „eingepackt“ werden muss, anstatt den Wert des Typen selbst zurückzugeben. Dies liegt daran, dass der Rückgabewert zum Zeitpunkt der Kontrollrückgabe noch nicht feststehen muss — dementsprechend sind viele Methoden sowohl in diesem Abschnitt als auch in den beiden folgenden (Abschnitt 3.2 und Abschnitt 3.3) asynchron durch das **`async`**-Schlüsselwort, da **`async`**-Methoden mittels **`await`** die `Task<T>`-Objekte wieder „auspacken“ können. Für eine effizientere [Kaw20] und einfacher handhabbare[6] Integration der asynchronen Konzepte in Unity verwende ich die *`UniTask`*[7]-Bibliothek — der Rückgabetyp wird in entsprechenden Methoden daher nicht wie eben beschrieben `Task<T>`, sondern `UniTask<T>` sein.

3.1.2 *Struktur*

In Abbildung 3.1 findet sich ein UML-Klassendiagramm dieses Projekts. Es ist noch wichtig anzumerken, dass es zwanzig weitere Klassen gibt, die in diesem Diagramm nicht enthalten sind, dabei handelt es sich um die Klassen des Namespace `SEE.Net.Dashboard.Model`. Dort befinden sich nahezu identische C#-Repräsentationen der JSON-Modelle, die im Handbuch der REST-API des Axivion-Dashboard beschrieben werden. Die Modelle wurden dabei nur sehr geringfügig angepasst, z. B. um die Vererbungsstruktur der *`Issue`*-Klassen besser in C# einbinden zu können — außerdem wurden, je nach Notwendigkeit, einige Methoden und Attribute hinzugefügt.

Issue*: Eine konkrete Ausprägung eines Code-Smells im Axivion-Dashboard.*

4 Hierbei ist noch das technische Detail wichtig, dass allein dadurch kein Code *parallel* (etwa auf verschiedenen CPU-Kernen) läuft, bei Verwendung einer **`await`**-Anweisung wird die Kontrolle lediglich an den Aufrufer zurückgegeben — hier handelt es sich also eher um Multithreading als um Multiprocessing.

5 Das heißt, sie würde nicht mehr auf Eingaben reagieren.

6 Zum Beispiel lassen sich durch `UniTask` die asynchronen C#-Konzepte mit Unitys sogenannten *Coroutinen* (`https://docs.unity3d.com/ScriptReference/Coroutine.`

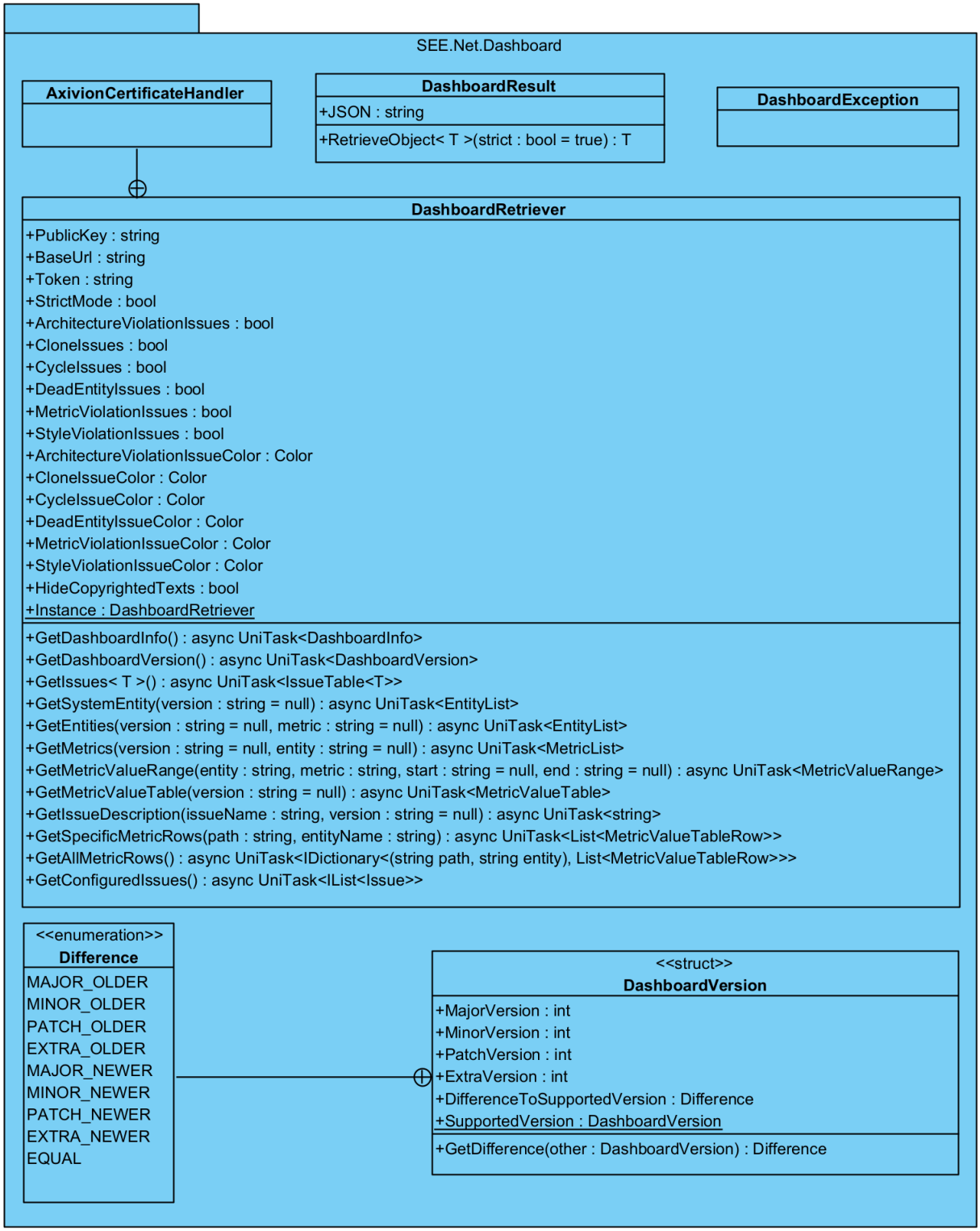


Abbildung 3.1: Ein vereinfachtes UML-Klassendiagramm für die Komponenten, die das Abrufen von Daten aus dem Dashboard ermöglichen. Das ⊕-Pfeilende bedeutet dabei, dass die Klasse am anderen Ende der Verbindung eine innere Klasse der Klasse an diesem Pfeilende ist.

Bevor wir auf die zentrale Komponente (den `DashboardRetriever`) zu sprechen kommen, gehen wir kurz auf die kleineren Klassen des Diagramms ein:

- **`DashboardException`**: Hier handelt es sich um eine eigene Exception, welche zusätzliche Informationen über einen fehlgeschlagenen Abruf enthält.
- **`DashboardResult`**: Eine Klasse, die intern die Rückgabe einer API-Anfrage an das Dashboard repräsentiert. Diese Klasse ist auch über die `RetrieveObject<T>`-Methode zuständig für das Deserialisieren der erhaltenen JSON-Daten zu den C#-Klassen des oben erwähnten `Model`-Namespace; hierfür wird die NEWTONSOFT *Json.NET*-Bibliothek[8] verwendet.
- **`DashboardVersion`**: Eine Klasse, die `IComparable` implementiert[9] und eine Versionsnummer des Dashboards in der Form `MAJOR.MINOR.PATCH.EXTRA`[10] repräsentiert. In der Klasse ist auch eine spezielle Vergleichsmethode `GetDifference` implementiert, welche den Grad des Unterschieds zweier Instanzen in Form der im Diagramm sichtbaren Aufzählung `Difference` zurückgibt. Diese Klasse wird benutzt, um die tatsächliche Version des Dashboards mit der unterstützen Version unserer Komponente abzugleichen — gleiche Versionen und solche, bei denen sich nur der `EXTRA`-Teil unterscheidet, ignorieren wir, aber im Falle eines sonstigen Unterschieds geschieht beim Programmstart folgendes (die Beschreibung bezieht sich hierbei immer auf die tatsächliche Version des Dashboards relativ zu der unterstützten):
  - *`MAJOR`/`MINOR` älter*: Da es in so einem Fall sein kann, dass wir noch nicht im Dashboard existente Features verwenden, geben wir eine Fehlernachricht an den Nutzer aus.
  - *`PATCH` älter*: Die Funktionalität sollte hierdurch nicht eingeschränkt werden, der Nutzer wird aber dennoch gewarnt, da der Patch Sicherheitslücken im Dashboard geschlossen haben könnte.
  - *`MAJOR` neuer*: Es könnten inkompatible Änderungen implementiert worden sein, also zeigen wir dem Nutzer ebenfalls eine Fehlernachricht.
  - *`MINOR`/`PATCH` neuer*: Diese Änderungen sollten nicht inkompatibel zum aktuellem Code sein, daher zeigen wir nur in

`html`, letzter Abruf: 12.09.2021) besser integrieren, was es etwa ermöglicht, ein `await` auszuführen, welches auf den nächsten Frame wartet.

7 https://github.com/Cysharp/UniTask (letzter Abruf: 27.08.2021)

8 https://www.newtonsoft.com/json (letzter Abruf: 27.08.2021)

9 Das bedeutet, Instanzen dieser Klasse lassen sich relativ zu anderen Instanzen der Klasse einordnen.

10 Siehe auch https://semver.org/ (letzter Abruf: 27.08.2021). Das Dashboard folgt diesem Standard wegen des *Extra*-Zusatzes nicht ganz.

der Konsole einen Hinweis, da der Code dennoch auf die neue Version aktualisiert werden sollte.

Nachdem wir den Nutzer auf den entsprechenden Umstand hinweisen, lässt sich die Klasse aber dennoch verwenden, da der Datenabruf weiterhin noch funktionieren könnte.

Die zentrale Komponente, welche auch in den späteren Abschnitten zum eigentlichen Abrufen der Daten verwendet werden wird, ist der `DashboardRetriever`. Dieser stellt die eigentlichen HTTP-Anfragen an das Dashboard und verwendet die in Abbildung 3.1 sichtbare Klasse `AxivionCertificateHandler`, um eine sichere Verbindung über HTTPS herstellen zu können. Hierfür werden nach außen mehrere Methoden definiert, um jeweils bestimmte Daten abzurufen, etwa für Metriken oder Issues. Diese lassen sich dabei auch nach bestimmten Parametern filtern, falls man z. B. nur Issues einer bestimmten Version sehen möchte. Ein Problem hierbei war, dass es für die Issue-Beschreibungen (siehe Abschnitt 2.2) keine offizielle API gab — auf Nachfrage bei einem Axivion-Mitarbeiter gab es nur eine inoffizielle Schnittstelle, die unter Eingabe des Namens eines Issues simples HTML zurückgibt, in welchem der Regeltext dann enthalten ist. Da ich aus Performance- und Zeitgründen keinen vollwertigen HTML-Renderer in SEE integrieren wollte, habe ich stattdessen unter Verwendung des *HTML Agility Pack*[11] *XML Path Language* (XPath)-Ausdrücke geschrieben, um die entsprechenden Teile des Regeltextes zu extrahieren.

XPath: *Eine Sprache, mit der sich Knoten in einem XML-Dokument (z. B. aus dem HTML-Dokument einer Website) herausfiltern lassen.*

Wie in Abbildung 3.1 ebenfalls zu sehen ist, hat der `DashboardRetriever` mehrere von außen setzbare Attribute, welche sich alle innerhalb von Unity durch den Unity-Inspektor über eine GUI konfigurieren lassen:

- `PublicKey` enthält den öffentlichen Schlüssel des SSL-Zertifikats für das Dashboard, dieser wird dann im `AxivionCertificateHandler` zur Sicherstellung der Integrität verwendet.
- `BaseUrl` enthält die URL zur Oberfläche des Dashboards.
- `Token` enthält den vorher im Webinterface des Dashboards erstellten Zugriffs-Token.
- `StrictMode` bestimmt, ob bei unbekannten JSON-Attributen ein Fehler auftreten soll.
- Für jeden Issue-Typ (siehe Abschnitt 2.1) lässt sich festlegen, ob dieser beim Aufrufen der Methode `GetConfiguredIssues` überhaupt abgerufen (und somit in den Code-Windows angezeigt) und in welcher Farbe dieser repräsentiert werden soll (siehe Abschnitt 3.2).

Ein weiterer erwähnenswerter Punkt des `DashboardRetrievers` ist, dass dieser weder als „normale“ Klasse — da der Anwendungsfall

11 https://html-agility-pack.net/ (letzter Abruf: 28.08.2021)

der Verwendung von mehreren verschiedenen Dashboard-Instanzen eher unwahrscheinlich wäre — noch als statische Klasse — da dann die Konfiguration über den Unity-Inspektor nicht möglich wäre — sondern als *Singleton* (unter Verwendung des `Lazy<T>`-Typen von C#) entworfen wurde. Der Zugriff auf die eine Instanz der Klasse erfolgt dementsprechend über das Attribut `Instance`. Singletons werden zugegebenermaßen stellenweise eher kontrovers gesehen [siehe z. B. Bri04], die Klasse ist aber von so einer Form, dass sie sich auch ohne zu großen Aufwand zu einem Nicht-Singleton umschreiben ließe.

Singleton: *Ein Software-Entwurfsmuster, in dem eine nach diesem Muster entworfene Klasse nur eine Instanz besitzen darf.*

3.2 INTEGRATION IN DIE CODE-WINDOWS

Das Ziel dieses Teilabschnitts war, wie in Abschnitt 2.2 beschrieben, die mit der Implementierung des vorhergehenden Abschnitts abrufbaren Code-Smells in den Code-Windows darzustellen. Hierzu ist es wichtig, erst auf ein paar Grundlagen der Code-Windows einzugehen, insbesondere, wie eine Quelldatei dort repräsentiert wird.

3.2.1 *Einlesen von Dateien*

Ein Code-Window ist ein *`MonoBehaviour`*, das instanziiert wird, sobald ein Nutzer einen Knoten in einer Code-City anklickt. Dieses sorgt bei der Instanziierung dann dafür, dass der Inhalt der mit dem Knoten assoziierten Datei geladen wird — um Syntax-Highlighting zu gewährleisten, wird die Eingabedatei dabei mithilfe eines von ANTLR[12] *(ANother Tool for Language Recognition)* generierten Lexers in einen sogenannten „*Token*-Stream“ zerlegt. Wir gehen diesen Strom an Tokens dann Stück für Stück durch und fügen dem Code-Window den Text des jeweiligen Tokens in einer Farbe, die vom Typ abhängt, zu. So entsteht das Syntax-Highlighting: Schlüsselwörter der Programmiersprache[13] etwa bekommen eine andere Färbung als Typennamen.

`MonoBehaviour`: *Ein bestimmter Klassen-Typ, den alle Unity-Skripte haben müssen. Diese dienen dann als Komponente eines GameObjects und werden (vereinfacht gesagt) während des Spiels ausgeführt.*

Token: *Ein einzelnes Element einer Programmiersprache, z. B. in C# ein Schlüsselwort wie `new`, oder der Name einer Variable.*

`TextMeshPro`: *Eine Komponente in Unity in Form eines `MonoBehaviours`, das Text auf vielfältige Weisen darstellen kann, z. B. in verschiedenen Farben und Schriftarten.*

Die Färbung selbst funktioniert dabei unter Verwendung des *`TextMeshPro`-Rich-Text*[14]-Systems. Hierbei handelt es sich um eine XML-ähnliche (aber nicht konforme) Syntax, die mit öffnenden und schließenden Tags das Format des Texts beeinflussen kann — so lässt sich z. B. das Wort „Hallo“ in rotem Text mit `<color=#FF0000>Hallo</color>` darstellen. Hiermit lässt sich über Verwendung des `<mark>`-Tags auch Text markieren bzw. mit `<ul>` unterstreichen, beides Funktionalitäten, welche wir später verwenden. Nun gibt es noch das Problem, dass im Quellcode selbst irgendwo ein Text wie `<color>` stehen könnte — dort wollen wir natürlich nicht, dass dies vom Rich-Text-System erkannt wird. Um solche Situationen zu vermeiden, gibt es den `<noparse>`-Tag: Aller von diesem Tag einge-

12 https://www.antlr.org/ (letzter Abruf: 07.09.2021)
13 SEE unterstützt dabei momentan C++, C# und Java.
14 http://digitalnativestudios.com/textmeshpro/docs/rich-text/ (letzter Abruf: 07.09.2021)

schlossene Text wird dann vom Rich-Text-System ignoriert, daher legen wir diesen jeweils um das eigentliche Wort herum, wodurch ein einzelner Token wie `new` am Ende in der Repräsentation des Code-Windows z. B. die Form `<color=#FF0000><noparse>new</noparse></color>` bekommen würde. Dies macht die Integration der Code-Smell-Daten jedoch komplizierter, worauf wir im nächsten Abschnitt eingehen.

3.2.2 *Integration der Code-Smell-Daten*

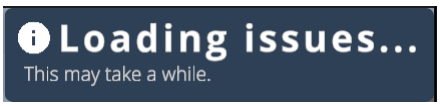
Nach dem Öffnen eines Code-Windows funktioniert dieses wie bereits zuvor. Sobald der Quelltext, wie oben beschrieben, aber fertig eingelesen wurde, werden im Hintergrund mittels des `DashboardRetrievers` alle Code-Smells dieser Datei abgerufen. Weiterhin im Hintergrund, d.h. unter Verwendung von `UniTask` auf einem separaten Thread, um das Code-Window während des Ladevorgangs weiter nutzen zu können, möchten wir nun die von Code-Smells betroffenen Bereiche visuell markieren[15]. Die API des Axivion-Dashboards gibt einem zu einem Code-Smell eine Start- und optional auch End-Zeilennummer, sowie den ebenfalls optionalen zu markierenden Text, falls nur ein Teil der Zeile(n) von dem Code-Smell betroffen ist.

Hier gab es jedoch mehrere Probleme, was mit dem vorhin beschriebenen Rich-Text-System einhergeht, denn das nachträgliche Einfügen von Tags in das `TextMeshPro` erforderte eine Art manuelles Parsen des bereits existierenden Pseudo-XML-Textes, um die neuen `<mark>`-Tags einzubauen. Außerdem gab es relativ viele Randfälle, z. B. gibt es noch die Möglichkeit von sich überlappenden Markierungen innerhalb einer Zeile, was vom `TextMeshPro` nicht unterstützt wird — in so einem Fall verwenden wir einfach die Farbe des ersten Code-Smells und konkatenieren die beiden Markierungen. In den Zusatzinformationen, die beim Anklicken angezeigt werden (siehe Unterabschnitt 3.2.3), werden dann beide Code-Smells erwähnt. Insgesamt hat die zusätzliche Komplexität durch das Rich-Text-System die Entwicklung dieses Teils deutlich in die Länge gezogen, letztendlich habe ich eine Woche länger als ursprünglich im Zeitplan vorgesehen benötigt. Zwar wurde die gewünschte Funktionalität erreicht, die bereits erwähnten Ladezeiten von 5–20 Sekunden, sowie die hohe Komplexität des Quellcodes, waren jedoch verbesserungswürdig.

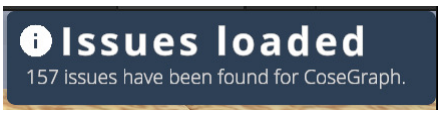
Ein wissenschaftlicher Mitarbeiter, der ebenfalls an SEE arbeitet, hatte dann einen alternativen Vorschlag: Wenn der Token-Stream zu Beginn gespeichert wird, lässt sich, nachdem die Code-Smells vom Dashboard abgerufen wurden, das Code-Window anschließend neu konstruieren — da wir diesmal aber die Code-Smell-Informationen

15 Wir verwenden hierfür die in Abbildung 2.1 sichtbare Markierung von Text, da die in Abbildung 2.2 sichtbare Unterstreichung, die ich zuerst verwendet habe, besonders bei kleinen Quellcodeelementen sehr schnell unterging und visuell nicht gut von normalem Text trennbar war.

bereits haben, können wir die Markierungen zum gleichen Zeitpunkt wie das Syntax-Highlighting in den Text integrieren, wir müssten uns also nicht mehr mit bereits existierendem Rich-Text beschäftigen. Das war ein deutlich sinnvollerer Ansatz, daher wurde die bisherige Implementierung dadurch ersetzt. Die Ladezeiten liegen nun nur noch zwischen 0,5–3 Sekunden, was eine sehr spürbare Verbesserung darstellt.



(a) Aktiver Ladevorgang.



(b) Abgeschlossener Ladevorgang.

Abbildung 3.2: Benachrichtigungen beim Laden und Verarbeiten der Code-Smells vom Axivion-Dashboard.

Während des Ladens wird der Nutzer über den Ladevorgang benachrichtigt (siehe Abbildung 3.2), währenddessen lässt sich das Code-Window normal nutzen. Die Lesbarkeit des Codes hat sich ebenfalls verbessert, da man nun deutlich weniger Sonderfälle behandeln muss (wir können jetzt z. B. das `<noparse>`-Tag einfach *nach* dem `<mark>`-Tag hinzufügen).

3.2.3 *Code-Smell-Beschreibung*

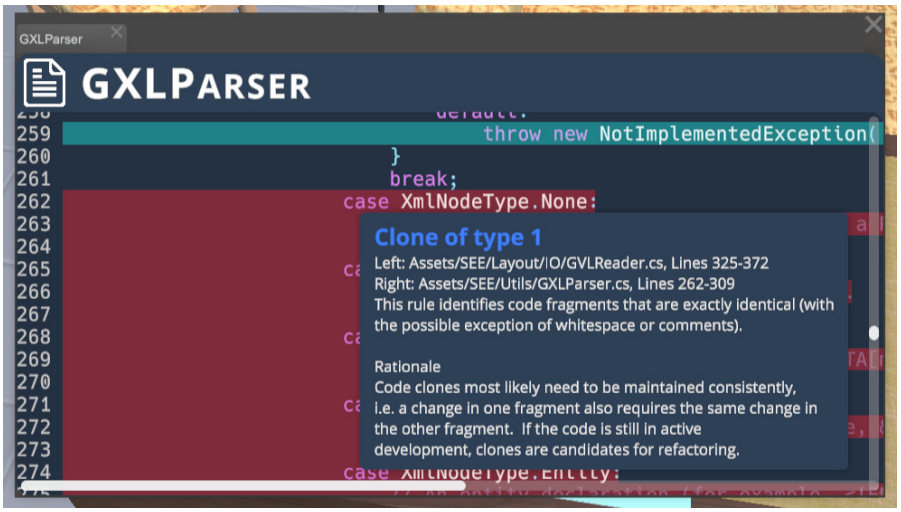


Abbildung 3.3: Details und eine Erklärung zu dem rot markierten Klon.

Eine weitere Anforderung war nun, die in Abschnitt 2.2 aufgestellt wurde, das Anzeigen von Informationen zu dem Code-Smell, sobald

man über die markierte Stelle hovert. Diese Anzeige war dank einer bereits vorhandenen `Tooltip`-Klasse[16] in SEE ohne größeren Aufwand umzusetzen. Wie ein solcher Informations-Tooltip aussieht, kann man in Abbildung 3.3 sehen. Die hier angezeigten Informationen sind folgende, unterschieden nach Code-Smell-Typ:

- **Architektur-Verletzung**: Art der Verletzung, Quell- und Zielentität in der Architektur, Quell- und Zielstelle im Quellcode
- **Klon**: Klontyp, beide Stellen im Quellcode
- **Zyklische Abhängigkeit**: Quell- und Zielstelle im Quellcode
- **Toter Code**: Die unerreichbare Stelle im Quellcode
- **Metrik-Verletzung**: Um welche Metrik es sich handelt, der erlaubte Bereich dieser Metrik, der tatsächlich gemessene Wert
- **Stil-Verletzung**: Eine Beschreibung der Stil-Verletzung
- Außerdem wird bei jedem Code-Smell-Typen am Ende des Informationstextes ein vom Dashboard abgerufener Erklärungstext für diesen Code-Smell angezeigt.

Für das Erkennen, ob die Maus über einem bestimmten Textabschnitt liegt, bietet das `TextMeshPro` eine entsprechende Methode, diese bringt jedoch bei größeren Dateien und insbesondere beim häufigen Aufrufen wieder vergleichsweise hohe Performance-Einbußen, daher öffnet sich die Anzeige der zusätzlichen Informationen stattdessen erst beim Anklicken. Geschlossen wird die Anzeige durch einen Rechtsklick; durch das Anklicken einer anderen Markierung wird der Inhalt der Anzeige entsprechend angepasst.

3.3 INTEGRATION IN DIE CODE-CITIES

Dieser Teil besteht aus zwei Unteraspekten, denen wir uns nacheinander widmen: Einerseits ist das die Integration der im Axivion-Dashboard gespeicherten Metriken in die Code-Cities und andererseits ist das die visuelle Darstellung der Code-Smell-Metriken an den Code-Cities.

3.3.1 *Integration der Metriken*

Bei der Integration der Metriken muss noch zwischen zwei Arten von Metriken unterschieden werden, nämlich einerseits den generellen Metriken wie z. B. Anzahl der Codezeilen und andererseits den Code-Smell-Anzahlen. Letztere müssen speziell behandelt werden, da sie nicht direkt als Metrik im Dashboard existieren. Hierauf kommen wir im Folgenden zu sprechen.

16 Mit dieser konnte beliebiger Text in einem kleinen Kasten, welcher sich mit der Maus mitbewegt, angezeigt werden.

CSV: *Ein simples Dateiformat für tabellarische Daten, in denen Spalten mit einem Komma und Reihen mit einem Zeilenumbruch getrennt werden..*

GENERELLE METRIKEN In der bisherigen Implementierung von SEE konnten Metriken entweder aus der GXL- oder einer *Comma-Separated Values* (CSV)-Datei ausgelesen werden. Hier wurde zusätzlich die Möglichkeit gegeben, die Metriken aus dem verknüpften Projekt des Axivion-Dashboards zu laden.

Die Implementierung war dabei vergleichsweise einfach, da die API des Dashboards eine Möglichkeit bietet, alle Metriken in einer tabellarischen Form (d.h. als Auflistung der Metrikwerte für jedes Element[17] im Dashboard) abzurufen. Dies tun wir also, sobald die Code-City geladen wurde, mittels `UniTask` wieder im Hintergrund, damit SEE auch während des Ladevorgangs die GUI nicht blockiert. Knoten von SEE werden über den Pfad der Datei und den Namen des Elements zu den Dashboard-Elementen zugeordnet, da diese beiden zusammengenommen eindeutig sein müssen. Die Metrikwerte werden dann in den Knoten der Elemente gespeichert und können über eine bereits in SEE existierende Funktionalität mithilfe des „Save Graph"-Buttons auch in der GXL-Datei persistiert werden.

CODE-SMELL-ANZAHLEN Die Daten über die Vorkommnisse der Code-Smells in den einzelnen Elementen sind nicht in der eben erwähnten Methode zum Abrufen aller Metriken enthalten, da diese nicht elementweise als Metrik im Dashboard gespeichert sind. Stattdessen erhalten wir die Code-Smells wie in Abschnitt 3.2 mit Dateipfad sowie optionaler Start- und Endzeilennummer und optionalem Inhalt innerhalb der Zeile(n), anstatt wie bei den generellen Metriken mit Dateipfad und Elementname. Dementsprechend ist die eindeutige Zuordnung hier etwas komplexer.

Wie ich dies letztendlich implementiert habe, kann in Form von vereinfachtem Pseudocode in Algorithmus 1 gesehen werden — dort ist die Funktion SetCodeSmellMetrics dafür zuständig, die vom Dashboard abgerufenen Code-Smell-Zahlen in die Knoten der Code-City einzuspeisen. Diese ruft für alle Knoten des Abhängigkeitsgraphen G die relevanten Code-Smells des jeweiligen Knotens ab und schreibt die Werte als Metrikattribute in die entsprechenden Knoten, wobei bestehende Werte nur überschrieben werden, wenn der Parameter O wahr ist (siehe Abschnitt 3.4). Code-Smells sind genau dann relevant für einen Knoten, wenn deren Zeilen die Zeilen des Knotens abdecken. Um diese Code-Smells zu bestimmen, gibt es die Funktion GetRelevantCodeSmells, die für einen übergebenen Knoten alle relevanten Code-Smells ermittelt. Dazu werden alle Zeilen des Knotens (der Bereich zwischen Start- und optionaler Endzeilennummer) mit den Zeilen aller Code-Smells verglichen; wir wählen dann solche, bei denen die letzteren eine Teilmenge der ersteren sind.

17 Ein Element kann dabei eine Klasse, eine Methode, ein Interface usw. der Quellcodehierarchie repräsentieren.

Algorithmus 1 Import der Code-Smells in SEEs Abhängigkeitsgraphen.

Vorbedingung: O ist wahr, wenn Metriken überschrieben werden sollen.
Vorbedingung: G ist der Abhängigkeitsgraph des Softwareprojekts in SEE.

```
1  function SetCodeSmellMetrics(G, O)
2      N ← ∅                                   ▷ Enthält alle bereits besuchten Knoten.
3      for all K ∈ V(G) do                     ▷ Alle Knoten im Graph.
4          for all r ∈ GetRelevantCodeSmells(K) do
5              T ← Type(r)                     ▷ Typ des Code-Smells.
6              if ∃K_T then                    ▷ Code-Smells des Typs T im Knoten K.
7                  if K ∈ N then
8                      K_T ← K_T + 1
9                  else if O then
10                     K_T ← 1                 ▷ Überschreiben, also von vorne zählen.
11                     N ← N ∪ {K}             ▷ Knoten als besucht markieren.
12             else
13                 K_T ← 1
14                 N ← N ∪ {K}

15 function GetRelevantCodeSmells(K)
16     C ← alle Code-Smells für den Dateipfad von K.
17     if Z_1 ∈ K then                         ▷ Knoten hat Startzeilennummer Z_1.
18         if Z_2 ∈ K then                     ▷ Knoten hat Endzeilennummer Z_2.
19             L_α ← {Z_1, Z_1 + 1, ..., Z_2}
20         else
21             L_α ← {Z_1}
22         R ← ∅      ▷ Alle Code-Smells relevant, deren Zeilen L_α abdecken.
23         for c ← {C_1 ... C_n} do
24             l_1 ← Startzeile von c
25             if l_2 ∈ c then                 ▷ Code-Smell hat Endzeile l_2.
26                 L_β ← {l_1, l_1 + 1, ..., l_2}
27             else
28                 L_β ← {l_1}
29             if L_β ⊆ L_α then
30                 R ← R ∪ {c}
31     else
32         R ← C                   ▷ Alle Code-Smells des Dateipfads sind relevant.
33     return R
```

Nachdem die Schritte von Algorithmus 1 ausgeführt wurden, werden mithilfe der bereits existierenden `MetricAggregator`-Klasse die Code-Smell-Anzahlen rekursiv auf die Eltern aggregiert — das heißt, ein innerer Knoten hat als Code-Smell-Anzahl die Summe der Code-Smell-Zahlen seiner Kindknoten[18].

3.3.2 *Darstellung in den Code-Cities*

Die bisherige Darstellung durch die Erosions-Icons (siehe Abbildung 2.3) musste zuerst angepasst werden, da diese seit längerem nicht mehr getestet wurde — außerdem waren die Icons bisher nur auf die Darstellung an Blattknoten ausgelegt und mussten insofern angepasst werden, dass die vorhin erwähnten aggregierten Code-Smell-Metriken an den inneren Knoten ebenfalls durch die Icons visualisiert werden.

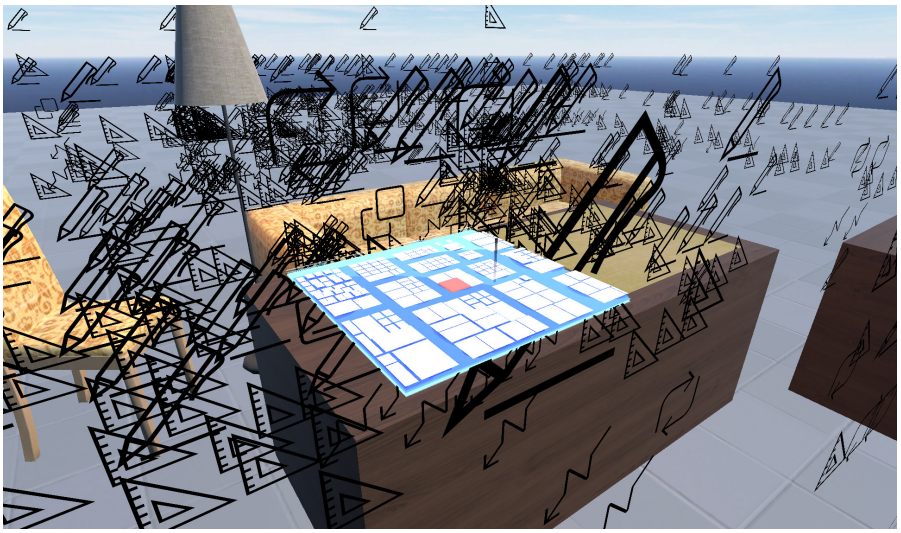


Abbildung 3.4: Darstellungsprobleme bei den Erosions-Icons.

Nachdem die Icons dementsprechend angepasst wurden, sah die Darstellung suboptimal aus (siehe Abbildung 3.4), und zwar in mehreren Aspekten, die ich nach und nach verbessert habe:

- Die Code-City befindet sich in einem so genannten *Portal*. Alles, was außerhalb des Portals liegt, wird nicht dargestellt — in Abbildung 3.4 lassen sich die Begrenzungen des Portals an der quadratischen blauen Fläche der Code-City erkennen, denn die Code-City wurde hier vergrößert, aber alles außerhalb der sichtbaren quadratischen Fläche wird abgeschnitten. Der Sinn des Portals ist, dass beim Bewegen und vor allem Vergrößern der Code-City dessen Elemente abgeschnitten werden, wenn sie den Tisch verlassen. Bei den Erosions-Icons ist dies offensichtlich nicht der Fall, da diese das Portal ignorieren.

18 Hierbei wird in post-order vorgegangen, damit die Code-Smell-Zahlen in den Blattknoten zuerst aggregiert werden.

  - Das Portal wird für die Code-City mittels eines speziellen *Shaders* umgesetzt. Diesen konnte ich nicht einfach für die Icons verwenden, da die Icons einen sogenannten `SpriteShader` verwenden, der speziell für die Darstellung von Icons ausgelegt ist. Dementsprechend habe ich eine modifizierte Version des `SpriteShaders` namens `SpritePortalShader` geschrieben, welcher das Portal auch für die Icons umsetzt.

Shader: *In Unity sind Shader auf der GPU laufende, in der C-ähnlichen Sprache HLSL geschriebene Programme, die bestimmen, wie Objekte des Spiels in Bezug auf einzelne Pixel dargestellt werden.*

- Es werden alle Symbole in der gleichen Größe gerendert, was es nahezu unmöglich macht, einzelne Icons irgendwelchen Knoten zuzuordnen, da diese visuell kaum voneinander trennbar sind.
  - In der bisherigen Implementierung musste eine maximale Breite für die Icons angegeben werden, diese werden dann abhängig von ihrem Vorkommen so skaliert, dass das Icon für den Code-Smell-Typ mit der größten Anzahl diese Maximalbreite bekommt. Die anderen Icons werden dann relativ zu dieser Maximalbreite skaliert.

    Um das eben geschilderte Problem zu lösen, wurde dies dahingehend geändert, dass die Maximalbreite nun der Breite des zugeordneten Knotens entspricht. Dadurch sind die Icons nicht mehr größer als der zugehörige Knoten. Als ein weiterer Nebeneffekt greift hier das *Level of Detail* (LOD) wegen der geringeren Größen der Icons deutlich besser: Weiter entfernte (und somit eher uninteressante) Icons werden dadurch schneller ausgeblendet. Dies sorgt für eine verbesserte Trennbarkeit der einzelnen Icons.

LOD: *Ein Mechanismus in 3D-Anwendungen, bei dem der Detailgrad eines Objekts reduziert wird (oder dieses sogar komplett ausgeblendet wird), je weiter der Spieler davon entfernt ist.*

- Schließlich wurden die Icons in der falschen Reihenfolge gezeichnet, d.h. die Icons wurden vor der Code-City gezeichnet, obwohl diese eigentlich über sie gezeichnet werden müssen. Dies hat den Effekt, dass (wie auch in Abbildung 3.4 zu sehen ist) die Erosions-Icons von der Code-City verdeckt wurden.
  - Da ich bereits für die Lösung des vorhergehenden Problems einen Shader geschrieben habe, war die Lösung hier einfach umzusetzen, da die Renderreihenfolge hierdurch auf einfache Weise auf einen höheren Wert als die der Code-City gesetzt werden konnte, was das Problem gelöst hat.

Die Ergebnisse dieser Anpassungen können in Abbildung 3.5 gesehen werden. Nun waren laut unserer Zielsetzung in Abschnitt 2.3 noch einige Dinge zu tun. Die relative Anzahl an Code-Smells wurde bisher immer über die Größe eines Icons im Vergleich zu den anderen Icons dargestellt — das wurde jedoch durch die Änderungen eben, durch welche die Skalierung nun über verschiedene Knoten hinweg unterschiedlich ist, zu verwirrend und machte den Unterschied nicht mehr gut erkennbar. Aus diesem Grund habe ich an dieser Stelle auf eine farbbasierte Darstellung der relativen Größe gewechselt: Knoten mit einer höheren

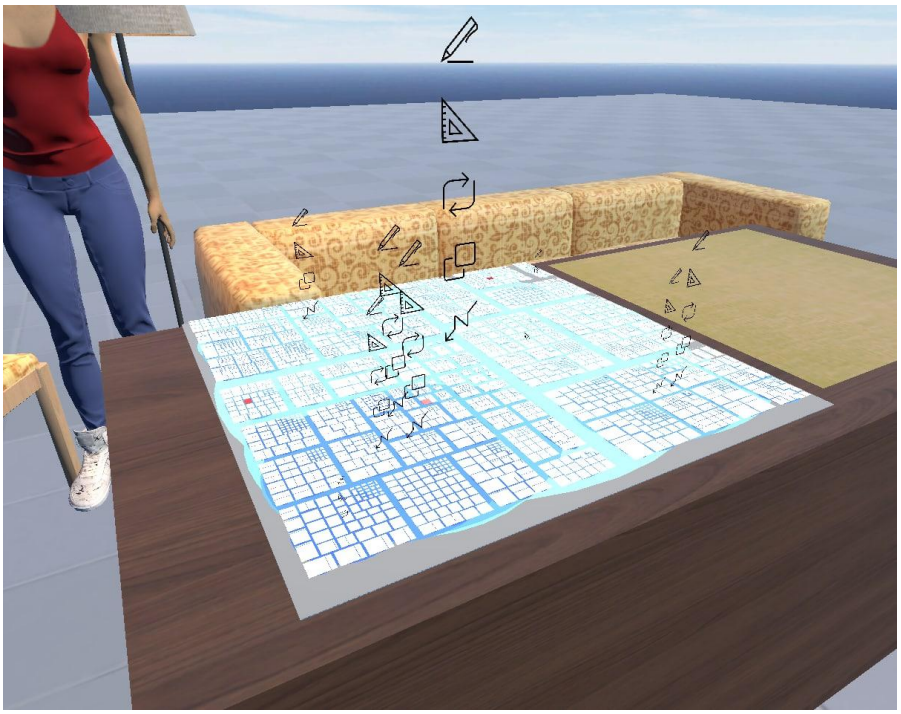


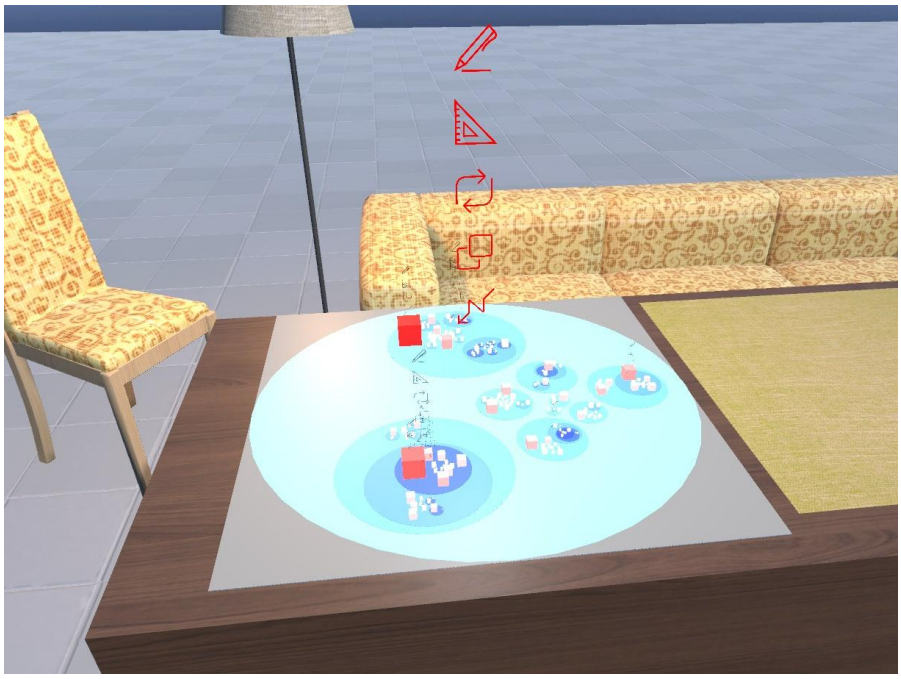
Abbildung 3.5: Verbesserungen bei den Erosions-Icons. Bei der Dame links oben handelt es sich um eine in SEE integrierte virtuelle Assistentin, sie spielt in dieser Bachelorarbeit aber keine Rolle.

Anzahl eines Code-Smell-Typs im Vergleich zum Rest bekommen rötere Erosions-Icons als Knoten mit geringeren Zahlen, bei denen die Icons schwärzer sind.

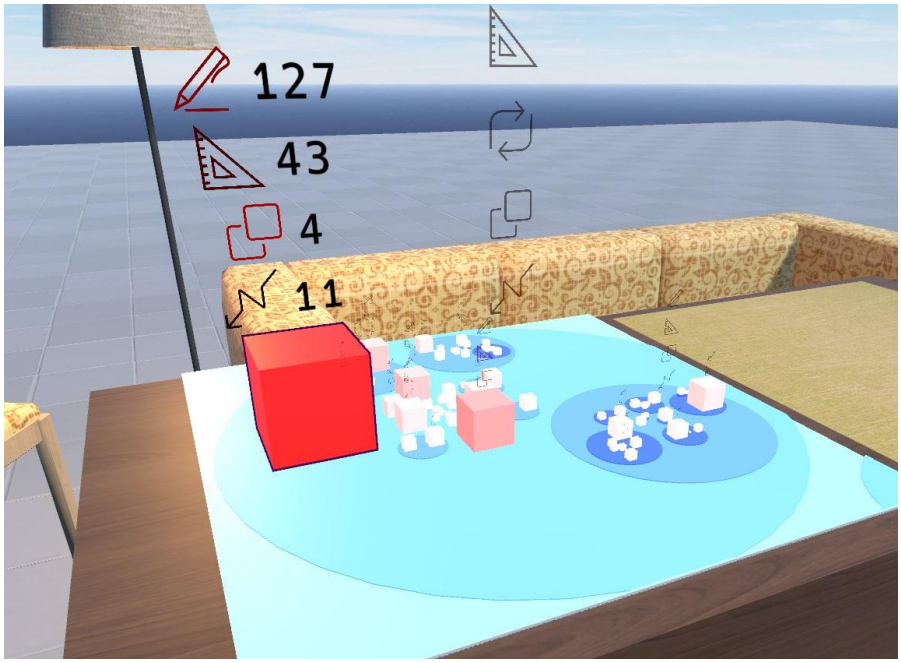
Da die Icons bisher nur auf Blattknoten ausgelegt waren, hatte die relative Färbung (bzw. früher die relative Größe) jedoch das Problem, dass der Wurzelknoten als Code-Smell-Zahlen die Summe aller Unterknoten bekam, was dazu führte, dass dieser im relativen Vergleich alle anderen Knoten in den Schatten stellte und somit als einziger stark rot gefärbt wurde, während die anderen alle schwarz gefärbt wurden. Um das zu lösen, wurde eine alternative Normalisierung implementiert, welche diese über *Ebenen* statt über den gesamten Abhängigkeitsgraphen hinweg berechnet.

Als letzte Änderung wurde schließlich noch ein Verhalten für das Hovern über die Knoten implementiert: Dabei werden (abgerundet durch eine Animation) erstens die Icons größer skaliert, um diese besser erkennen zu können und zweitens die exakten Code-Smell-Zahlen neben dem entsprechenden Erosions-Icon angezeigt. Das Ergebnis dieser letzten Änderungen kann in Abbildung 3.6 gesehen werden, wobei in Abbildung 3.6a anhand des Wurzelknotens die rote Verfärbung[19] der Icons sowie das LOD und in Abbildung 3.6b die Situation nach dem Hovern gut erkannt werden kann.

19 Hierbei haben alle Icons die volle Rotfärbung abbekommen, da der Wurzelknoten das einzige Element seiner Ebene ist.



(a) Sicht auf den Wurzelknoten von SEE mit aktiven Erosions-Icons.



(b) Sicht auf einen Knoten in SEE, während die Maus darüber hovert.

Abbildung 3.6: Finales Aussehen der Erosions-Icons.

3.4 KONFIGURATION

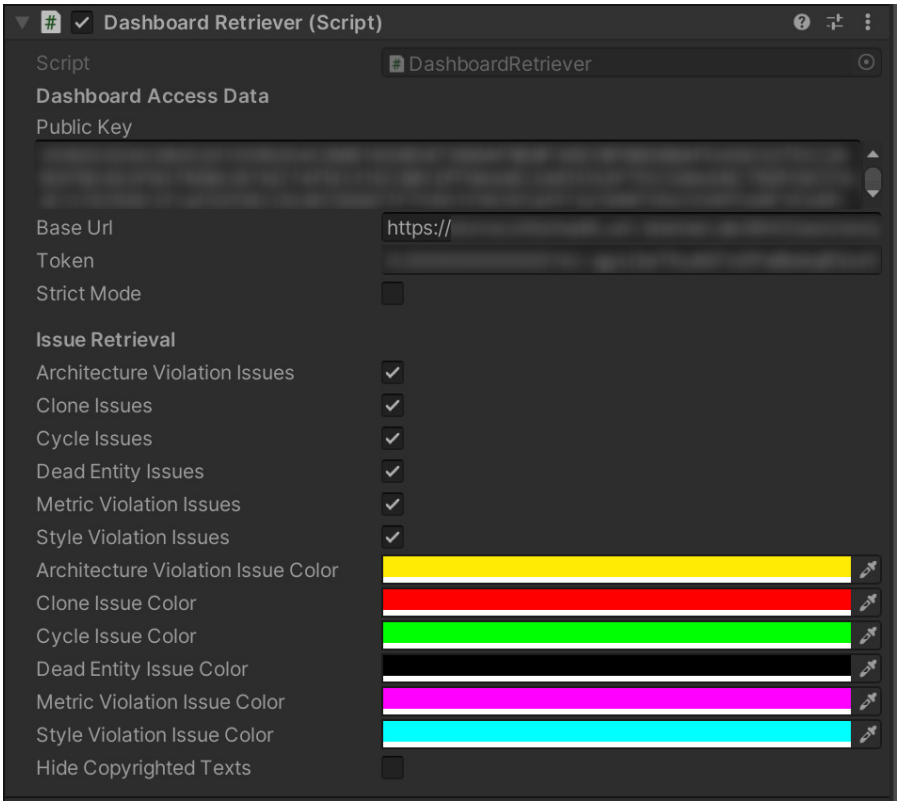


Abbildung 3.7: Die Konfigurationsoptionen im Unity-Inspektor für die Abrufkomponente und die Anzeige der Code-Smells in den Code-Windows. Die URL, der öffentliche Schlüssel und der Token für das Dashboard wurden hierbei zensiert.

Die Daten für die Abrufkomponente (siehe Abschnitt 3.1) — d.h., der öffentliche Zertifikatsschlüssel, die URL des Axivion-Dashboards sowie dessen Token — können im oberen Teil des in Abbildung 3.7 sichtbaren Unity-Inspektors für die `PlayerSettings`-Komponente[20] gefunden werden. Hier lässt sich außerdem die in Unterabschnitt 3.1.2 erklärte Option *Strict Mode* aktivieren.

Im unteren Teil der Abbildung lassen sich die Anzeigeoptionen für die Code-Smell-Anzeige in den Code-Windows (siehe Abschnitt 3.2) konfigurieren. Wie zu sehen ist, lässt sich für jeden der sechs Code-Smell-Typen (wie in Unterabschnitt 3.1.2 dargelegt) einstellen, ob und in welcher Farbe diese angezeigt werden sollen. Die Option *„Hide Copyrighted Texts“* versteckt alle Texte, die urherberrechtlich geschützt sein könnten, z. B. weil sie Zitate aus dem MISRA-Standard sind[21] und wurde hinzugefügt, nachdem mich ein Axivion-Mitarbeiter auf diesen Umstand aufmerksam machte.

20 Diese wurde hierfür gewählt, da sich dort auch andere Einstellungen von SEE befinden.
21 Dies lässt sich am Regelnamen erkennen, hierfür wurde also eine Blacklist an nicht erlaubten Regeltexten implementiert.

Abbildung 3.8: Die Konfigurationsoptionen im Unity-Inspektor für die Integration und Darstellung der Code-Smells in den Code-Cities.

In Abbildung 3.8 ist sichtbar, wie sich die in Abschnitt 3.3 beschriebenen Funktionalitäten konfigurieren lassen: Dort kann eingestellt werden, ob die Code-Smells mithilfe der Erosions-Icons an inneren Knoten und Blattknoten angezeigt werden sollen. Außerdem kann das Abrufen der Metriken (inklusive Code-Smell-Zahlen) aus dem Axivion-Dashboard hier aktiviert werden, ebenso kann eine Version angegeben werden, um nur die Issues zu verwenden, die ab einer bestimmten Version im Vergleich zur aktuellen Version hinzugekommen sind. Schließlich kann noch angegeben werden, ob in den GXL- oder CSV-Dateien bereits existierende Metriken überschrieben werden sollen (dies wäre in Algorithmus 1 die Variable O) und mit welchem Faktor die Größe der Erosions-Icons skaliert werden soll.

3.5 ZWISCHENFAZIT

Zusammenfassend wurde, wie in Kapitel 2 geplant, eine Programmierschnitstelle in SEE für das Abrufen der Code-Smells implementiert, die anschließend für die IDE-ähnliche Darstellung in den Code-Windows und der Visualisierung in den Code-Cities verwendet wurde. Da es sich bei dem letzten Punkt um den vergleichsweise unkonventionellsten Teil handelt, soll dieser im nächsten Abschnitt im Hinblick auf die Eignung evaluiert werden, um die Forschungsfrage („Eignen sich Code-Cities besser als traditionelle tabellarische Darstellungsformen, um Code-Smells übersichtlich zu visualisieren?“) zu klären.

4

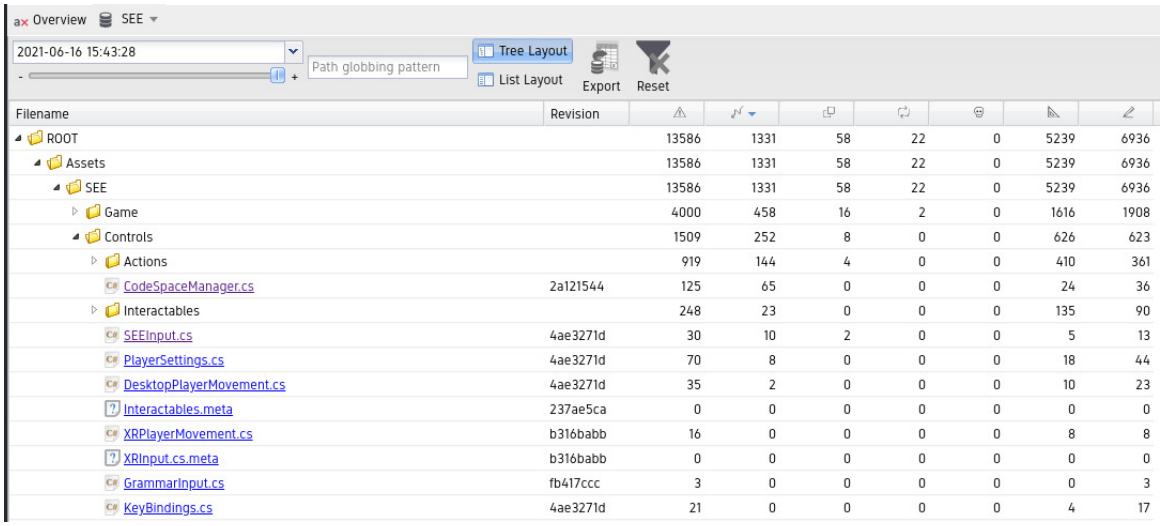
EVALUATION

Im Folgenden wird die Darstellung der Code-Smells in den CodeCities durch eine Nutzerstudie evaluiert. Dies wird getan, indem die Darstellung der Code-Smells in SEE mit der Darstellung der Code-Smells im AXIVION-Dashboard verglichen wird.

Wir beschreiben hierzu zuerst das Vergleichsprodukt (das AXIVION-Dashboard), gehen auf die Wahl des Fragebogens und die Aufgabe selbst ein, beschreiben dann die konkrete Umsetzung und schließen mit der Auswertung und möglichen Threats to Validity ab.

4.1 AXIVION-DASHBOARD

In diesem Abschnitt wird die mit SEE zu vergleichende Software kurz eingeführt: Das AXIVION-Dashboard, eine Software, welche unter anderem Code-Smells eines analysierten Projekts strukturiert anzeigen kann.



Filename	Revision							
ROOT		13586	1331	58	22	0	5239	6936
Assets		13586	1331	58	22	0	5239	6936
SEE		13586	1331	58	22	0	5239	6936
Game		4000	458	16	2	0	1616	1908
Controls		1509	252	8	0	0	626	623
Actions		919	144	4	0	0	410	361
CodeSpaceManager.cs	2a121544	125	65	0	0	0	24	36
Interactables		248	23	0	0	0	135	90
SEEInput.cs	4ae3271d	30	10	2	0	0	5	13
PlayerSettings.cs	4ae3271d	70	8	0	0	0	18	44
DesktopPlayerMovement.cs	4ae3271d	35	2	0	0	0	10	23
Interactables.meta	237ae5ca	0	0	0	0	0	0	0
XRPlayerMovement.cs	b316babb	16	0	0	0	0	8	8
XRInput.cs.meta	b316babb	0	0	0	0	0	0	0
GrammarInput.cs	fb417ccc	3	0	0	0	0	0	3
KeyBindings.cs	4ae3271d	21	0	0	0	0	4	17

Abbildung 4.1: Der *Project Index* des AXIVION-Dashboards für den Quellcode von SEE.

Während es zwar noch viele weitere Features gibt — wie z. B. die REST-API, welche wir für das Abrufen der Code-Smell-Daten verwenden (siehe Abschnitt 2.1) — ist für die Evaluation nur der *Project Index* relevant, der auch in Abbildung 4.1 zu sehen ist. Dort wird der Ordnerbaum des zu untersuchenden Projekts mit den zugehörigen aggregierten und nach Typ getrennten Code-Smell-Zahlen angezeigt.

Diese auch *Tree View* genannte Ansicht lässt sich[1] einerseits durch die Angabe eines (Teil)pfads in der Box *Path globbing pattern* filtern und andererseits entweder nach Name, nach Revisions-Hash, nach der Code-Smell-Anzahl insgesamt oder nach der Code-Smell-Anzahl eines bestimmten Typs sortieren.

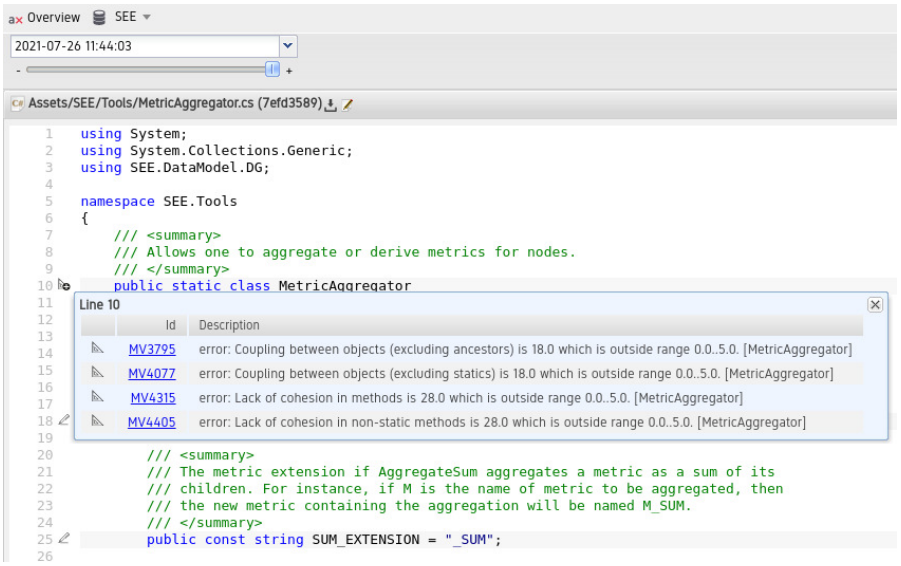


Abbildung 4.2: Der Quellcode von `MetricAggregator.cs` angezeigt im Code-Viewer des Axivion-Dashboards.

Durch das Anklicken einer Datei öffnet sich der Code-Viewer des Dashboards für diese Datei (siehe Abbildung 4.2). Dort lassen sich neben dem Quellcode der Datei Details zu konkreten Code-Smells mit deren Position im Quelltext anzeigen. Dies wird mit kleinen Icons am Zeilenanfang angezeigt — so steht das Stift-Icon in Zeile 18 etwa für eine Stil-Verletzung an dieser Stelle. Durch das Anklicken des Icons erscheinen Details zu den Issues in der Zeile, in der Abbildung wurde dies in Zeile 10 beispielhaft getan. Insofern handelt es sich hier um ein grobes Äquivalent zu den Code-Windows (siehe Abschnitt 2.2) — diese sind zwar nicht Teil der Evaluation, der Code-Viewer des Dashboards jedoch implizit schon, da sich dadurch die Länge der Datei bestimmen lässt, was für die Aufgabe (siehe Abschnitt 4.5) notwendig ist.

Eine alternative Ansicht wäre die Navigation nach Issues statt nach Dateien in der sogenannten *Issue-Counts*-Ansicht: Hier lassen sich alle Code-Smells eines bestimmten Typs anzeigen und nach dessen Attributen (z. B. Quelldatei, Quellzeile, Besitzer des Code-Smells, usw.) sortieren und filtern. Diese Sicht ist jedoch weiter von der zu evaluierenden Funktionalität in SEE (siehe Abschnitt 3.3) entfernt — der *Project Index* kommt der Darstellung der Code-Smells in den Code-Cities am nächsten, denn bei beiden lässt sich mit der Navigation durch den Ordnerbaum des Projekts jeweils die Anzahl an Code-Smells aufge-

1 Hier werden nicht alle Möglichkeiten des *Project Index* angesprochen, z. B. ist der *Export* oder das *List Layout* für die nachfolgende Evaluation irrelevant.

schlüsselt nach dem Typ anzeigen, während ein Anklicken der Datei den Code-Viewer bzw. ein Code-Window für diese Datei öffnet.

Es gibt jedoch auch einige Unterschiede, die bei der Durchführung der Evaluation eine Rolle spielen könnten:

- In SEE ist die Navigation durch Ordnerebenen etwas flüssiger als im Dashboard, da man in Ersterem durch Zoomen der Code-City nahtlos mehrere Ebenen betrachten kann, während man sich in Letzterem manuell durch die Ordner durchklicken muss. Auf der anderen Seite kann die Navigation im Dashboard für Nutzer jedoch gewohnter sein, da dies traditionellen Dateimanagern wie z. B. dem *Windows Explorer* ähnelt.
- Im Dashboard lassen sich die Dateien und Ordner im Gegensatz zu SEE sortieren — nicht nur nach dem Dateinamen, sondern auch nach der Anzahl an Issues, was das Aufspüren z. B. von *Issue-Hotspots* sehr erleichtert. Dabei lässt sich neben der gesammelten Anzahl an Issues auch aufgeschlüsselt nach Typ sortieren. Die Sortierung findet dabei immer pro Ordnerebene statt, solange man sich im *Tree Layout* befindet.

  Issue-Hotspot: *Stelle im Projekt, an der überproportional viele Issues existieren.*
- In SEE lassen sich zusätzliche Metriken visualisieren — hier ist insbesondere die Dateigröße von Relevanz, da wir diese später in der Evaluation nutzen. Im Dashboard ließe sich die Dateigröße nur ermitteln, indem die entsprechende Datei im Code-Viewer geöffnet wird.
- Das Dashboard stellt im Gegensatz zu SEE zusätzlich die Anzahl aller Issues (also aller Typen) summiert dar, es lässt sich ebenfalls danach sortieren.
- In SEE werden die exakten Issue-Zahlen im Gegensatz zum Dashboard erst beim Hovern angezeigt, vorher lässt sich die relative Anzahl an Issues pro Ebene allerdings schon durch visuelle Indikatoren (siehe Abschnitt 3.3) abschätzen.

4.2 ziel und hypothesen

In diesem Experiment ist das Ziel, die in Abschnitt 1.3 genannte Forschungsfrage zu klären, indem das Axivion-Dashboard mit der in Kapitel 3 beschriebenen Implementierung in SEE verglichen wird. In der Forschungsfrage steht die *Eignung* der beiden Formen im Zentrum. Unter diesem Stichwort sollen die folgenden genaueren Aspekte in diesem Experiment gemessen werden — die Hypothese und Nullhypothese jedes Aspekts ist hier ebenfalls aufgeführt:

a) **Korrektheit**: Hierbei nennen wir die Korrektheit beim Bearbeiten der Aufgaben in SEE C_S und in dem Dashboard C_D. Die Korrekt-

heit ist hier definiert als der Anteil korrekter Antworten innerhalb einer Aufgabe.

- *Nullhypothese* H_{a_0}: Die Korrektheit in SEE ist kleiner als oder gleich der Korrektheit im Dashboard: $C_S \leqslant C_D$
- *Alternative Hypothese* H_{a_1}: Die Korrektheit in SEE ist größer als die Korrektheit im Dashboard: $C_S > C_D$

b) **Geschwindigkeit**: Die Bearbeitungszeit für SEE nennen wir t_S, die für das Dashboard t_D.

- *Nullhypothese* H_{b_0}: Die Bearbeitungszeit der Aufgabe ist in SEE größer als oder gleich der Bearbeitungszeit im Dashboard: $t_S \geqslant t_D$
- *Alternative Hypothese* H_{b_1}: Die Bearbeitungszeit der Aufgabe ist in SEE kürzer als die Bearbeitungszeit im Dashboard: $t_S < t_D$

Usability: *Beschreibt, wie gut ein (insbesondere Software-)System benutzt werden kann.*
SUS: *Ein verbreiteter post-study Fragebogen bestehend aus 10 Fragen.*
Post-Study: *Ein Fragebogen, welcher erst nach allen Aufgaben abgefragt wird, also am Ende der Studie.*
ASQ: *Ein post-task Fragebogen bestehend aus 3 Fragen.*
Post-Task: *Ein Fragebogen, welcher nach jeder einzelnen Aufgabe abgefragt wird.*

c) ***Usability***: Hier gibt es zwei Unteraspekte: Den *System Usability Scale* (SUS)-Score als *Post-Study*-Ergebnis und das Ergebnis des *After-Scenario Questionnaire* (ASQ) als *Post-Task*-Ergebnis (siehe Abschnitt 4.4).

i) **SUS**: Den SUS-Score bezeichnen wir für SEE als S_S und für das Dashboard als S_D.

- *Nullhypothese* H_{c_0}: Der SUS-Score ist für SEE niedriger als oder gleich dem für das Dashboard: $S_S \leqslant S_D$
- *Alternative Hypothese* H_{c_1}: Der SUS-Score ist für SEE größer als der für das Dashboard: $S_S > S_D$

ii) **ASQ**: Hier müssen wir erneut in zwei Aspekte teilen, da der ASQ in unserem Fall aus zwei Fragen besteht (siehe Unterabschnitt 4.4.2).

1) Den ASQ-Score für den *Aufwand*[2] bezeichnen wir für SEE als A_{eS} und für das Dashboard als A_{eD}.
   - *Nullhypothese* H_{d_0}: Der ASQ-Score für den Aufwand ist für SEE niedriger als oder gleich dem für das Dashboard: $A_{\alpha S} \leqslant A_{\alpha D}$
   - *Alternative Hypothese* H_{d_1}: Der ASQ-Score für den Aufwand ist für SEE größer als der für das Dashboard: $A_{\alpha S} > A_{\alpha D}$
2) Den ASQ-Score für die *Komplexität* bezeichnen wir für SEE als $A_{\beta S}$ und für das Dashboard als $A_{\beta D}$.
   - *Nullhypothese* H_{e_0}: Der ASQ-Score für die Komplexität ist für SEE niedriger als oder gleich dem für das Dashboard: $A_{\beta S} \leqslant A_{\beta D}$

2 Ein höherer ASQ-Score bedeutet einen niedrigeren Aufwand, daher sind die Hypothesen hier so aufgebaut.

- *Alternative Hypothese* H_{e_1}: Der ASQ-Score für die Komplexität ist für SEE größer als der für das Dashboard: $A_{\beta S} > A_{\beta D}$

Das Experiment soll dabei an drei Partitionen von Personen durchgeführt werden, die alle gemein haben, dass sie etwas mit Software-Entwicklung zu tun haben und somit Teil der Zielgruppe von SEE sind:

1. **Informatik-Studenten**[3]**:** Diese haben weder Erfahrung mit SEE, noch mit dem Dashboard.
2. **SEA-Studenten:** Diese haben Erfahrung mit SEE, aber keine mit dem Dashboard.
3. **Axivion-Mitarbeiter:** Diese haben keine Erfahrung mit SEE, aber mit dem Dashboard.

Die Erfahrung mit der jeweiligen Software wird dabei am Anfang des Fragebogens (siehe Abschnitt 4.4) abgefragt. Theoretisch existiert noch eine vierte Partition an Personen, nämlich solche, die sowohl Erfahrung mit SEE als auch mit dem Dashboard haben — mir war jedoch keine solche Person bekannt, und letzendlich nahm auch keine solche Person an der Evaluation teil (siehe Abschnitt 4.7).

Zusammengefasst soll das Experiment also die Visualisierung von Code-Smells in Code-Cities analysieren, indem SEE und das Axivion-Dashboard in drei Nutzergruppen mit unterschiedlichem Vorwissen — Informatik-Studenten, SEA-Studenten, Axivion-Mitarbeiter — miteinander verglichen werden.

4.3 AUFBAU

Um den Teilnehmenden nicht zu viel zuzumuten und um Ermüdungseffekte gegen Ende des Experiments zu vermeiden, sollte die Teilnahme insgesamt nicht länger als eine Stunde dauern. Dies hat einen Einfluss auf die Wahl des Fragebogens und auch auf die Anzahl der Aufgaben, welche außerdem noch eine gerade Zahl sein muss, denn jeder Teilnehmer soll die Hälfte der Aufgaben in SEE und die andere Hälfte im Axivion-Dashboard bearbeiten, um genügend Daten zu jedem Produkt zu sammeln. Hierbei soll die Reihenfolge bei 50% der Teilnehmer SEE→Dashboard und bei den anderen 50% der Teilnehmer Dashboard→SEE sein, um zu vermeiden, dass die Auswertung durch die Bearbeitungsreihenfolge gestört wird. Die Zuordnung wird dabei randomisiert vorgenommen. Ziel ist es, wie in der Literatur für unseren Fragebogen (siehe Abschnitt 4.4) empfohlen, $n \geqslant 12$ Teilnehmer zu bekommen [SL09, S. 95–96; Lew18, S. 579]. Die Teilnehmer erhalten wir dabei durch *Convenience-Sampling*, wobei nach Möglichkeit jede

Convenience-Sampling: *Eine Auswahlstrategie für Teilnehmer einer Studie, bei der diese nach Einfachheit (statt z. B. per Zufall) gewählt werden.*

3 Dies schließt auch Studenten von Studiengängen wie *Digitale Medien* oder *Wirtschaftsinformatik* mit ein.

Partition (siehe Abschnitt 4.2) eine ungefähr gleiche Anzahl an Personen enthalten sollte.

Das Abrufen der Daten, wie beschrieben in Abschnitt 2.1, wird nicht evaluiert, da es sich hier hauptsächlich um den Abruf von Daten einer extern angebotenen API handelt — würde man hier z. B. die Performance messen, würde man letztendlich nur die Performance der Server, auf denen das Axivion-Dashboard läuft (und eventuell des C#-Clients) testen. Ebenfalls werden die Code-Windows (siehe Abschnitt 2.2) nicht evaluiert, denn hierbei handelt es sich nicht um einen unkonventionellen Ansatz. Dementsprechend soll lediglich die Darstellung der Code-Smells in den Code-Cities (siehe Abschnitt 3.3) evaluiert werden.

Neben der Auswertung mithilfe eines Fragebogens an die Teilnehmer wird auch eine Auswertung der jeweils benötigten Zeit und der Korrektheit beim Lösen der Aufgabe erfolgen. Zuerst gehe ich also auf die Wahl des Fragebogens ein und anschließend auf die Aufgabe selbst.

4.4 FRAGEBOGEN

Das Ziel war es, einen Post-Study- und einen Post-Task-Fragebogen zu verwenden — Ersterer wird nach der gesamten Teilnahme und Letzterer nach jeder einzelnen Aufgabe gezeigt. Wir werden dafür nun jeweils mehrere Fragebögen vergleichen, um den für unsere Zwecke geeignetsten Fragebogen zu finden. Da Post-Task-Fragen eher Informationen zur konkret bearbeiteten Aufgabe und Post-Study-Fragen darüber hinausgehende Aspekte abfragen, wird in der Literatur (so z. B. von Sauro und Lewis [SL09, S. 1617]) empfohlen, beide Fragebogen-Typen zu verwenden. Weiterhin sollen ganz am Anfang des Experiments in einem einleitenden Fragebogen mehrere demographische Fragen gestellt werden.

4.4.1 *Demographischer Fragebogen*

Die Teilnehmenden sollten zu Beginn einen demographischen Fragebogen ausfüllen, wie in der Literatur empfohlen [z. B. Lew18, S. 587] — dies ist auch im Hinblick auf die Verwendung des SUS wichtig, denn z. B. das Alter kann dort einen Einfluss auf das Ergebnis haben [BKM08, S. 585]. In diesem Fragebogen wird Alter, Geschlecht und (Hoch-)Schulabschluss abgefragt. Wichtig ist an dieser Stelle auch noch, dass laut Mclellan, Muddimer und Peres [MMP11] bestehende Erfahrung des zu untersuchenden Systems einen erheblichen Einfluss (in deren konkreten Fall 15–16%) auf den resultierenden SUS-Score haben kann, daher werden in unserer Studie die Teilnehmer nach dem demographischen Teil ebenfalls nach ihrer allgemeinen Erfahrung als Software-Entwickler sowie zur speziellen Erfahrung mit SEE und dem Axivion-Dashboard gefragt. Außerdem fragen wir noch nach der

Erfahrung der Teilnehmer mit 3D-Videospielen, da diese Kenntnis die Bedienung von SEE erleichtern könnte.

4.4.2 *Post-Task-Fragebogen*

Da dieser Fragebogen nach jeder einzelnen Aufgabe abgefragt wird, war hier einer der Anforderungen, nicht mehr als drei Fragen zu haben, um die Konzentration des Teilnehmers nicht zu unterbrechen und die Evaluation nicht zu umfangreich werden zu lassen.

NASA TASK LOAD INDEX (NASA-TLX) Hier handelt es sich um einen von der NASA entwickelten Fragebogen, welcher in sechs Fragen jeweils die *mentale, physische* und *temporale Anstrengung* sowie die *erbrachte Leistung*, den *Aufwand* und die *Frustration* abfragt [übersetzt aus HS88, S. 169]. Während dieser Fragebogen zwar viele Aspekte erfasst und verbreitete Nutzung findet, spricht unser Kriterium der maximalen drei Fragen gegen die Verwendung des NASA Task Load Index.

USABILITY MAGNITUDE ESTIMATION (UME) In diesem erst *RSME*, kurz für „Rating Scale Mental Effort" [ZD85, S. 47], genannten unkonventionellen Fragebogen soll, unter Verwendung der sogenannten *Magnitude Estimation*, die Schwierigkeit einzelner Aufgaben zueinander ins Verhältnis gesetzt werden. Hierzu muss zuerst eine objektive Definition von Usability aufgestellt werden, dann ein Ziel[4] gesetzt werden und anschließend wird der Teilnehmende jeweils nach einer Zahl gefragt, welche die Usability des Ziels einschätzt. Die erste Zahl kann dabei beliebig gewählt werden und bietet die Basis für darauffolgende Aufgaben, um diese zueinander ins Verhältnis zu setzen. Wählt der Teilnehmer z. B. nach der ersten Aufgabe eine beliebige Zahl x und empfindet anschließend die nächste Aufgabe als doppelt so schwer, würde er hierfür die Zahl $2x$ wählen sollen.

Es handelt sich hier zwar um einen interessanten Ansatz der laut seines Autors „höchst effektiv, effizient und zufriedenstellend" [übersetzt aus McG03, S. 695] gegenüber traditionelleren Usability-Metriken sei, jedoch werde der Ansatz laut Sauro und Dumas [SD09, S. 1607–1608] sehr häufig nicht verstanden, weswegen es sich wohl im Vergleich zu einer traditionellen *Likert-Skala* nicht lohne, Usability Magnitude Estimation einzusetzen, daher tun wir dies auch nicht.

Likert-Skala: *Eine Skala, die in der Psychometrie verwendet wird, in welcher eine Antwort auf einer linearen Skala von „Ich stimme zu" bis „Ich stimme nicht zu" gegeben wird.*

SUBJECTIVE MENTAL EFFORT QUESTION (SMEQ) Die Subjective Mental Effort Question ist eine einzelne Frage, welche jedoch statt einer traditionellen siebenstufigen Likert-Skala eine 150-stufige Skala verwendet (siehe Abbildung 4.3). Während diese Frage zwar mehr als zwanzigmal mehr Auswahlmöglichkeiten als eine typische siebenstufige Skala

4 Hier geht es konkret um das zu bewertende Objekt, also z. B. ein Produkt oder eine Aufgabe.

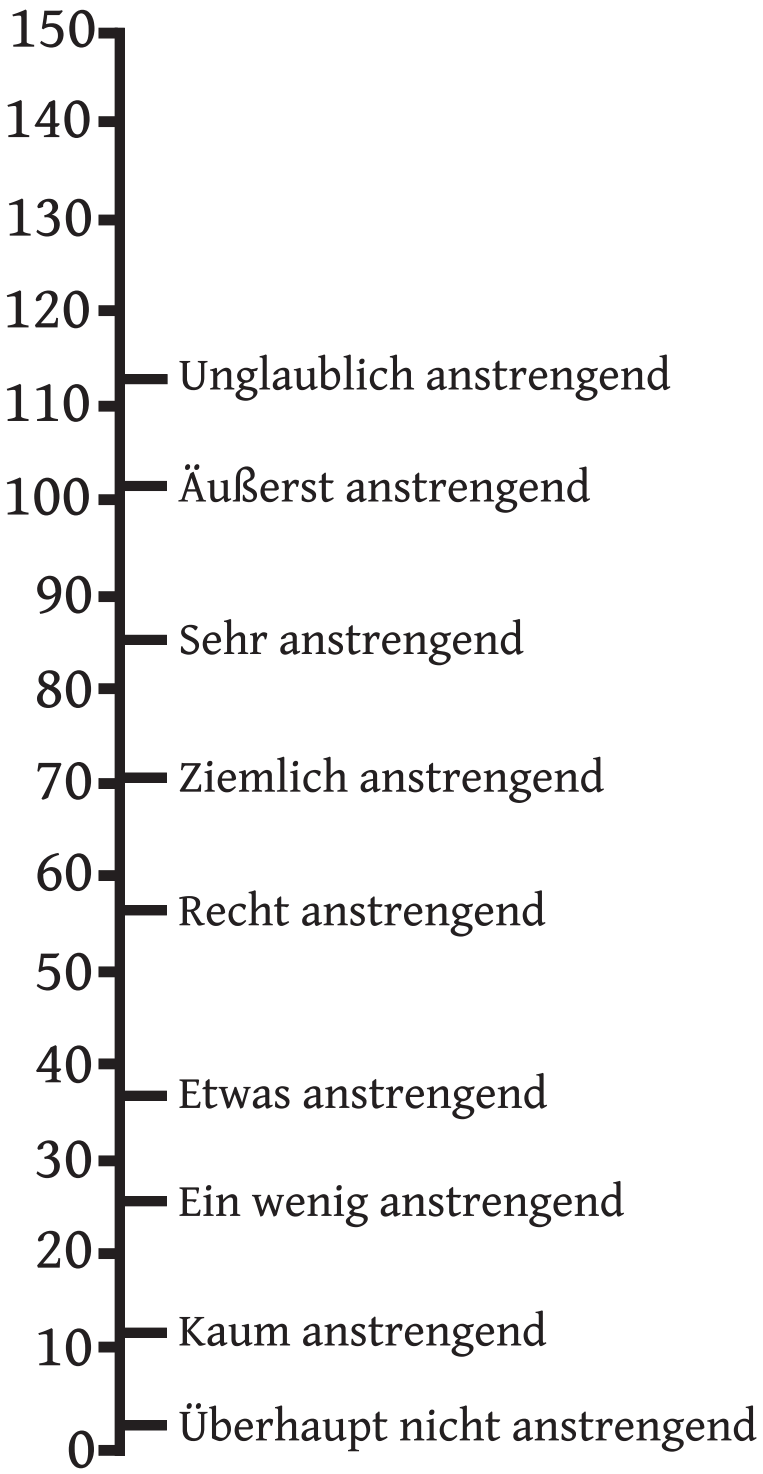


Abbildung 4.3: Der SMEQ-Fragebogen. Grafik wurde basierend auf Zijlstra und Doorn [ZD85, S. 69] erstellt und aus dem Holländischen übersetzt.

bietet, ist es zweifelhaft, ob dies wirklich auch eine Verzwanzigfachung der Messgenauigkeit bewirkt [SD09, S. 1607]. Da Nutzer gewöhnliche Likert-Skalen wohl eher gewohnt sind und außerdem das Problem besteht, dass sich die Frage nicht ohne weiteres in das hier verwendete Online-Fragebogen-Tool (siehe Unterabschnitt 4.6.1) integrieren lässt, verwenden wir die Subjective Mental Effort Question nicht.

SINGLE EASE QUESTION (SEQ) Wie der Name vermuten lässt, ist die Single Ease Question eine einzelne Frage, welche die Teilnehmenden unter Verwendung einer einfachen siebenstufigen Likert-Skala abfragt, wie schwer sie die Aufgabe fanden. Dieser Post-Task-Fragebogen wirkte aufgrund seiner Einfachheit und Kürze sehr attraktiv, da dies nur eine minimale Unterbrechung beim Bearbeiten der Aufgaben darstellen würde. Die SEQ wurde daher in der Pilotstudie (siehe Unterabschnitt 4.6.2) verwendet — es kam allerdings das Feedback, dass manche Aufgaben zwar kognitiv kaum anstrengend, aber sehr mühselig seien. Dies wäre durch eine einzige Frage nicht erfassbar gewesen, daher wurde die Single Ease Question in der finalen Studie nicht verwendet.

AFTER-SCENARIO QUESTIONNAIRE (ASQ) Hier handelt es sich um einen aus drei Fragen bestehenden Post-Task-Fragebogen. In den drei Fragen wird jeweils die Zufriedenheit mit der Einfachheit, der benötigten Zeit und den verfügbaren Support-Informationen abgefragt [Lew91]. Dies eignet sich also hervorragend, um die Nachteile des oben besprochenen SEQs auszugleichen, da jetzt zwischen dem kognitiven und temporalen Aufwand unterschieden wird. Die dritte Frage bezieht sich auf Support-Informationen wie Online-Dokumentationen oder Hilfenachrichten — da wir beides weder in SEE noch im AXIVION-Dashboard anbieten oder messen wollen, lassen wir diese dritte Frage (wie auch z. B. Tedesco und Tullis [TT06, S. 11] es tun) aus.

Da der After-Scenario Questionnaire ursprünglich auf Englisch verfasst wurde, benötigen wir noch eine deutsche Version. Hierfür nehmen wir die von Funk und Roegele [FR20, S. 32] im Rahmen einer App-Evaluation angefertigte Übersetzung.

4.4.3 *Post-Study-Fragebogen*

Bei dem Post-Study-Fragebogen geht es hauptsächlich darum, detaillierte Informationen zur Usability des jeweiligen Gesamtsystems zu sammeln, also unabhängig von einzelnen Aufgaben. Hier sollten es möglichst auch nicht zu viele Fragen werden, um die Teilnehmenden nicht zu ermüden — diese müssen den Fragebogen zweimal ausfüllen, einmal nach der Dashboard- und einmal nach der SEE-Aufgabe. Wir legen uns hier auf eine Obergrenze von zwanzig Fragen fest. Ebenfalls ist es wichtig, dass eine validierte deutsche Fassung des Fragebogens existiert, da eine Verwendung des englischen Fragebogens Probleme

für Nicht-Muttersprachler mit sich bringen kann [Fin06]. Wie so eine Validierung aussehen kann, wird bei der Erläuterung des SUS erklärt.

SOFTWARE USABILITY MEASUREMENT INVENTORY (SUMI) Dieser Fragebogen besteht aus 50 Fragen mit jeweils drei Antwortmöglichkeiten („Stimme zu", „Weiß nicht" und „Stimme nicht zu"), welche jeweils auf fünf Skalen *Effizienz, Affekt, Hilfe und Unterstützung, Kontrollierbarkeit* und *Erlernbarkeit* misst [übersetzt aus Kir94, S. 10]. Es existiert zwar eine „offizielle" deutsche Version [Kir94, S. 16], jedoch bietet der Autor diesen Fragebogen nur unter hohen Lizenzgebühren[5] an [Kir]. Da der Fragebogen zusätzlich noch unsere obere Grenze der Anzahl an Fragen deutlich überschreitet, verzichten wir auf die Verwendung des Software Usability Measurement Inventory.

QUESTIONNAIRE FOR USER INTERACTION SATISFACTION (QUIS) Dieser von Chin, Diehl und Norman [CDN88] an der *University of Maryland* entwickelte Post-Study-Fragebogen existiert in zwei Versionen: Einerseits existiert eine aus 41 Fragen bestehende Kurzversion und andererseits eine aus 122 Fragen bestehende Langversion. Es können hier zwar sehr detaillierte Informationen zur Usability abgefragt werden, jedoch sprengt selbst die Kurzversion schon deutlich den von uns gestellten Rahmen — hinzu kommt noch, dass an die erwähnte Universität eine Lizenzgebühr zur Verwendung des Fragebogens gezahlt werden muss, dementsprechend verwenden wir den Questionnaire for User Interaction Satisfaction nicht.

POST-STUDY SYSTEM USABILITY QUESTIONNAIRE (PSSUQ) Bei diesem Fragebogen gibt es insgesamt sechzehn (in der ursprünglichen Version waren es achtzehn [Lew92]) jeweils auf einer siebenstufigen Likert-Skala zu beantwortende Fragen, welche drei Faktoren messen: *Nützlichkeit des Systems, Informationsqualität* und *Interface-Qualität* [Lew92, S. 1260]. Dieser Fragebogen erfüllt unsere Anforderungen an die Fragenanzahl und erwies sich in mehreren Studien als gut generalisierbar [Lew92; Lew02; FL05]. Leider existiert keine validierte deutsche Fassung des Post-Study System Usability Questionnaire, weswegen dieser nicht verwendet wurde.

SYSTEM USABILITY SCALE (SUS) Schließlich schauen wir uns noch diesen aus zehn Fragen bestehenden Post-Study-Fragebogen an, welcher als eine „quick and dirty Usability scale" [Bro96, S. 189] erstellt wurde. Dieser hat gleich mehrere große Vorteile, weswegen ich mich letztendlich dafür entschieden habe:

5 Es existiert zwar auch die Möglichkeit einer kostenfreien Nutzung zu akademischen Zwecken, die Anmeldung hierzu ist jedoch im Vergleich zu den anderen Fragebögen, welche direkt verwendet werden können, recht aufwändig.

- Er ist sehr verbreitet: Sauro und Lewis [SL09, S. 1615] finden z. B. bei 40 untersuchten Post-Study-Fragebögen heraus, dass der SUS mit 17 von 40 Verwendungen (42,5%) am häufigsten genutzt wurde. Lewis [Lew18, S. 577] benennt als weitere Quelle die 5.664 Zitierungen[6], die das Original-Paper von Brooke [Bro96] (welches den Fragebogen einführt) erhalten hat.
- Er eignet sich gut zum Vergleich von zwei Systemen [PPP13], laut Bangor, Kortum und Miller [BKM08, S. 590–591] insbesondere auch zur Gegenüberstellung zweier Interface-Ansätze.
- Er ist im Hinblick auf Validität, Sensitivität und Verlässlichkeit generell robust und vielfältig einsetzbar, um die Usability eines Software-Systems zu ermitteln [PPP13; BKM08; Lew18].
- Es werden keine Lizenzgebühren benötigt [Bro96, S. 194].
- Auch bei dem SUS ist es wichtig, eine validierte deutsche Übersetzung zu haben. Glücklicherweise existiert eine von dem deutschen Unternehmen SAP frei verwendbare Veröffentlichung einer validierten deutschen Übersetzung des SUS [RR]. Die Validierung sah in diesem Fall so aus, dass die deutschen Versionen von unabhängigen Muttersprachlern aus den USA und Großbritannien zurück auf Englisch übersetzt wurde, um dann zu überprüfen, ob die Fragen die gleiche Formulierung wie der ursprüngliche SUS erhalten.
- Entgegen der ursprünglichen Annahme von z. B. Brooke [Bro96] und Bangor, Kortum und Miller [BKM08], dass der SUS nur ein eindimensionales Ergebnis liefert, hatte eine Faktoranalyse von Lewis und Sauro [LS09] das Ergebnis, dass sich durch den SUS tatsächlich zwei Faktoren analysieren lassen, die die Autoren *Usability* und *Learnability* nennen. Dementsprechend können wir in der späteren Analyse (siehe Abschnitt 4.7) die Usability des Systems mithilfe dieser zwei Faktoren noch etwas feingranularer betrachten.

Also wird für die Studie als Post-Study-Fragebogen die von Rummel und Ruegenhagen [RR] angefertigte deutsche Übersetzung des System Usability Scale eingesetzt. Hierdurch umgehen wir das von Bangor, Kortum und Miller [BKM08] und Finstad [Fin06] angesprochene Problem, dass Frage 8 („I found the system very cumbersome to use“) insbesondere auch von Personen, deren Muttersprache nicht Englisch ist, missverstanden wird.

6 Stand: 13.3.2018

4.5 AUFGABENSTELLUNG

Kommen wir nun zu den eigentlichen Aufgaben, die die Teilnehmenden in dieser Studie bearbeiten müssen. Um passende Aufgaben zu finden, wurden mehrere Axivion-Mitarbeiter nach Anwendungsszenarien des hier zu untersuchenden *Project Index* (siehe Abschnitt 4.1) gefragt. Die Antworten können in `DashboardUseCases.txt` nachgelesen werden. Basierend auf diesen Informationen wurde folgendes repräsentatives Szenario von mir erstellt, was als Basis für die Aufgaben dienen soll:

> Ein Projektmanager informiert einen Software-Architekten, dass dessen Projekt laut einer internen Untersuchung zu viele *Gottklassen* hat. Bevor größere neue Features implementiert werden, soll der Architekt nun die bestehenden Gottklassen identifizieren und in kleinere Klassen aufteilen, damit das Projekt eine übersichtlichere Struktur bekommt. Zum Erkennen der Gottklassen verwendet dieser einfach erst einmal die grobe Metrik der Anzahl an Codezeilen. Wenn er schon dabei ist, möchte er auch gleich versuchen, bei diesen Klassen die Ursachen bestimmter Typen von Code-Smells zu entfernen; z. B. lassen sich Stil-Verletzungen recht schnell beheben. Um abzuschätzen, wieviel Zeit dies in Anspruch nehmen wird, verwendet er den *Project Index* des Axivion-Dashboard bzw. die Darstellung der Code-Smells in SEE.

Gottklasse: *Eine Klasse, welche zu groß ist oder für zu viele Dinge zuständig ist, was wiederum die Lesbarkeit bzw. die Übersicht über den Quellcode mindert.*

Wir kommen hier also auf Aufgaben, in denen für Gottklassen Informationen zu bestimmten Code-Smells herausgefunden werden sollen. In den ersten beiden Aufgaben wird es darum gehen, jeweils Gottklassen innerhalb eines bestimmten Moduls[7] zu identifizieren und dann die Anzahl bestimmter Code-Smell-Typen zu bestimmen. In der dritten Aufgabe werden die Gottklassen vorgegeben[8] und es muss jeweils der häufigste Code-Smell-Typ bestimmt werden.

Als Projekt, das analysiert werden soll, verwenden wir den Quellcode von SEE selbst — hier handelt es sich um ein reales und repräsentatives Projekt, das bereits sowohl im Dashboard als auch in SEE verfügbar ist und somit keinen zusätzlichen Einrichtungsaufwand verursacht. Konkret lauten die drei Aufgaben (dabei sind X, Y, Z konkrete Ordner bzw. Namespaces in der Implementierung von SEE):

1. Finde im Ordner X die fünf größten Dateien. Wie viele Architektur-Verletzungen haben diese jeweils?

7 Für das Dashboard dienen die Ordner als *Module* und die Dateien als zugehörige Elemente, während es bei SEE die C#-Namespaces und -Klassen sind, da dies jeweils auch die dargestellten Objekte sind.

8 Dies wurde aus Zeitgründen getan, da eines der Ziele war, dass Teilnahmen nicht länger als eine Stunde dauern. Die Aufgabe sollte aber dennoch repräsentativ für ein realistisches Szenario sein — der Architekt hätte schließlich auch eine Liste an Gottklassen vom Projektmanager bekommen können.

2. Finde im Ordner Y die fünf größten Dateien. Wie viele Stil-Verletzungen/Metrik-Verletzungen[9] haben diese jeweils?

3. Finde für die fünf gegebenen größten Dateien des Ordners Z jeweils den häufigsten Code-Smell-Typ.

Wir betrachten hier nur *Architektur-Verletzungen*, *Stil-Verletzungen* und *Metrik-Verletzungen*, da Klone, Zyklen und toter Code zu selten im Quellcode von SEE vorkamen. Wir trennen dabei die Architektur-Verletzungen in Aufgabe 1 von den Stil-Verletzungen bzw. Metrik-Verletzungen, da die letzteren beiden in ihrer Anzahl deutlich stärker miteinander korrelieren ($r \approx 0,66$) als Architektur-Verletzungen und Stil-Verletzungen ($r \approx 0,24$) oder Architektur-Verletzungen und Metrik-Verletzungen ($r \approx 0,13$)[10].

Pearson-Korrelationskoeffizient (r): *Hiermit kann eine lineare Korrelation zwischen zwei Variablen X und Y gemessen werden:*

$$r = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \cdot \sigma_Y}$$

Cov *ist hierbei die Kovarianz und* σ *die Standardabweichung.*

4.5.1 *Wahl der Ordner*

Nun müssen natürlich noch die Ordner X, Y und Z für die Aufgaben festgelegt werden. Da wir diese drei Aufgaben zweimal durchführen (einmal mit dem Dashboard und einmal mit SEE) brauchen wir insgesamt zwei Belegungen für jede dieser Variablen, also sechs verschiedene Ordner. Diese bezeichnen wir jeweils als X_1 bzw. X_2, Y_1 bzw. Y_2 und Z_1 bzw. Z_2. Wir möchten nur Ordner von SEE verwenden, die direkt unter dem Wurzelverzeichnis liegen, damit die Navigation in SEE bzw. dem Dashboard nicht zu kompliziert wird. Von diesen höchsten Modulen möchten wir möglichst die größten[11] verwenden, da das Identifizieren der Gottklassen für kleinere Module schnell trivial wird und diese schlechter in SEE sichtbar sind. Die sechs größten Ordner auf der höchsten Hierarchiestufe des Quellcodes von SEE sind die folgenden:

1. `Game` (39 Elemente)
2. `Utils` (37 Elemente)
3. `GameObjects` (18 Elemente)
4. `Controls` (13 Elemente)
5. `Net` (9 Elemente)
6. `Layout` (8 Elemente)

Wir definieren uns die Menge dieser sechs Ordner, repräsentiert als Anzahl der enthaltenen Elemente $S = \{39, 37, 18, 13, 9, 8\}$. Diese

9 Hiermit ist gemeint, dass in einer Aufgabe Stil-Verletzungen und in der anderen Aufgabe Metrik-Verletzungen gefunden werden sollen.

10 Die Korrelation wurde hier basierend auf den Vorkommnissen im Quellcode von SEE berechnet. Das verwendete Skript zum Berechnen dieser Werte ist im Anhang unter dem Namen `findCorrelation.py` zu finden.

11 Wobei mit den „größten“ Modulen hier diejenigen gemeint sind, die die meisten Dateien als direkte Kinder (also nicht in Unterordnern) enthalten.

sechs Ordner wollen wir nun in zwei Partitionen teilen, die jeweils drei Elemente enthalten — so haben wir drei Ordner für den SEE-Teil und drei Ordner für den Dashboard-Teil[12]. Wir suchen nun die zweielementige Partition von S, deren jeweilige Elemente aufsummiert die geringste Differenz zueinander haben. Dies tun wir, um den SEE- und Dashboard-Teil möglichst einheitlich zu halten, da mit einer größeren Anzahl an Elementen auch eine längere Bearbeitungszeit einhergehen würde. Unter Verwendung der Abbildung $f : \mathfrak{S}_S \longrightarrow \mathbb{N}_0$ und dessen Abbildungsvorschrift

Symmetrische Gruppe ($\mathfrak{S}$): *Die Gruppe aller Permutationen einer Menge X, wobei diese Menge in unserer Notation als Index geschrieben wird, also z. B. $\mathfrak{S}_X$.*

Argumente der Maxima (argmax): *Stellen einer Funktion, an der die Funktionswerte maximiert werden.*

$$f : (p_i)_{i=1}^{n=6} \longmapsto \left| \sum_{i=1}^{3} p_i - \sum_{i=4}^{6} p_i \right|$$

berechnen wir also $\operatorname{argmax}_{\mathfrak{S}_S} f$. Als Ergebnis erhalten wir die Permutation $p = (39, 13, 9, 37, 18, 8)$, wobei wir hiervon dann als erste Partition die ersten drei Elemente $\{39, 13, 9\}$ und als zweite Partition die letzten drei Elemente $\{37, 18, 8\}$ verwenden. Dementsprechend verwenden wir für den ersten Lauf an Aufgaben die Partition {`Game`, `Controls`, `Net`} und für den zweiten Lauf {`Utils`, `GameObjects`, `Layout`}.

4.5.2 *Zuweisung der Ordner*

Nun bleibt noch zu klären, welche Aufgabe welche beiden Ordner (einen für den ersten und einen für den zweiten Lauf) zugewiesen bekommt. Um unsere Einschränkung einzuhalten, die Teilnahme nicht länger als eine Stunde werden zu lassen, weisen wir für die dritte Aufgabe `Game` und `Utils` zu, da dies die mit einigem Abstand größten Ordner sind — die Bearbeitungszeit der dritten Aufgabe ist deutlich weniger als die ersten beiden an die Anzahl der Elemente gekoppelt, da die größten Dateien hier vorgegeben sind.

Die meisten Architektur-Verletzungen finden sich in absteigender Reihenfolge in folgenden Ordnern:

1. ~~`Game`~~: Ist bereits vergeben.
2. `Net`: Frei, verwenden wir also für Aufgabe 1.
3. ~~`Controls`~~: Ist in der gleichen Partition wie `Net`.
4. `GameObjects`: Frei, verwenden wir also für Aufgabe 1.

Wir verwenden die gleiche Strategie für Aufgabe 2. Die meisten Metrik-Verletzungen finden sich in absteigender Reihenfolge in ~~`Game`~~, `Layout`, `Controls`, ~~`Net`~~ und die meisten Stil-Verletzungen in ~~`Game`~~, `Layout`, ~~`GameObjects`~~, ~~`Utils`~~ — dementsprechend sind hier `Controls`

12 Es ist natürlich nicht so, dass dann drei Ordner *nur* in SEE bzw. dem Dashboard vorkommen, die Studie ist schließlich so aufgebaut, dass 50% der Teilnehmer zuerst SEE und die anderen 50% zuerst das Dashboard verwenden — beide haben aber die gleiche Reihenfolge der Aufgaben und somit auch der Ordner.

für Metrik-Verletzungen und `Layout` für Stil-Verletzungen am besten geeignet.

Zusammengefasst: Im ersten Durchlauf lautet die Reihenfolge der Ordner `GameObjects, Layout, Utils` (mit Stil-Verletzungen) und im zweiten Durchlauf `Net, Controls, Game` (mit Metrik-Verletzungen).

4.6 UMSETZUNG

In diesem Teil möchten wir noch einige relevante Details zur konkreten Umsetzung bzw. Durchführung dieser Studie benennen. Zuerst gehen wir dazu auf die von mir verwendete Software ein, beschreiben dann die durchgeführte Pilotstudie und berichten abschließend von der Durchführung.

4.6.1 *Online-Tool*

Die wichtigsten Anforderungen an die Software zur Durchführung der Studie waren folgende:

- Die Studie soll zeitunabhängig und ohne Terminvergabe stattfinden können, das heißt die Teilnehmenden können jederzeit die Studie beginnen, ohne vorher mit mir Kontakt aufnehmen zu müssen. Dazu soll die Studie online stattfinden. Dies war wegen der COVID-19-Pandemie nötig und erlaubt außerdem eine leichte Verteilung der Studie und sorgt so wahrscheinlich für mehr Teilnehmer, da auch keine Terminplanung notwendig ist.
- Mit SEE als unausweichlicher Ausnahme soll die Studie komplett im Browser stattfinden können, um die Durchführung möglichst einfach zu halten.
- Der gewählte Fragebogen (siehe Abschnitt 4.4) soll in dem Online-Tool umsetzbar sein.
- Es soll die Möglichkeit geben, die Zeit zwischen Antworten zu speichern. So können wir eine unserer untersuchten abhängigen Variablen, die benötigte Zeit, messen.
- Die Verwendung sollte komplett kostenfrei sein.

Nach einem Vergleich mehrerer Umfrage-Tools wie z. B. *Google Forms*[13] oder *Survey Monkey*[14] erschien *KoBoToolbox*[15] das am ehesten geeignete zu sein.

KoBoToolbox erfüllt all unsere Anforderungen und erlaubt ein sehr flexibles Erstellen von Fragebögen (der Web-Editor für den Fragebogen kann in Abbildung 4.4 gesehen werden), daher habe ich mich

13 https://www.google.com/forms/about/ (letzter Abruf: 22.08.2021)
14 https://www.surveymonkey.com/ (letzter Abruf: 22.08.2021)
15 https://www.kobotoolbox.org/ (letzter Abruf: 22.08.2021)

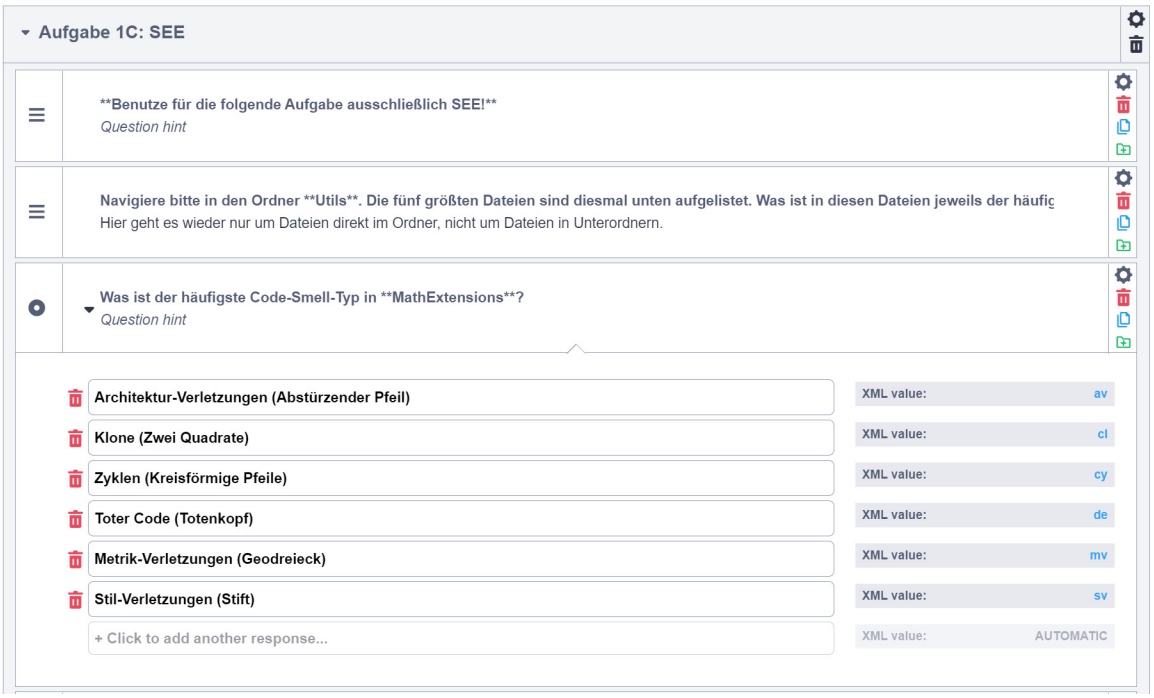


Abbildung 4.4: Der Fragebogeneditor aus KoBoToolbox. Manche Funktionen ließen sich jedoch nur durch das Bearbeiten der XLSForm-CSV-Datei nutzen.

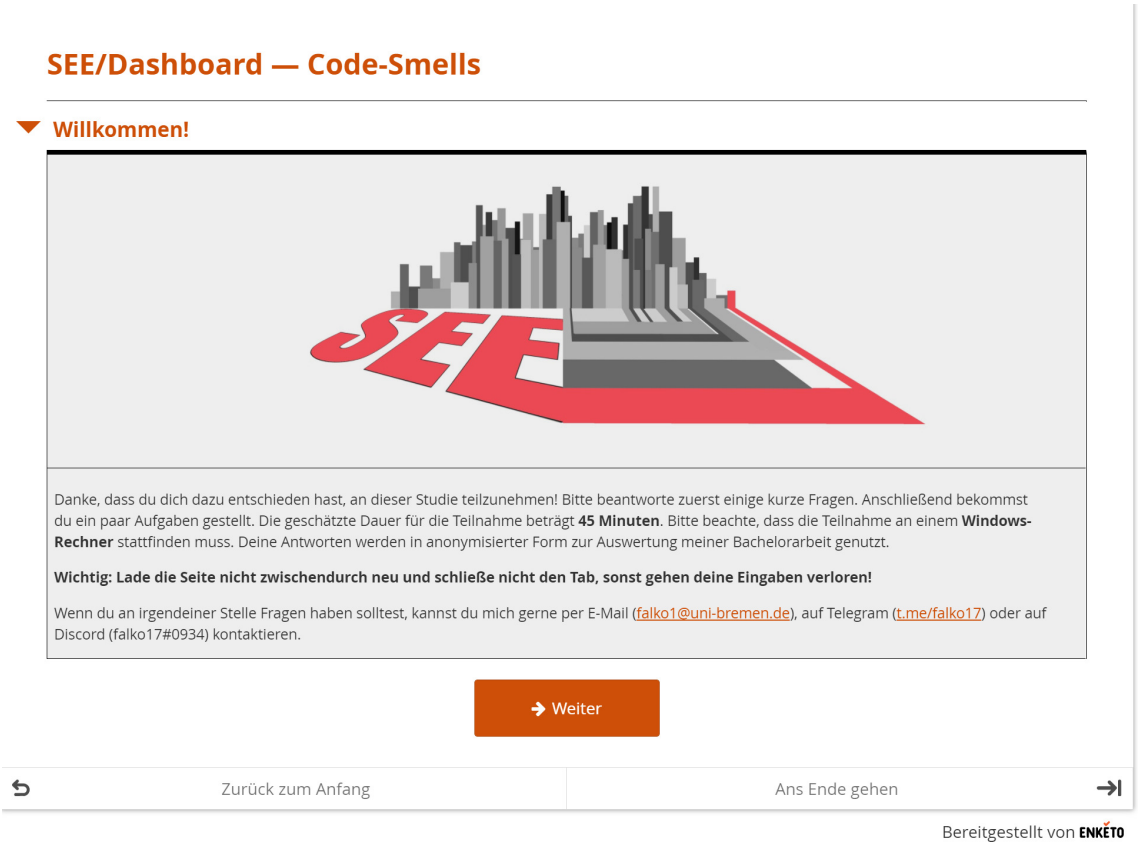


Abbildung 4.5: Die Startseite des Fragebogens der Evaluation.

dafür entschieden. Der dort erstellte Fragebogen existiert in zwei Versionen — einer mit der SEE-Aufgabe zuerst, der andere mit der Dashboard-Aufgabe zuerst — und kann im XLSForm-Format in der Datei `Common-Eval.xlsx` gefunden werden.

Die Startseite dieses Online-Fragebogens ist in Abbildung 4.5 abgebildet. Er hat grob den folgenden Aufbau und Inhalt:

1. Einleitung
2. Demographische Fragen
   a) „Bitte gib dein Geschlecht an.“
   b) „Bitte gib dein Alter an.“
   c) „Was ist dein höchster Schul- oder Hochschulabschluss?“
3. Spezifische Erfahrung
   a) „Wie lange programmierst du schon?“
   b) „Wie lange hast du schon für größere Softwareprojekte (z. B. in einer Firma oder Open-Source-Projekte) programmiert?“
   c) „Kennst du **SEE**?“
   d) „Kennst du das **AXIVION-Dashboard**?“
   e) „Hast du Erfahrung mit 3D-Videospielen für Desktop-PCs?“
4. Download-Link[16] und Erklärung[17] zu SEE bzw. dem Dashboard
   - Bevor die Studie weitergeführt werden konnte, musste eine Frage zu SEE bzw. dem Dashboard beantwortet werden, um sicherzustellen, dass der Teilnehmer die Erklärung verstanden hat.
5. Aufgabe 1: „Navigiere in den Ordner X. Was sind die fünf größten Dateien und wie viele **Architektur-Verletzungen** gibt es dort jeweils?“
   - ASQ:
     – „Ich bin zufrieden mit der Zeit, die es gedauert hat, diese Aufgabe zu lösen.“
     – „Insgesamt bin ich zufrieden, wie leicht diese Aufgabe zu lösen war. (‚Leicht‘ in Bezug auf die Komplexität bzw. den kognitiven Anspruch der Aufgabe, nicht wie mühselig die Aufgabe war.)“

16 Die Version von SEE, die die Teilnehmenden bekommen haben, kann im Anhang unter `SEE.zip` gefunden werden.

17 Der Erklärung lag außerdem ein Video bei, welches die Steuerung in Kombination mit den gedrückten Tasten zeigt: `https://youtu.be/dGY8RFn5WvA` (letzter Abruf: 15.09.2021)

6. Aufgabe 2: „Navigiere in den Ordner Y. Was sind die fünf größten Dateien und wie viele **Stil-Verletzungen/Metrik-Verletzungen** gibt es dort jeweils?“
    - ASQ (s.o.)
7. Aufgabe 3: „Navigiere in den Ordner Z. Die fünf größten Dateien sind diesmal unten aufgelistet. Was ist in diesen Dateien jeweils der häufigste Code-Smell-Typ?“
    - ASQ (s.o.)
8. SUS-Fragebogen für SEE bzw. das Dashboard
9. Wiederholung von 4 bis 8, aber diesmal mit dem zweiten Lauf an Ordnern und dem anderem System
10. „Gibt es ansonsten noch etwas, das du zu deiner Teilnahme oder allgemein anmerken möchtest?“

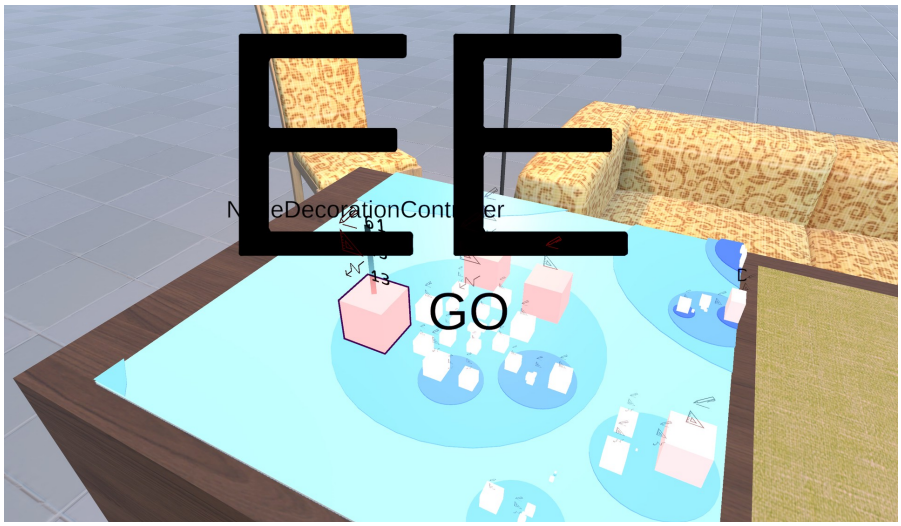
Vor jeder Aufgabe gab es einen Button mit der Beschriftung „Sobald du bereit bist, wähle unten ‚OK‘, ab dann wird die Zeit gestoppt. *Bitte beachte, dass ab diesem Zeitpunkt die Zeit gemessen wird, bis du die Aufgabe beendest!* Bitte mach dann erst eine Pause, nachdem du die Aufgabe abgeschlossen hast.“ Es wird bei jeder Teilnahme die Zeit eingetragen, zu dem die Teilnehmenden diesen Button betätigt haben. Außerdem wird für jedes Eingabefeld der Aufgaben der Zeitpunkt eingetragen, zu dem das entsprechende Feld zuletzt bearbeitet wurde. Diese Zeiten nehmen wir dann als Basis für die Auswertung. Nach dem Abschicken des Fragebogens können die Daten in anonymisierter Form verwendet werden.

4.6.2 *Pilotstudie*

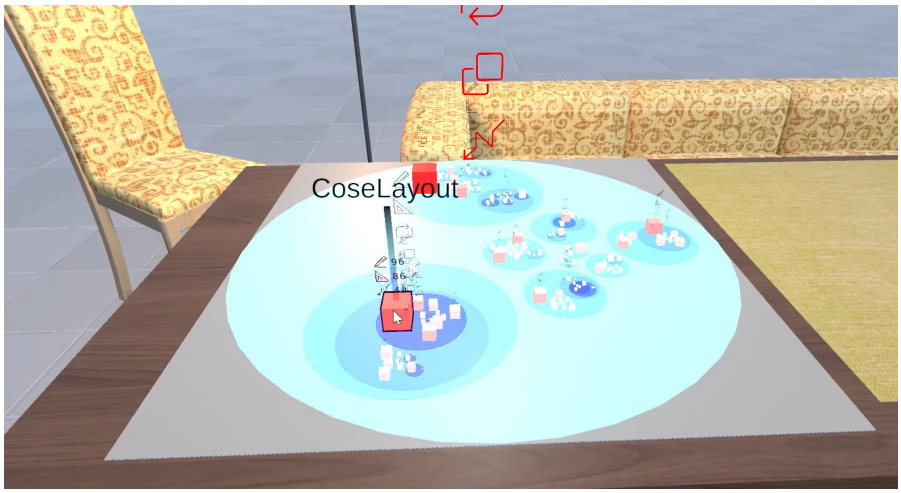
Zum Testen der Studie wurde diese an vier verschiedenen Personen verschickt, die aufgetretene Fehler oder Verbesserungsvorschläge gemeldet haben. Die gesammelten Daten dieser vier Personen wurden nicht für den Datensatz verwendet. Folgende Änderungen wurden aufgrund der Pilotstudie durchgeführt:

- Anstatt die Beschriftungen der Knoten dauerhaft darzustellen, werden diese nun erst beim Bewegen der Maus auf die entsprechenden Knoten darüber angezeigt (in Abbildung 4.6 ist ein Vergleich dargestellt).
- Wird die Taste `X` gedrückt gehalten, werden die Beschriftungen überhaupt nicht eingeblendet. So wird verhindert, dass die Beschriftungen in bestimmten Situationen dabei stören, die Zahlen an den Erosions-Icons abzulesen.

(a) Labels, die konstant in der Mitte der Knoten eingeblendet werden.



(b) Konstante Labels sorgen z. B. beim Reinzoomen für Erkennungsprobleme.



(c) Labels, die erst beim Hovern eingeblendet werden, lenken weniger ab.

Abbildung 4.6: Konstant eingeblendete Labels im Vergleich zu Labels, die beim Hovern eingeblendet werden.

- Es wurde eine Anmerkung hinzugefügt, dass über die Taste `R` die Stadt auf ihren ursprünglichen Ort und ihre ursprüngliche Größe zurückgesetzt wird.
- Für die Post-Task-Frage wurde von SEQ auf ASQ gewechselt, um kognitiven von temporalem Aufwand trennen zu können (siehe Abschnitt 4.4).

Außerdem wurde dadurch ermittelt, dass die Teilnahmedauer der Studie ca. 45 Minuten beträgt und somit unsere Anforderung erfüllt. Diese Information wurde der Einleitung der Studie hinzugefügt, um den Teilnehmenden ein besseres Bild von der benötigten Zeit zu geben.

4.6.3 *Durchführung*

Zur Durchführung der Studie wurde ein Link an potenzielle Teilnehmende verteilt, der alternierend auf die Dashboard→SEE- und auf die SEE→Dashboard-Version des Fragebogens weiterleitet. Die Studie ist dort weiterhin erreichbar[18], da diese Bachelorarbeit jedoch schon verfasst wurde, werden weitere Teilnahmen natürlich nicht in den Datensatz der Auswertung einfließen. Der Link wurde über zwei Wochen hinweg über mehrere Kanäle verteilt, unter anderem an AXIVION-Mitarbeiter, Studenten des SEA-Bachelorprojekts und diverse andere Studenten aus informatiknahen Studiengängen (hauptsächlich der Universität Bremen) in verschiedenen Semestern. Die Zugehörigkeit an einzelne Personengruppen wurde nicht speziell gespeichert, da die relevanten Informationen bereits über die Frage nach der Erfahrung mit SEE und dem Dashboard extrahiert werden können.

Letztendlich konnten insgesamt $n = 20$ Teilnehmende für die Studie gewonnen werden, mit 10 Personen je Reihenfolge (SEE zuerst, Dashboard zuerst). Deren Daten wollen wir im folgenden Abschnitt auswerten.

4.7 AUSWERTUNG

Durch die Verwendung des angehänten R-Skripts `calc_results.R` können die Ergebnisse dieses Abschnitts, inklusive aller Diagramme, direkt reproduziert werden.

18 Dashboard→SEE: https://ee.kobotoolbox.org/single/fbf29547bee96cf545f3eeb451d58ca9
SEE→Dashboard: https://ee.kobotoolbox.org/single/0a570c402b59f04af9b3cee8906ed301
Zu beachten ist außerdem, dass sich das Projekt selbst und die Metrikzahlen mittlerweile geändert haben könnten. Um an den Tutorialfragen vorbeizukommen: Die Datei mit den meisten Stil-Verletzungen in `Assets/SEE/Tools` lautet `ReflexionAnalysis.cs` und hat 2088 davon. Im Ordner `CameraPaths` ist die größte Datei `CameraPath`, diese hat 60 Stil-Verletzungen.

Wie zuvor beschrieben, hat die Hälfte der Probanden die ersten 50% der Aufgaben mit SEE gelöst und anschließend das Dashboard verwendet, während die andere Hälfte der Probanden die umgekehrte Reihenfolge hatte — die erste Gruppe (SEE→Dashboard) nennen wir *Gruppe 1* und die zweite (Dashboard→SEE) nennen wir *Gruppe 2*. Die Reihenfolge der Aufgaben selbst ist jedoch bei beiden gleich, was zur Folge hat, dass wir hier jeweils die gemessenen Daten aus Gruppe 1 mit denen aus Gruppe 2 vergleichen, da wir so gleichzeitig auch immer das Dashboard mit SEE vergleichen.

An mehreren Stellen verwende ich unter Nutzung einer speziellen R-Bibliothek von Patil [Pat21] sogenannte *Violin-Plots*, z. B. in Abbildung 4.8: Dort sieht man die Verteilung des Alters dargestellt als eine Art Wölbung, aufgeteilt auf die beiden Gruppen. Die einzelnen Werte sind dort als rote bzw. blaue Punkte abzulesen, während der Median jeweils als größerer grüner Punkt eingetragen ist. Die Punkte bekommen einen kleinen, zufälligen Versatz auf der x-Achse, da sonst die Überlagerung einzelner Punkte die Unterscheidung dieser sehr erschweren würde. Ausreißer werden in diesem Diagramm pro Gruppe durch eine von Tukey [Tuk77] entwickelte Regel speziell als schwarze Punkte markiert, deren genauer Wert wird, wie auch der Median, direkt in das Diagramm integriert.

Um die Unterschiede zwischen den beiden Gruppen zu messen, verwende ich öfters den *Mann-Whitney-U-Test*. Im Gegensatz zu z. B. dem *t-Test* müssen die Populationen hierfür keine Normalverteilung besitzen, sondern müssen lediglich aus Populationen mit der gleichen Verteilung stammen. Dies ist in unserem Fall wegen der randomisierten Verteilung in die Gruppen gegeben (siehe Unterabschnitt 4.6.3). Ein weiterer Vorteil des Mann-Whitney-U-Test ist, dass dieser keine Intervallskalen voraussetzt, sondern nur Ordinalskalen erfordert — dies ist z. B. bei der Analyse des demographischen Fragebogens in Unterabschnitt 4.7.1 sinnvoll, wo etwa der höchste Abschluss abgefragt wird. Außerdem spielen wegen der rangbasierten Analyse dieses Tests Ausreißer eine geringere Rolle, in den Violin-Plots markieren wir diese aber sicherheitshalber dennoch. Wir legen uns hier im Vorfeld auf ein Signifikanzniveau von $\alpha = 0,05$ fest.

Mann-Whitney-U-Test: *Ein Test, welcher zwei unabhängige Stichproben mit Ordinalskalen vergleicht, um Unterschiede zu finden. Die Stichproben müssen aus Populationen stammen, die die gleiche Verteilung haben.*

t-Test: *Ein Test, welcher die Mittelwerte zweier unabhängiger Stichproben mit Intervallskalen vergleicht, um Unterschiede zu finden. Die Stichproben müssen aus normalverteilten Populationen mit annähernd gleichen Varianzen stammen.*

Wir hatten in Abschnitt 4.2 fünf zu überprüfende Hypothesen festgelegt, nämlich jeweils in Bezug auf die Korrektheit, die Geschwindigkeit, und die Usability im Hinblick auf SUS und die zwei ASQ-Faktoren. Zuvor möchten wir jedoch einmal den demographischen Teil des Fragebogens auswerten. Die Rohdaten können in der Datei `Dashboard_SEE.csv` abgerufen werden, in Abbildung 4.7 können die Antworten aber auch in der Form von Balkendiagrammen betrachtet werden. Besonders wichtige oder interessante Ergebnisse im Vergleich zwischen den Gruppen werden dabei durch zusätzliche Violin-Plots dargestellt.

4.7.1 *Demographische Angaben*

Hier untersuchen wir kurz die in Unterabschnitt 4.4.1 eingeführten Fragen und stellen sicher, dass keine signifikanten[19] Unterschiede zwischen den beiden Gruppen bestehen, um Störfaktoren auszuschließen. Dafür verwenden wir ungerichtete Nullhypothesen der Form „Die Variable X unterscheidet sich nicht zwischen Gruppe 1 und Gruppe 2." und arbeiten deshalb mit einem halbierten Signifikanzniveau von $\frac{\alpha}{2} = 0,025$. Der kritische Wert liegt in diesem Fall für $\frac{\alpha}{2}$ und $n = m = 10$ bei 23. Als „Variable X" untersuchen wir im folgenden die abgefragten Aspekte des demographischen Fragebogens.

GESCHLECHT Das Geschlecht spielt in diesem Fall keine Rolle, da alle zwanzig Teilnehmer männlich sind.

ALTER Für das Alter A haben wir insgesamt einen Durchschnitt von insgesamt $\overline{A} \approx 30,25$ und einen Median von $\widetilde{A} = 27$ ($s_A \approx 10,12$), die Verteilung aufgeteilt nach Gruppe kann in Abbildung 4.8 betrachtet werden. Da wir nicht wissen, ob das Alter in der Population normalverteilt ist, verwenden wir den Mann-Whitney-U-Test. Hier erhalten wir als Ergebnis, dass es keinen signifikanten Unterschied gibt ($U = 59,5; p \approx 0,5$).

Empirische Standardabweichung (s): *Die Standardabweichung einer Stichprobe, welche Variable X misst. Wird wie folgt berechnet (n ist die Anzahl der Datenpunkte):*

$$\frac{1}{n-1} \cdot \sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2}$$

HÖCHSTER ABSCHLUSS Einen Durchschnitt anzugeben ist wegen fehlender Intervallskalen ab hier nicht mehr möglich, aber die mit zwölf Antworten am häufigsten vorgekommene Option „Abitur" lag mit Abstand vorne (siehe Abbildung 4.7a), vermutlich wegen der hohen Anzahl an abgefragten Studenten. Der Median[20] liegt ebenfalls bei „Abitur".

Wir wenden erneut den Mann-Whitney-U-Test[21] an: Die Unterschiede sind auch hier nicht signifikant ($U = 48,5; p \approx 0,93$).

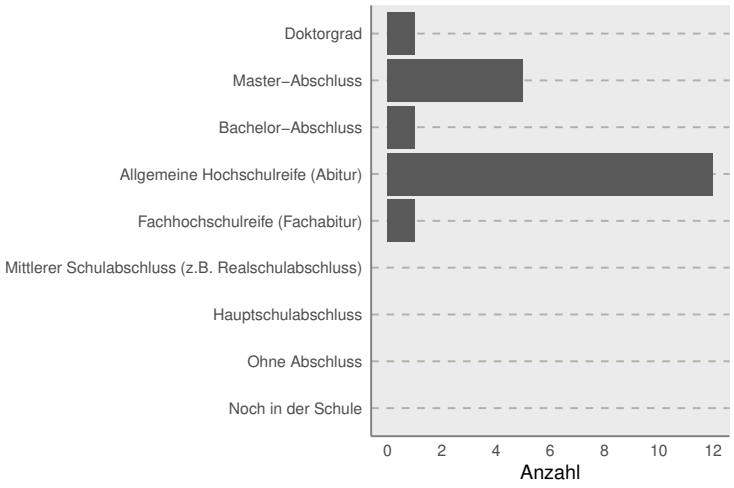
PROGRAMMIERERFAHRUNG Hier und in folgenden Fragen hätte man zwar auch nach einer spezifischen Zahl fragen können, welche die Erfahrung in Jahren angibt. Stattdessen wurde aber eine Diskretisierung in Bereiche vorgenommen, da sich nicht jede Person exakt daran erinnern kann, wie viele Jahre sie schon programmiert.

Von 12 Personen wurde „3–9 Jahre" gewählt (siehe Abbildung 4.7c), dies ist der Modus und der Median zugleich. Es gab einen Teilnehmer ohne Programmiererfahrung.

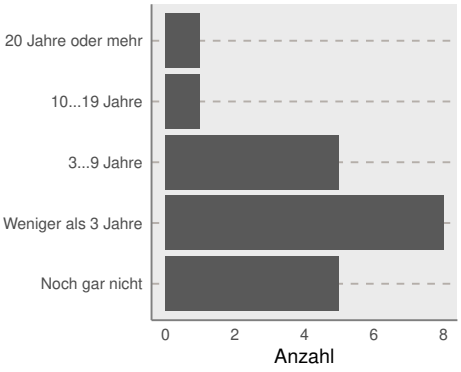
19 Wann immer ich in dieser Arbeit das Wort „signifikant" verwende, meine ich *statistische Signifikanz*.

20 Bei einer geraden Anzahl an Elementen verwenden wir auf Ordinalskalen immer den Obermedian, ansonsten wie üblich den Mittelwert aus Ober- und Untermedian.

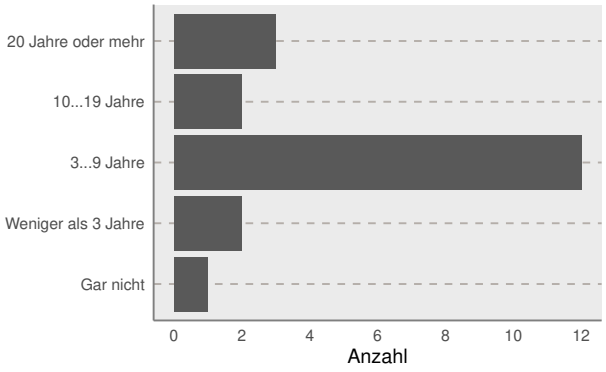
21 Wir müssen uns hier auf eine Ordinalskala festlegen. Das mag natürlich etwas umstritten sein, aber wir verwenden hier die gleiche Reihenfolge wie im Fragebogen (siehe `Common-Eval.xlsx`).



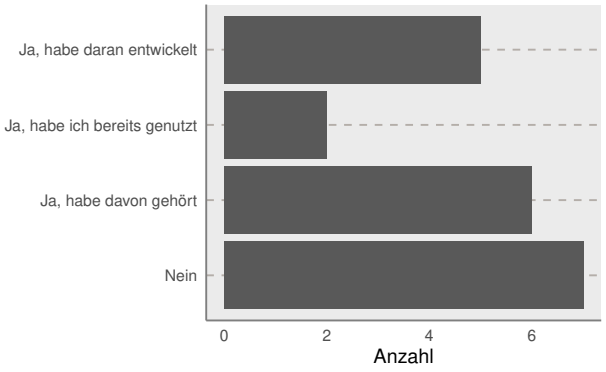
(a) Höchster (Hoch-)Schulabschluss.



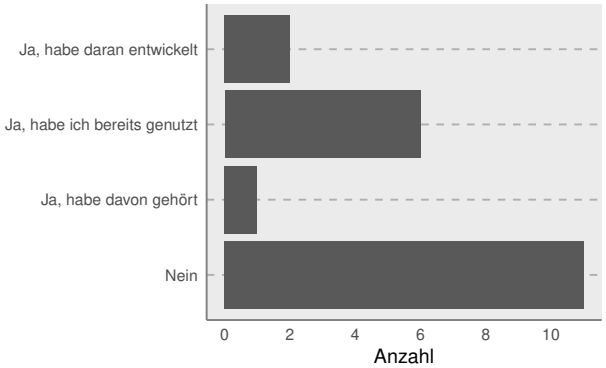
(b) Bisherige Erfahrung in größeren Softwareprojekten.



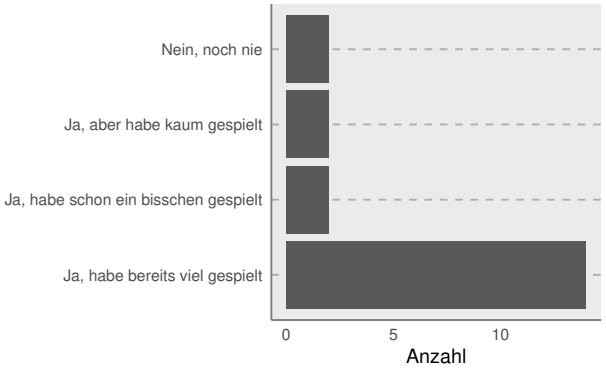
(c) Bisherige Programmiererfahrung.



(d) Bisherige Erfahrung in SEE.



(e) Bisherige Erfahrung mit dem Axivion-Dashboard.



(f) Bisherige Erfahrung mit Videospielen.

Abbildung 4.7: Antworten aller Teilnehmer auf die Fragen des in Unterabschnitt 4.4.1 beschriebenen einführenden Fragebogens.

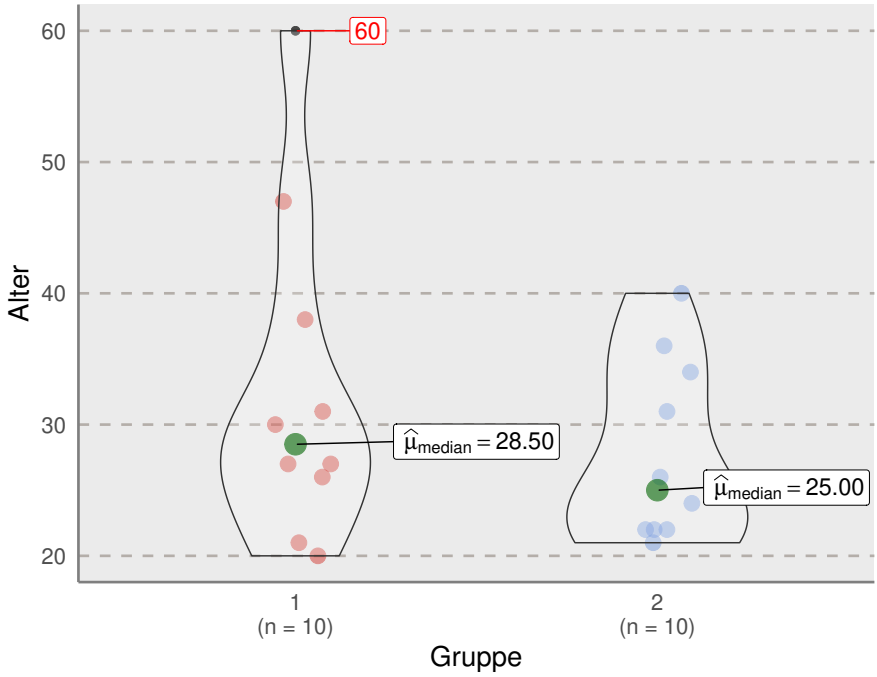


Abbildung 4.8: Altersverteilung in den beiden Gruppen.

Der Mann-Whitney-U-Test ergab erneut keinen signifikanten Unterschied ($U = 48; p \approx 0,9$).

ERFAHRUNG IN GRÖSSEREN PROJEKTEN Am häufigsten wurde die Option „Weniger als 3 Jahre“ gewählt, nämlich von acht Teilnehmern (siehe Abbildung 4.7b), dies ist auch der Median. Der Mann-Whitney-U-Test ergab keinen signifikanten Unterschied ($U = 44; p \approx 0,66$).

ERFAHRUNG MIT SEE Die häufigste Antwort mit sechs Teilnehmern war „Nein“ (siehe Abbildung 4.7d), während der Median bei „Ja, habe davon gehört“ liegt. Laut dem Mann-Whitney-U-Test liegt hier ebenfalls kein signifikanter Unterschied vor ($U = 60; p \approx 0,45$), da diese Frage aber besonders wichtig ist, lohnt es sich, das zugehörige Diagramm in Abbildung 4.9 anzusehen, denn trotz des fehlenden signifikanten Unterschieds lässt sich klar erkennen, dass Gruppe 2 fünf statt nur zwei Personen hat, die nie von SEE gehört haben, während Gruppe 1 z. B. zwei Personen hat, die Erfahrung mit SEE haben (Gruppe 1 hat keine solche Person).

ERFAHRUNG MIT DEM AXIVION-DASHBOARD Die meisten Teilnehmer (elf) hatten noch gar keine Erfahrung mit dem Dashboard (siehe Abbildung 4.7e), der Median liegt ebenfalls bei „Nein“ als Antwort. Nach dem Mann-Whitney-U-Test ist der Unterschied zwischen den Gruppen erneut nicht signifikant ($U = 37,5; p \approx 0,31$). Wie vorhin lohnt es sich aufgrund der besonderen Relevanz dieser Frage jedoch

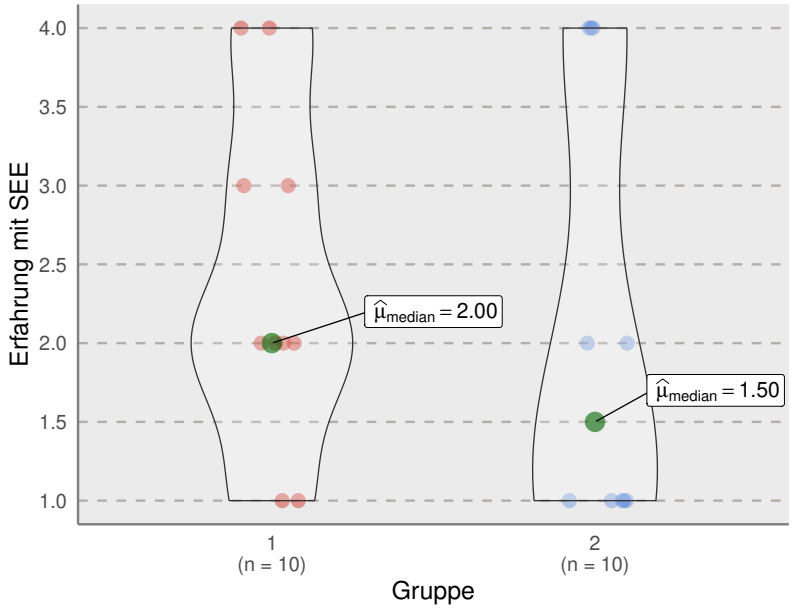


Abbildung 4.9: Erfahrung mit SEE in den beiden Gruppen. 1 bedeutet „Nein“, 2 „Ja, habe davon gehört“, 3 „Ja, habe ich bereits genutzt“ und 4 „Ja, habe daran entwickelt“. Der Median in Gruppe 2 liegt hierbei eigentlich bei „Ja, habe davon gehört“, da wir den Obermedian verwenden.

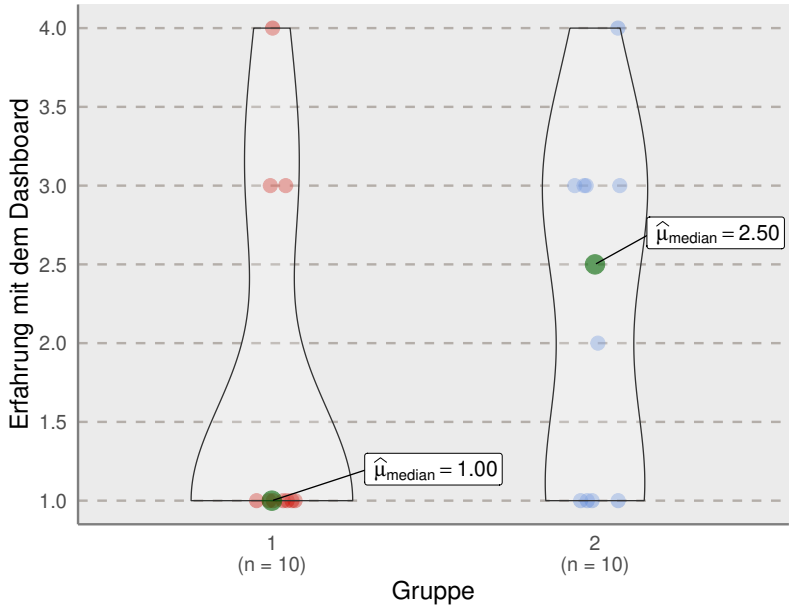


Abbildung 4.10: Erfahrung mit dem AXIVION-Dashboard in den beiden Gruppen. 1 bedeutet „Nein“, 2 „Ja, habe davon gehört“, 3 „Ja, habe ich bereits genutzt“ und 4 „Ja, habe daran entwickelt“. Der Median (Obermedian) in Gruppe 2 liegt hierbei eigentlich bei „Ja, habe ich bereits genutzt“, da es sich um eine Ordinalskala mit vier Werten handelt.

wieder, Abbildung 4.10 zu betrachten. Dort fällt auf, dass z. B. der Obermedian nur in Gruppe 1 bei „Nein“ liegt und in Gruppe 2 bei „Ja, habe ich bereits genutzt“. In Kombination mit dem Vergleich der Erfahrungswerte von SEE im vorherigen Absatz bedeutet dies also, Gruppe 1 hat etwas mehr Erfahrung mit SEE und Gruppe 2 hat etwas mehr Erfahrung mit dem AxIVION-Dashboard — dies ist eine Tatsache, die wir für die spätere Auswertung der Ergebnisse im Hinterkopf behalten sollten.

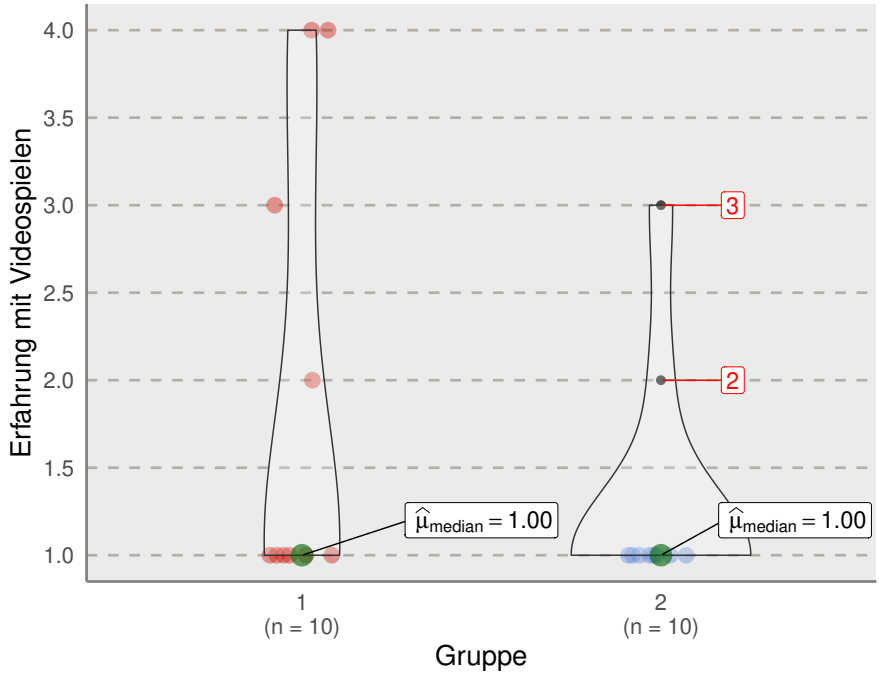


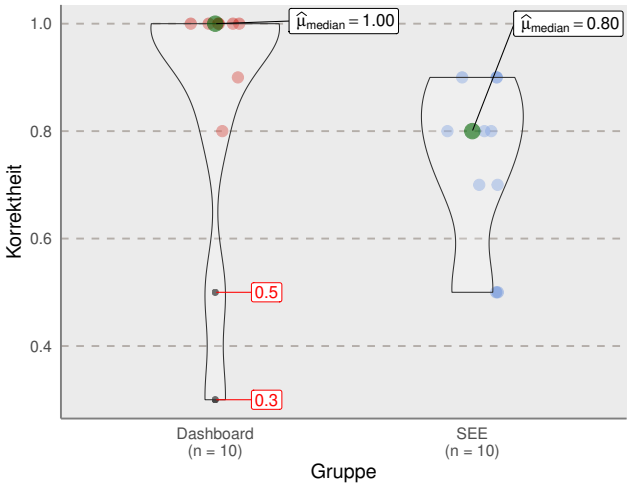
Abbildung 4.11: Erfahrung mit Videospielen, von 1 („Noch nie gespielt“) bis 4 („Viel gespielt“), als Vergleich zwischen den beiden Gruppen.

ERFAHRUNG MIT VIDEOSPIELEN Die mit Abstand häufigste Antwort war von vierzehn Teilnehmern „Ja, habe bereits viel gespielt“ (siehe Abbildung 4.7f), deswegen befindet sich dort auch der Median. Es ist zwar beim Ansehen der Verteilungen schon eher ein Unterschied erkennbar (siehe Abbildung 4.11), allerdings gab es auch hier laut dem Mann-Whitney-U-Test keinen signifikanten Unterschied ($U = 62; p \approx 0,28$).

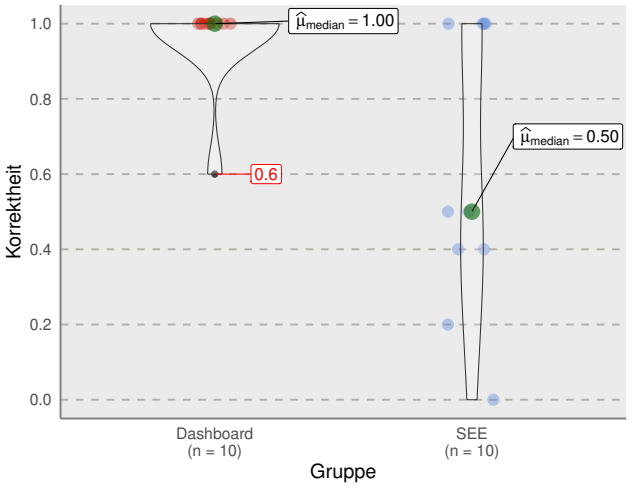
Erfreulicherweise gibt es also bei keiner Variable zwischen den beiden Gruppen signifikante Unterschiede. Wir fahren nun also mit der Analyse der uns eigentlich wichtigen Werte fort.

4.7.2 *Korrektheit*

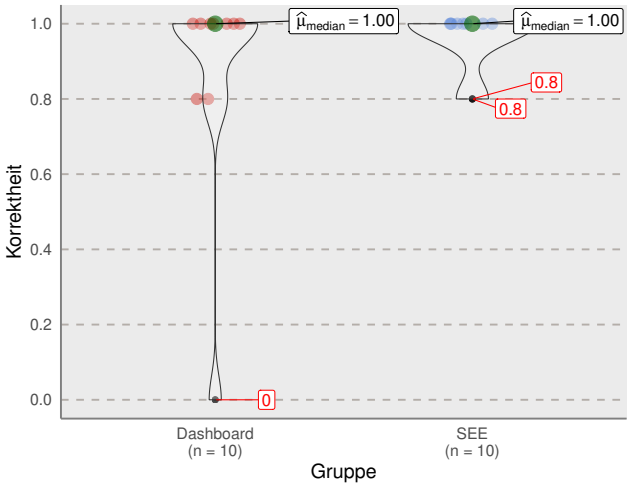
In diesem Teil vergleichen wir die Korrektheit, definiert als der Anteil korrekter Antworten innerhalb einer Aufgabe, zwischen SEE und dem



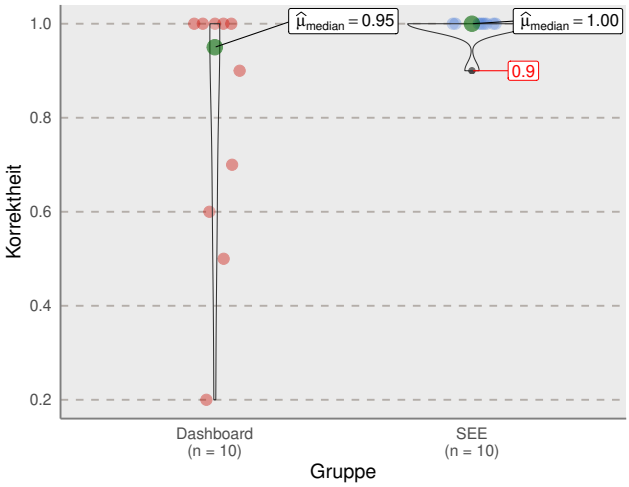
(a) Ergebnisse der Korrektheit von Aufgabe 1.



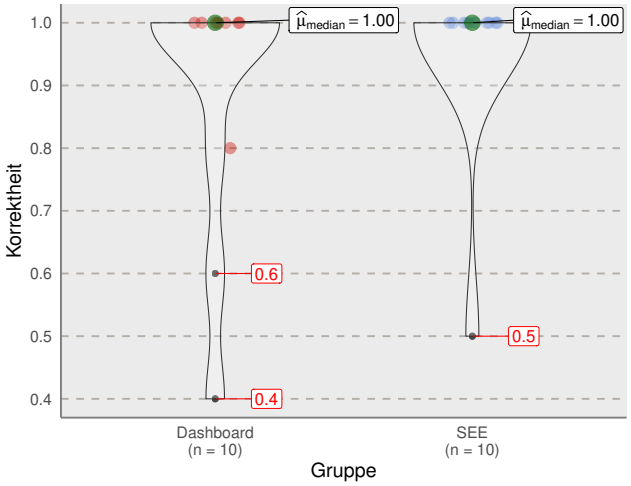
(b) Ergebnisse der Korrektheit von Aufgabe 2.



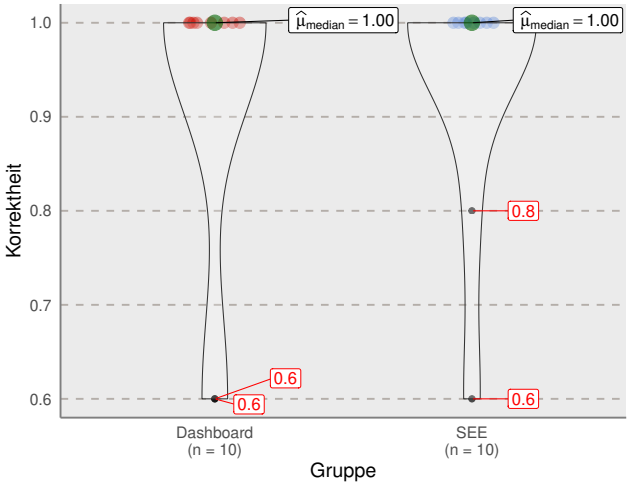
(c) Ergebnisse der Korrektheit von Aufgabe 3.



(d) Ergebnisse der Korrektheit von Aufgabe 4.



(e) Ergebnisse der Korrektheit von Aufgabe 5.



(f) Ergebnisse der Korrektheit von Aufgabe 6.

Abbildung 4.12: Korrektheit (definiert als Prozentsatz der korrekt beantworteten Fragen) aller Teilnehmer, verglichen zwischen dem Dashboard und SEE.

Dashboard. Die richtigen Antworten können für SEE in der angehängten `SEE-Solution.txt` und für das Dashboard in `Dashboard-Solution.txt` nachgelesen werden.

Insgesamt gibt es drei Aufgabentypen (siehe Abschnitt 4.5) die einmal für das Dashboard und einmal für SEE abgefragt werden, diese nummerieren wir daher von 1–6 — die jeweils verwendeten Ordner wurden in Unterabschnitt 4.5.2 festgelegt. Die Auswertung wurde dabei so vorgenommen, dass die Antworten der Teilnehmenden manuell mit den korrekten Antworten abgeglichen worden und die Ergebnisse als zusätzliche Spalten ans Ende der Datei *`Dashboard_SEE.csv`* mit dem Suffix `_c` eingetragen wurden. Dies wurde manuell statt automatisch gemacht, da ansonsten Tippfehler der Teilnehmenden als falsche Antworten gewertet werden würden. In den Aufgaben, in denen sowohl ein Dateiname, als auch die Anzahl der entsprechenden Code-Smells abgefragt wurden, wurde die Antwort auf die letztere Frage nur dann als falsch bewertet, wenn die Anzahl nicht mit der tatsächlichen Anzahl der in der ersten Antwort gegebenen Datei übereinstimmt (so wurden Folgefehler vermieden).

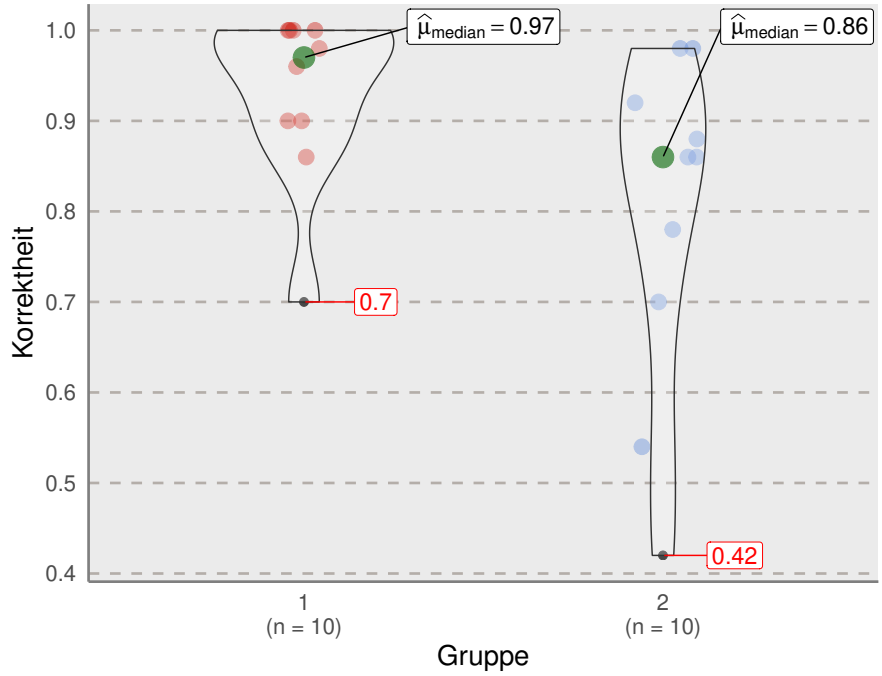


Abbildung 4.13: Vergleich der Korrektheit (in Prozent) zwischen den Teilnehmern der beiden Gruppen (nicht zwischen SEE und Dashboard!)

In Abbildung 4.12 können die vorher bereits eingeführten Violin-Plots für einen Vergleich der jeweiligen Aufgaben zwischen SEE und Dashboard betrachtet werden, in Abbildung 4.13 werden hingegen die beiden *Gruppen* 1 und 2 miteinander verglichen — dort sieht man bereits ein schlechteres Ergebnis in Gruppe 2, verglichen mit Gruppe 1, was sich auch in den Kennwerten widerspiegelt: In Gruppe 1 existiert

ein Durchschnitt von $\overline{K_1} = 0,93$ ($s_{K_1} \approx 0,1$) mit einem Median von $\widetilde{K_1} = 0.97$, während in Gruppe 2 der Durchschnitt bei $\overline{K_2} \approx 0,79$ ($s_{K_2} \approx 0,19$) mit einem Median von $\widetilde{K_2} = 0,86$ liegt. Ein ungerichteter Mann-Whitney-U-Test bestätigt diesen Eindruck: Wir erhalten einen U-Wert von $U = 21,5$ ($p \approx 0,03$), was unterhalb des kritischen Werts von 23 liegt, somit besteht ein signifikanter Unterschied zwischen den beiden Gruppen. Dies kann mehrere Ursachen haben, z. B.:

- Manche Teilnehmer kommen mit der Aufgabenstellung zufällig besser zurecht als andere.
- Die für Aufgaben 1–3 festgelegten Ordner lassen sich einfacher in SEE bearbeiten als im Dashboard, oder umgekehrt für Aufgaben 4–6.
- Wir haben in Unterabschnitt 4.7.1 herausgefunden, dass Gruppe 1 etwas mehr Erfahrung mit SEE und Gruppe 2 etwas mehr Erfahrung mit dem Dashboard hatte. Das ist natürlich etwas spekulativ, aber da das Dashboard ein eher traditionelles Interface hat, welches Nutzer bereits gewöhnt sein könnten, SEE hingegen eher eine neuartige Möglichkeit der Softwareanalyse darstellt, könnte der Effekt von bestehender Erfahrung mit SEE um einiges größer als der Effekt von bestehender Erfahrung mit dem Dashboard sein.

Der letzte Punkt oben lässt sich am ehesten überprüfen. Dies tun wir in Unterabschnitt 4.7.5, wo wir herausfinden, dass genau das Gegenteil der Fall ist: Der Einfluss von bestehender Erfahrung mit dem Dashboard ist um einiges größer als der Einfluss von bestehender Erfahrung mit SEE.

Um nun die Korrektheit bei der Bearbeitung der Aufgaben mit SEE und dem Dashboard zu vergleichen, ziehen wir jeweils die Ergebnisse einer Aufgabe heran und vergleichen diese, bevor wir am Ende noch einmal einen Gesamtvergleich über all Aufgaben hinweg durchführen. Dabei liegt der kritische Wert für $\alpha = 0,05$ und $n = m = 10$ bei $U = 27$.

A1. Die Ergebnisse können in Abbildung 4.12a betrachtet werden. Insgesamt hatten wir hier einen Durchschnitt von 80% mit dem Median bei 85% ($s \approx 21\%$). Der gerichtete Mann-Whitney-U-Test ergibt einen U-Wert von 75 ($p \approx 0,98$), was deutlich über dem kritischen Wert liegt. Führen wir den umgekehrten Vergleich durch, erhalten wir hier im Gegenteil das Ergebnis, dass Aufgabe 1 mit dem Dashboard fehlerfreier als mit SEE bearbeitet werden konnte ($U = 25; p \approx 0,03$).

A2. Die Ergebnisse können in Abbildung 4.12b betrachtet werden. Der Durchschnitt liegt insgesamt bei 78% ($s \approx 33\%$), der Median bei 100%. Hier kann auch in der Abbildung schon gesehen werden, dass abgesehen von einem Ausreißer alle Teilnehmer mit dem Dashboard 100% richtig beantwortet haben, während mit SEE ca.

die Hälfte der Teilnehmer weniger als 50% richtig beantwortet haben. Der Mann-Whitney-U-Test liefert uns ein noch extremeres Ergebnis als eben, nämlich einen U-Wert von 78 ($p \approx 0,99$), im umgekehrten Vergleich stellt das Dashboard entsprechend wieder ein signifikant besseres Ergebnis dar ($U = 22; p \approx 0,01$).

A3. Hier haben wir einen Durchschnitt von 91% ($s = 23\%$) und einen Median von 100%. Der einseitige Mann-Whitney-U-Test ergibt kein signifikantes Ergebnis, mit einem U-Wert von 56 ($p \approx 0,29$). Wie in Abbildung 4.12c zu sehen ist, gibt es jedoch einen extremen Ausreißer beim Dashboard, mit 0% richtig beantworteten Fragen. Eine manuelle Betrachtung der Antworten legt nahe, dass der Teilnehmer ausversehen einen Unterordner geöffnet hat, und diesen mit dem eigentlich relevanten Ordner verwechselt hat.

A4. Der Durchschnitt liegt hier bei 89% ($s = 22\%$), der Median bei 100%. In Abbildung 4.12d lässt sich sehen, dass in SEE bis auf einen Ausreißer kontrastierend zum Dashboard alle Teilnehmer 100% der Fragen richtig beantwortet haben. Der Mann-Whitney-U-Test gibt uns hier zwar ein signifikantes Ergebnis, mit $U = 28$ ($p = 0,02$), hier fällt aber schon an dem U-Wert auf, dass wir das Ergebnis hinterfragen müssen, da dieser nicht unter dem kritischen Wert liegt.

Der Grund dafür liegt darin, dass es sehr viele „Ties“ (d.h. gleiche Werte) gibt, was bei einem rangbasiertem Test suboptimal ist. Die von uns verwendete Implementierung des Mann-Whitney-U-Test verwendet in einem solchen Fall automatisch das sogenannte *Mid-rank*-Verfahren [McG18, S. 5]. Bei einem Anteil von Ties, der über 15% liegt (bei uns liegt er bei 70%) eignen sich laut McGee [McG18, S. 15–16] aber rangbasierte Verfahren mit Tie-Korrekturen wie dem Mid-Rank-Verfahren weniger gut, stattdessen werden bspw. Permutationstests empfohlen. Wir verwenden hier einen solchen Permutationstest, nämlich den für unseren Fall passenden *Fisher-Pitman-Randomisierungstest* [Bor+08, S. 228–231], mit einer Implementierung in R in Form des `coin`-Pakets von Hothorn u. a. [Hot+08]. Hiermit erhalten wir als Ergebnis $Z \approx 2,03$. Da der p-Wert hierfür bei etwa $0,02$ liegt, können wir weiterhin von einem signifikant besseren Ergebnis auf Seiten SEEs ausgehen.

Fisher-Pitman-Randomisierungstest: *Ein Permutationstest, welcher überprüft, ob zwei unabhängige Stichproben aus der gleichen Population stammen. Dabei werden keine Anforderungen an die Populationen gemacht.*

A5. Hier ist der Durchschnitt 91,5% ($s = 19\%$) und der Median 100%. Beim Dashboard gibt es drei, bei SEE eine Person, die Fehler in der Aufgabe gemacht haben, der Rest hat alles richtig beantwortet (siehe Abbildung 4.12e). Es gibt laut dem Mann-Whitney-U-Test keinen signifikanten Unterschied zwischen den beiden Gruppen ($U = 59,5; p \approx 0,86$).

A6. Hier gibt es einen Durchschnitt von 93% ($s = 15\%$) und einen Median von 100%. Wie in Abbildung 4.12f zu sehen ist, gibt es

in beiden Gruppen zwei Ausreißer, alle anderen liegen bei 100% richtigen Antworten. Laut dem Mann-Whitney-U-Test haben wir erneut keinen signifikanten Unterschied ($U = 49; p \approx 0,48$).

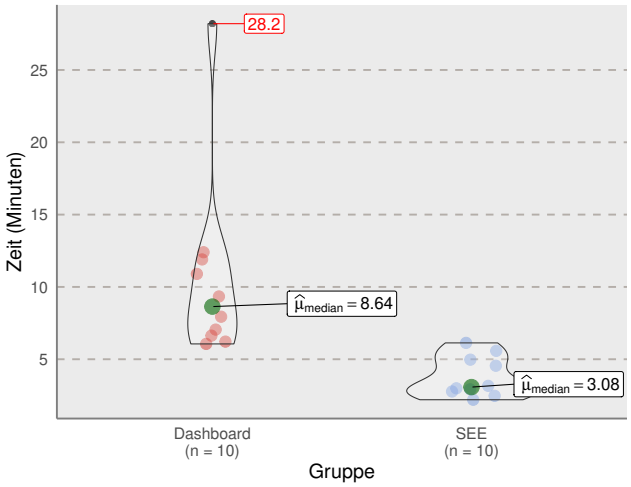
Zusammengefasst haben wir also bei zwei Aufgaben ein besseres Ergebnis beim Dashboard und bei einer Aufgabe ein besseres Ergebnis bei SEE, alle anderen haben keine signifikanten Unterschiede. Insofern müssen wir davon ausgehen, dass man unter Verwendung des Dashboards korrektere Ergebnisse beim Bearbeiten der Aufgaben bekommt als unter Verwendung von SEE. Unsere Nullhypothese H_{a_0} dürfen wir also *nicht verwerfen*.

4.7.3 *Geschwindigkeit*

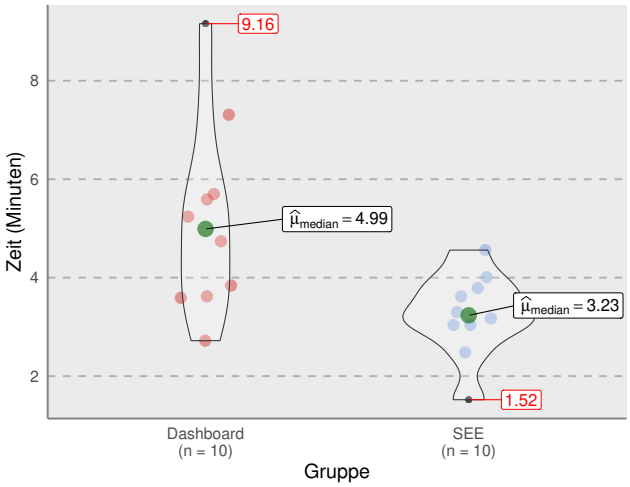
Hierbei wollen wir die benötigte Zeit für jede Aufgabe analysieren, um die Hypothese H_b zu klären. Hierbei wird nur die tatsächliche Zeit, in der die Teilnehmenden die Aufgabe bearbeitet haben, eingerechnet, nicht etwa Zeit, die zum Ausfüllen der Fragebögen verwendet wurde: Die Umsetzung sah dabei so aus, dass ein Teilnehmer vor einer Aufgabe einen Button mit dem Hinweis „Sobald du diesen Button klickst, wird die Zeit gemessen“ betätigen musste — den Moment, zu dem der Teilnehmer das tut, bezeichnen wir als t_1. Am Ende der Aufgabe gab es einen entsprechenden „Klicke hier, sobald du die Aufgabe fertig bearbeitet hast“-Button, welcher dann die Zeit t_2 abspeichert. Weiterhin wird für jedes Eingabefeld i der n_E Eingabefelder der Zeitpunkt der letzten Bearbeitung als t_i gespeichert. Die Bearbeitungszeit der Aufgabe wird schließlich durch $\max(t_2, t_3, \ldots, t_{n_E}) - t_1$ errechnet. Indem wir hier das Maximum der Zeiten der Eingabefelder mitverwenden, verhindern wir, dass ein Teilnehmer ganz am Anfang bereits angibt, er sei fertig, danach aber noch Eingaben tätigt.

Wir beginnen in Abbildung 4.15 mit einem Vergleich zwischen Gruppen 1 und 2, bevor wir mit dem eigentlichen Vergleich zwischen Dashboard und SEE weitermachen. In der Abbildung sieht man die benötigte Zeit, die insgesamt zum Bearbeiten der Aufgaben benötigt wurde. Hierfür wurden einfach die Zeiten für die einzelnen Aufgaben summiert. Es fällt auf, dass sich die beiden Verteilungen sehr ähneln, wenngleich die erste Gruppe im Durchschnitt ($\overline{T_1} \approx 27,46 > \overline{T_2} \approx 23,48$ mit $s_{T_1} \approx 9,9$ und $s_{T_2} \approx 7,2$) und Median ($\widetilde{T_1} = 24,73 > \widetilde{T_2} = 21,82$) etwas länger gebraucht hat. Ein ungerichteter Mann-Whitney-U-Test bestätigt ($U = 65; p \approx 0,28$), dass kein signifikanter Unterschied zwischen den beiden Gruppen besteht.

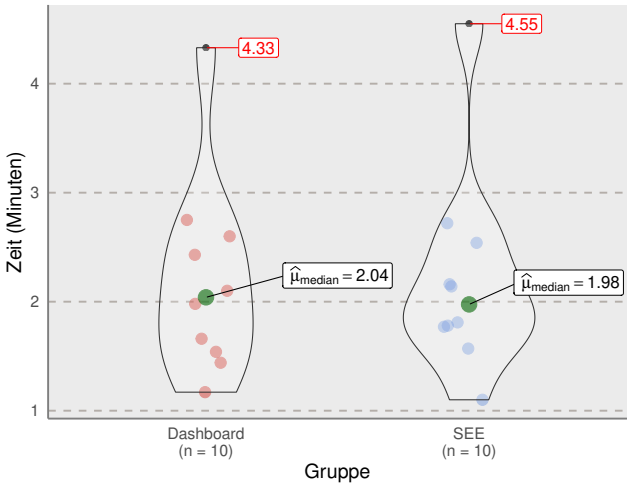
Wir vergleichen nun, ähnlich wie auch in Unterabschnitt 4.7.2, zusammen mit Abbildung 4.14 im Folgenden die einzelnen Aufgaben je nach verwendeter Software im Hinblick auf die benötigte Zeit, wieder unter Verwendung eines einseitigen Mann-Whitney-U-Tests mit $\alpha = 0,05$ und dem kritischen Wert dementsprechend bei $U = 27$:



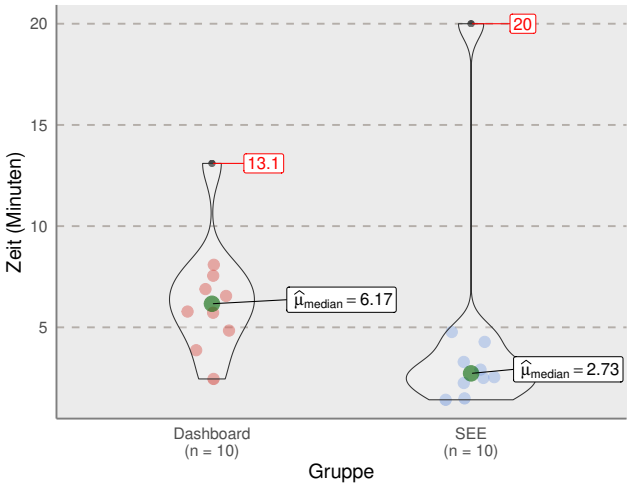
(a) Ergebnisse der benötigten Zeit von Aufgabe 1.



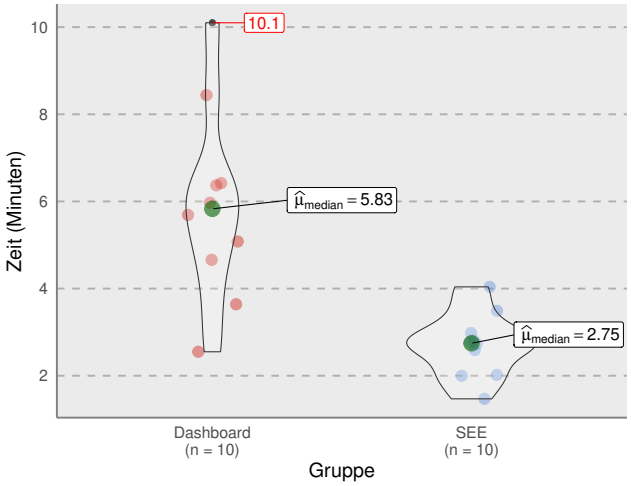
(b) Ergebnisse der benötigten Zeit von Aufgabe 2.



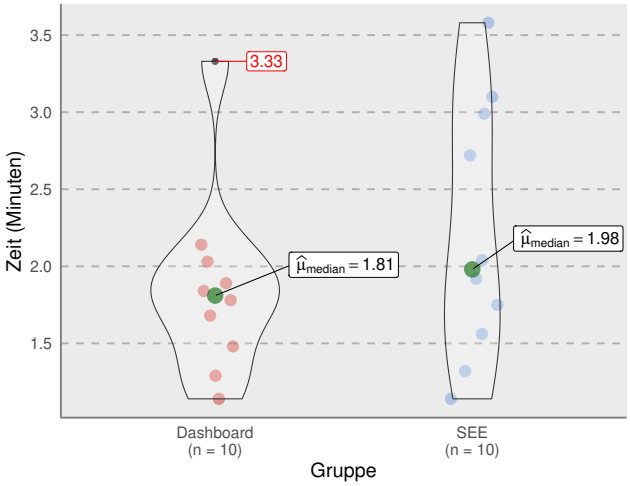
(c) Ergebnisse der benötigten Zeit von Aufgabe 3.



(d) Ergebnisse der benötigten Zeit von Aufgabe 4.



(e) Ergebnisse der benötigten Zeit von Aufgabe 5.



(f) Ergebnisse der benötigten Zeit von Aufgabe 6.

Abbildung 4.14: Benötigte Zeit aller Teilnehmer je Aufgabe, verglichen zwischen dem Dashboard und SEE. Alle Angaben in Minuten. *Achtung, die Skalen unterscheiden sich teilweise stark voneinander.*

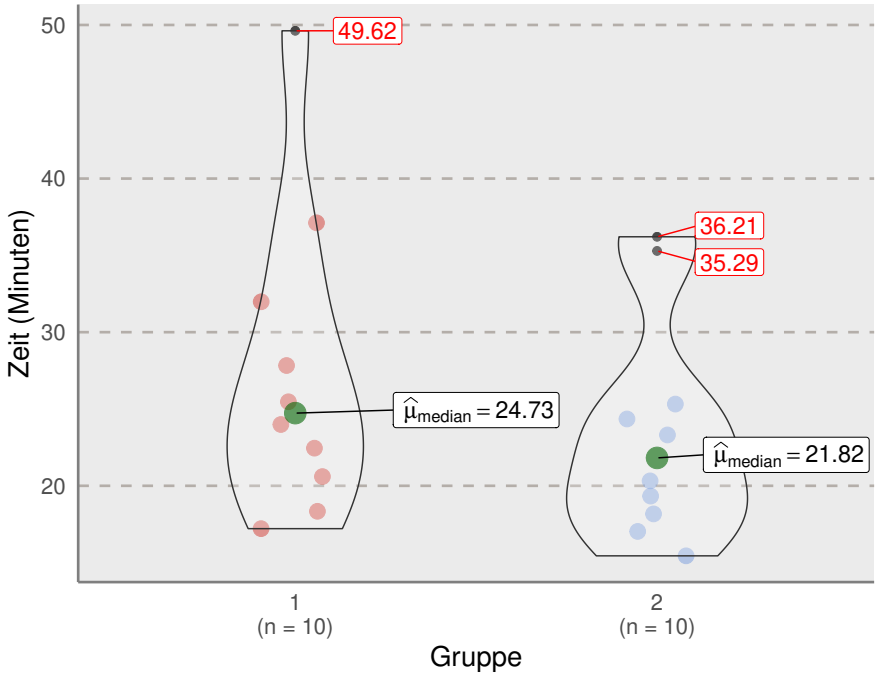


Abbildung 4.15: Vergleich der benötigten Zeit zwischen den beiden Gruppen (nicht zwischen SEE und Dashboard!)

A1. Insgesamt existiert hier ein Durchschnitt von ca. 7,2 Minuten ($s \approx 5,83$) und ein Median von ca. 6,1 Minuten. In Abbildung 4.14a fällt (mehr als in allen anderen Aufgaben) bereits der starke Unterschied auf. Der Mann-Whitney-U-Test ergibt ein extrem eindeutiges Ergebnis von $U = 1$ ($p \approx 10^{-5}$) und bestätigt somit klar den visuellen Eindruck der Abbildung: Mit SEE konnte diese Aufgabe eindeutig schneller als mit dem Dashboard gelöst werden. Wir haben hier, wie in der Abbildung zu sehen ist, einen extremen Ausreißer, der fast 30 Minuten für diese Aufgabe benötigte. Aus den Daten geht nicht klar hervor, wieso dies der Fall ist, es gab jedoch eine 25-minütige Pause zwischen dem Start der Aufgabe und zwischen dem Eintragen in das Diagramm, mglw. hatte der Teilnehmer zwischendurch entgegen der Anweisungen eine Pause gemacht.[22]

A2. Hier haben wir einen Durchschnitt von rund 4,2 Minuten ($s \approx 1,75$) und einen Median von rund 3,7 Minuten. Etwas weniger eindeutig als im vorherigen Bild, aber immer noch erkennbar, ist der Unterschied in Abbildung 4.14b. Laut dem Mann-Whitney-U-Test gibt es hier erneut einen signifikanten Unterschied ($U = 17,5; p \approx 0,008$), der darauf hindeutet, dass sich diese Aufgabe ebenfalls mit SEE schneller bearbeiten lässt.

22 Es handelt sich übrigens um den gleichen Ausreißer wie in Aufgabe 1 der Korrektheit (siehe 4.12a), d.h. nach der 25-minütigen Pause wurde nur 30% Korrektheit erzielt. Eventuell war der Teilnehmer nach der Pause von der Aufgabe abgelenkt.

A3. Im Vergleich zu den vorherigen Aufgaben ist die Zeit bei einem Durchschnitt von ca. 2,2 Minuten ($s \approx 0,9$) und Median von etwa zwei Minuten kürzer als bei den vorherigen Aufgaben, was vermutlich an der anderen Aufgabestellung liegt — hier muss nur der am meisten vorkommende Code-Smell-Typ gefunden werden, weder die größte Datei noch die genaue Zahl selbst (siehe Abschnitt 4.5). Ein Blick in Abbildung 4.14c zeigt den diesmal eher vernachlässigbaren Unterschied der beiden Verteilungen. Wir erhalten diesmal im Vergleich einen U-Wert von 52 ($p \approx 0,57$), dementsprechend gibt es hier also keinen signifikanten Unterschied zwischen den beiden Zeiten.

A4. Der Durchschnitt liegt hier bei ca. 5,5 Minuten ($s \approx 4,4$), der Median bei 4,5 Minuten. Mit Ausnahme eines Ausreißers, der 20 Minuten benötigt hat — hier wurde in der Kommentarspalte erwähnt, dass ein wichtiger Anruf dazwischenkam — sehen wir hier wieder in Abbildung 4.14d, dass die SEE-Teilnahmen im Schnitt kürzer dauerten als die Dashboard-Teilnahmen. Der Mann-Whitney-U-Test zeigt hier ein signifikantes Ergebnis ($U = 18; p \approx 0,007$), also können wir davon ausgehen, dass die Bearbeitung mit SEE schneller erfolgt.

A5. Hier haben wir einen Durchschnitt von ungefähr 4,3 Minuten ($s \approx 2,29$) und einen Median von rund 3,6 Minuten. Das Ergebnis des Mann-Whitney-U-Tests ist entsprechend Abbildung 4.14e recht eindeutig: Wir haben einen U-Wert von nur 8 ($p \approx 0,0003$), womit hier klar ein signifikanter Unterschied zugunsten SEEs besteht.

A6. Der Durchschnitt beträgt hier rund 2 Minuten ($s \approx 0,73$), der Median rund 1,9 Minuten. Wie auch in Abbildung 4.14f schon auffällt, ist der Unterschied diesmal geringer und fällt auch eher zugunsten des Dashboards aus. Der Mann-Whitney-U-Test bestätigt mit einem U-Wert von 61,5 ($p \approx 0,82$), dass wir nicht von einem signifikanten Unterschied ausgehen können.

Zusammengefasst besteht bei vier Aufgaben ein besseres Ergebnis bei SEE, die anderen beiden Aufgaben haben keine signifikanten Unterschiede. In Unterabschnitt 4.7.2 haben wir herausgefunden, dass die Verwendung von SEE in Aufgaben 1 und 2 zu einer geringeren Korrektheit als beim Dashboard geführt hat, während bei Aufgabe 4 umgekehrt SEE besser war. Kombinieren wir das mit den Ergebnissen der Geschwindigkeit hier, können wir sagen, dass sich zumindest Aufgaben 4 und 5 in SEE schneller als im Dashboard bearbeiten lassen — bei den ersten beiden signifikanten Ergebnissen ging die erhöhte Geschwindigkeit allerdings mit einer höheren Fehlerquote (im Vergleich zum Dashboard) einher.

4.7.4 *Usability*

In diesem Abschnitt möchten wir sowohl die Ergebnisse unseres Post-Study-Fragebogens, den SUS, als auch die Ergebnisse des Post-Task ASQ analysieren, um die in Abschnitt 4.2 aufgestellten Hypothesen H_c, H_d und H_e zu testen.

SUS Hier wurde der SUS zuerst wie z. B. von Lewis [Lew18, S. 578] beschrieben entsprechend der Formel

$$S = 2,5 \cdot (20 + \sum_{i=1}^{10} (-1)^{i+1} \cdot S_i)$$

berechnet (wobei S_1 bis S_{10} die Ergebnisse der einzelnen Fragen sind). Der Faktor $(-1)^{i+1}$ existiert dort, da die Fragen abwechselnd positiv und negativ gestellt wurden. Das Ergebnis dieser Berechnung für jeden Teilnehmer lässt sich in Tabelle 4.1 nachlesen.

D	55	95	93	65	55	80	85	85	68	75	98	95	78	63	100	53	95	70	85	75
S	43	90	63	58	55	78	85	83	58	58	88	60	93	65	78	45	35	68	63	88

Tabelle 4.1: Die SUS-Scores aller 20 Teilnehmer, in der ersten Zeile für das AXIVION-Dashboard, in der zweiten für SEE. Es wurde auf ganze Zahlen gerundet.

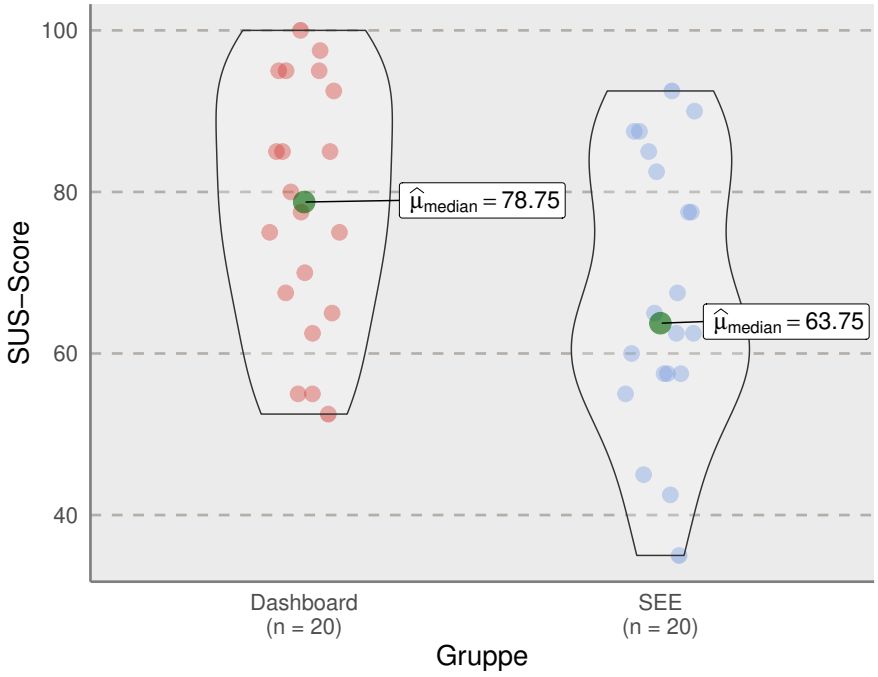


Abbildung 4.16: Vergleich des SUS-Scores zwischen Dashboard und SEE.

Es wurde für das Dashboard ein Durchschnitt von $\overline{S_D} = 78,25$ ($s_{S_D} \approx 15,22$) und ein Median von $\widetilde{S_D} = 78,75$ gemessen — da $\overline{S_D} \approx$

$\widetilde{S_D}$, scheint es kaum Ausreißer zu geben. Für SEE betragen diese Werte jeweils $\overline{S_S} \approx 67,38$ ($s \approx 16,89$) und $\overline{S_D} = 63,75$. Wie man hieran bereits sehen kann, bekommt SEE eindeutig schlechtere SUS-Scores als das Dashboard. Dies kann auch in Abbildung 4.16 erkannt werden.

Dies wird im Mann-Whitney-U-Test, den wir hier wegen der unbekannten Populationsdistribution verwenden[23], noch einmal deutlicher: Hierfür haben wir wegen $\alpha = 0,05$ und $n = m = 20$ einen kritischen Wert von $U = 138$. Bei einem einseitigen Vergleich, um unsere in Abschnitt 4.2 aufgestellte Nullhypothese H_{c_0} zu überprüfen, erhalten wir einen U-Wert von 270 — dies entspricht einer Wahrscheinlichkeit von $p = 97,19\%$, dass wir die uns vorliegenden Werte beobachten, falls die Nullhypothese gilt. Diese können wir also definitiv nicht verwerfen. Im Gegenteil: Die Durchführung des Mann-Whitney-U-Test auf der „anderen Seite“ — d.h., mit H_{f_0} als „Der SUS-Score ist für SEE höher als oder gleich dem für das Dashboard“ ($S_S \geqslant S_D$) und H_{f_1} als „Der SUS-Score ist für SEE niedriger als der für das Dashboard“ ($S_S < S_D$) — ergibt einen U-Wert von $U = 130$. Da dieser unter dem kritischen Wert liegt, müssten wir H_{f_0} *verwerfen* und H_{f_1} akzeptieren.

Hieraus müssen wir folgern, dass die Usability, gemessen durch den SUS, beim Axivion-Dashboard als signifikant besser angesehen wird als bei SEE. Unsere ursprüngliche Hypothese H_{c_0} müssen wir hingegen beibehalten und dürfen sie *nicht verwerfen*. Verschiedene von den Teilnehmern dafür angegebene Gründe lassen sich in Unterabschnitt 4.7.6 finden.

Wie in Unterabschnitt 4.4.3 bereits erwähnt, lässt sich der SUS in zwei Faktoren aufteilen: *Usability* und *Learnability* [LS09, S. 97]. Diese wollen wir hier noch kurz untersuchen. Die aus den Fragen 1–3 und 5–9 bestehende Usability-Skala wird, wie von Peres, Pham und Phillips [PPP13, S. 99] empfohlen, mit der folgenden Formel auf den gleichen Wertebereich von 0–100 wie der normalen SUS-Score gestreckt:

$$S' = 3,125 \cdot (10 + S_1 - S_2 + S_3 + S_5 - S_6 + S_7 - S_8 + S_9)$$

Hier erhalten wir für das Dashboard den Durchschnitt $\overline{S'_D} \approx 73,59$ ($s_{S'_D} \approx 18,54$) mit einem Median von $\widetilde{S'_D} \approx 73,44$, während SEE einen Durchschnitt von $\overline{S'_S} \approx 63,59$ ($s_{S'_S} \approx 17,54$) und einen Median von $\widetilde{S'_S} = 56,25$ hat. Wie einen diese Werte bereits vermuten lassen, ergibt der Mann-Whitney-U-Test hier einen signifikanten Unterschied ($U = 136,5$; $p \approx 0,04$) zugunsten des Dashboards.

Für die *Learnability* betrachten wir Fragen 4 und 10 und verwenden dementsprechend die Formel $S'' = 12,5 \cdot (10 - S_4 - S_{10})$. Das Dashboard bekommt dafür den Durchschnitt $\overline{S''_D} \approx 69,88$ ($s_{S''_D} \approx 6,88$) und den Median $\widetilde{S''_D} = 100$, bei SEE sind es $\overline{S''_S} = 82,5$ ($s_{S''_S} \approx 22,73$) und

23 Theoretisch haben wir hier einen Mix aus gepaarten und ungepaarten Datenpunkten, da wir aber eher α- als β-Fehler vermeiden wollen, verwenden wir den weniger trennscharfen ungepaarten Test.

$\widetilde{S''_S} = 87,5$. Wenig überraschend erhalten wir beim Mann-Whitney-U-Test das gleiche Ergebnis: Bei einem U-Wert von $U = 103$ ($p \approx 0,002$) müssen wir auch hier das Dashboard als den Sieger einstufen.

ASQ Abschließend wollen wir noch den Post-Task ASQ betrachten, der nach jeder Aufgabe abgefragt wurde. Dementsprechend betrachten wir wieder alle Aufgaben getrennt voneinander, um eine möglichst hohe Vergleichbarkeit zu gewährleisten.

Der ASQ bestand in unserem Fall aus zwei Fragen (siehe Unterabschnitt 4.4.2), eine nach der kognitiven Komplexität und eine nach dem nötigen Aufwand. Diese beiden Faktoren werde ich getrennt analysieren anstatt z. B. den Durchschnitt zu ziehen, da es sich um unterschiedliche Konzepte handelt, deren Werte stark voneinander abweichen können. In Abbildung 4.19 kann die Verteilung des ersten ASQ-Faktors (kognitive Komplexität) zwischen den beiden Gruppen gesehen werden, hier haben wir einen Durchschnitt von 32,1 ($s \approx 7,64$) und einen Median von 33, bei 42 maximal möglichen Punkten, wobei eine höhere Punktzahl eine einfacherere Einschätzung repräsentiert. In Abbildung 4.20 kann die Verteilung des Aufwands-Faktors gesehen werden, dort ist der Durchschnitt 28,75 ($s \approx 5,39$) und der Median 27 — die Aufgaben wurden also tendenziell eher aufwändig als kognitiv komplex eingeschätzt.

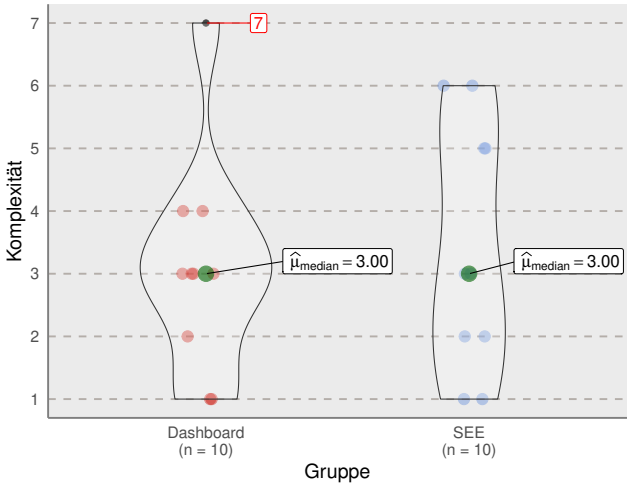
Schauen wir uns nun jede Aufgabe an. Da es sich aus kombinatorischen Gründen um zwölf Abschnitte handelt, beschreiben wir nur die Ergebnisse des Mann-Whitney-U-Tests, die einen signifikanten Unterschied zwischen Dashboard und SEE bedeuten. Alle Abschnitte, die nicht aufgelistet sind, haben also keinen signifikanten Unterschied. Dessen Ergebnisse können (wie auch die gelisteten) in Abbildung 4.17 und Abbildung 4.18 gesehen werden. Unser kritischer Wert liegt wegen $\alpha = 0,05$ und $n = m = 10$ bei $U = 27$.

- *Aufgabe 1, Aufwand:* Hier haben wir einen U-Wert von 25,5 ($p \approx 0,03$), also können wir von einem signifikanten Unterschied zugunsten SEEs ausgehen. Dieser Unterschied kann auch in Abbildung 4.18a gut erkannt werden.

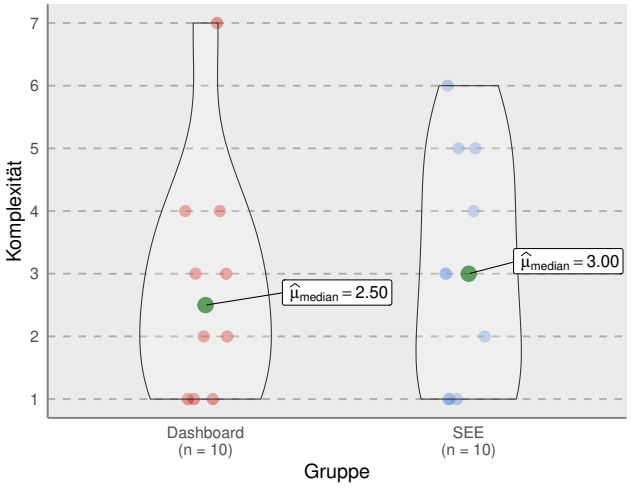
- *Aufgabe 4, Aufwand:* Ein U-Wert von 13,5 ($p \approx 0,003$) zeigt an dieser Stelle klar ein signifikant besseres Ergebnis bei SEE im Vergleich zum Dashboard — in Abbildung 4.18d ist dieser Unterschied ebenfalls sichtbar.

- *Aufgabe 5, Aufwand:* Durch einen U-Wert von 20,5 ($p \approx 0,01$) haben wir auch hier ein signifikantes Ergebnis, welches SEE erneut als die weniger aufwändigere Option darstellt (siehe Abbildung 4.18e).

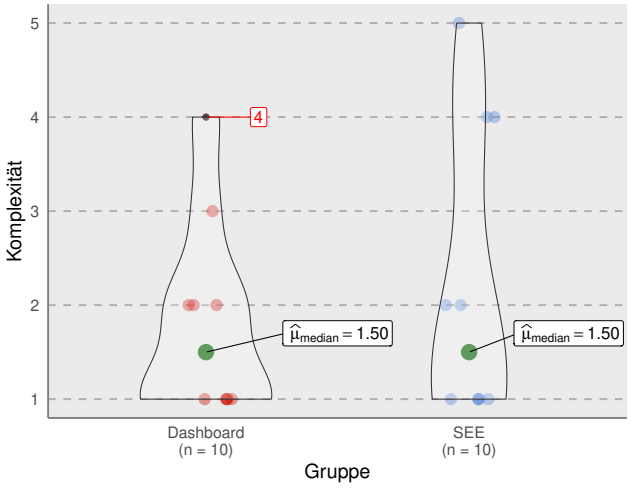
Während in manchen Diagrammen (z. B. Abbildungen 4.17e, 4.17f, 4.18b) zwar auch gewisse Unterschiede zu erkennen sind, erreichen diese im Mann-Whitney-U-Test keinen U-Wert, der unter den kritischen



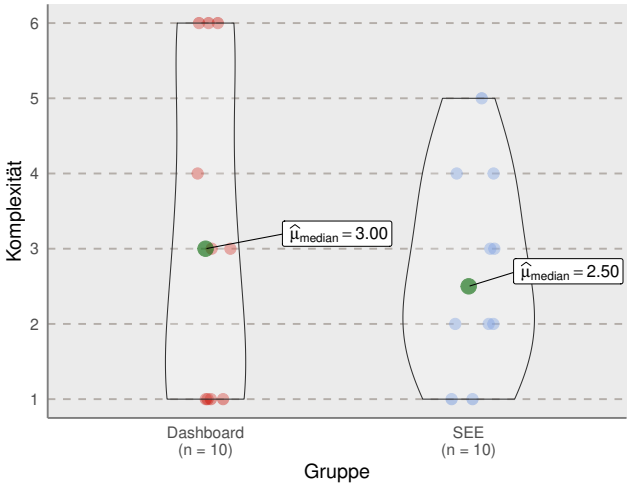
(a) Komplexität von Aufgabe 1.



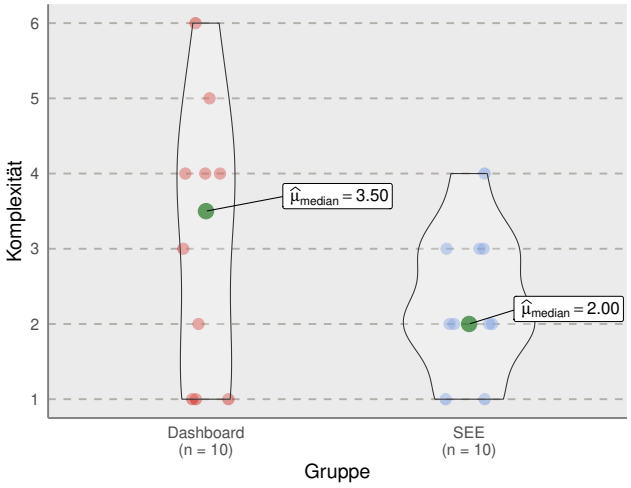
(b) Komplexität von Aufgabe 2.



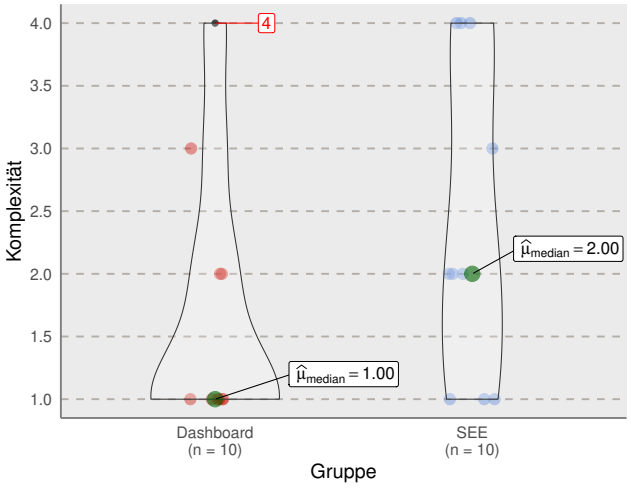
(c) Komplexität von Aufgabe 3.



(d) Komplexität von Aufgabe 4.

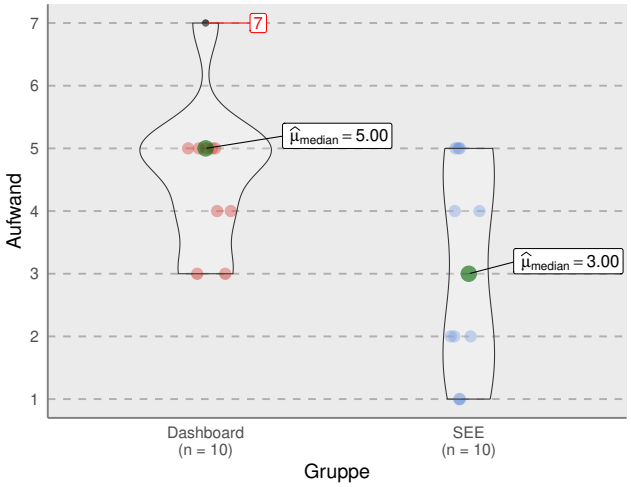


(e) Komplexität von Aufgabe 5.

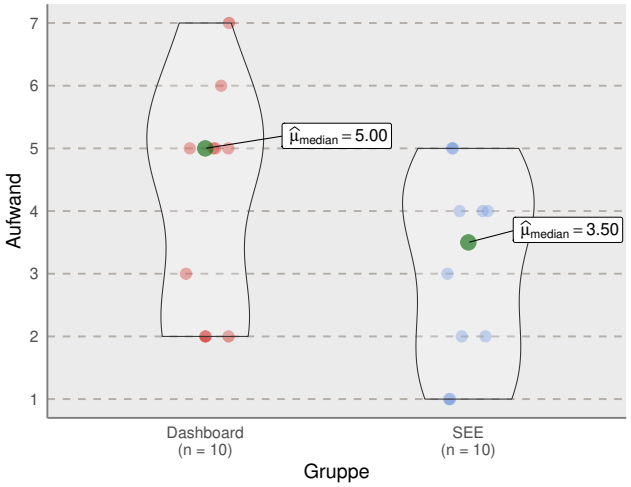


(f) Komplexität von Aufgabe 6.

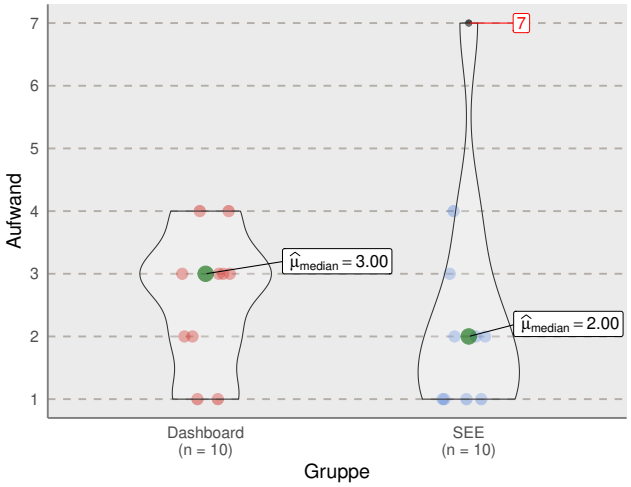
Abbildung 4.17: Die Ergebnisse der ersten Frage des ASQ, „Insgesamt bin ich zufrieden, wie leicht diese Aufgabe zu lösen war. (‚Leicht' in Bezug auf die Komplexität bzw. den kognitiven Anspruch der Aufgabe, nicht wie mühselig die Aufgabe war.)" 1 repräsentiert den niedrigsten Aufwand, 7 den höchsten.



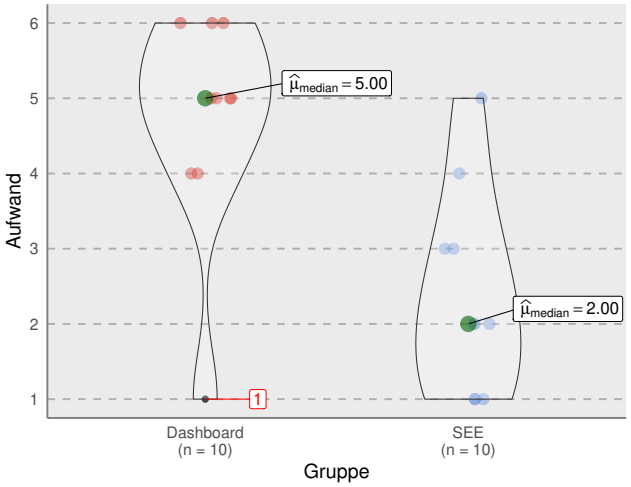
(a) Aufwand von Aufgabe 1.



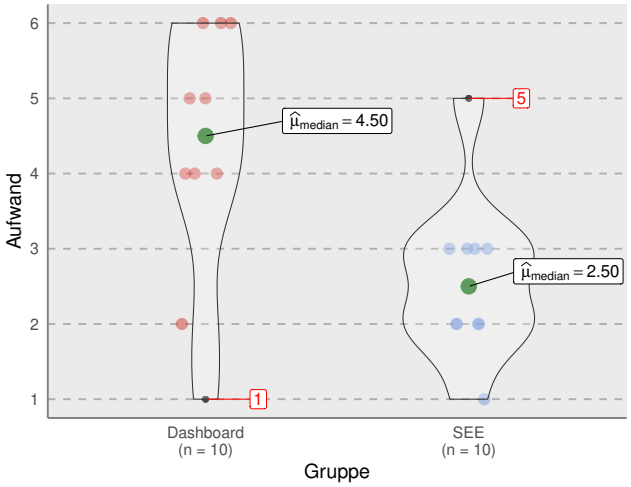
(b) Aufwand von Aufgabe 2.



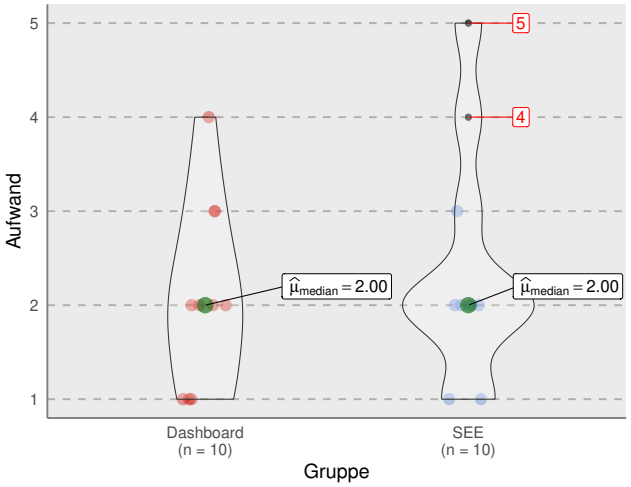
(c) Aufwand von Aufgabe 3.



(d) Aufwand von Aufgabe 4.



(e) Aufwand von Aufgabe 5.



(f) Aufwand von Aufgabe 6.

Abbildung 4.18: Die Ergebnisse der zweiten Frage des ASQ, „Ich bin zufrieden mit der Zeit, die es gedauert hat, diese Aufgabe zu lösen.“ 1 repräsentiert den niedrigsten Aufwand, 7 den höchsten.

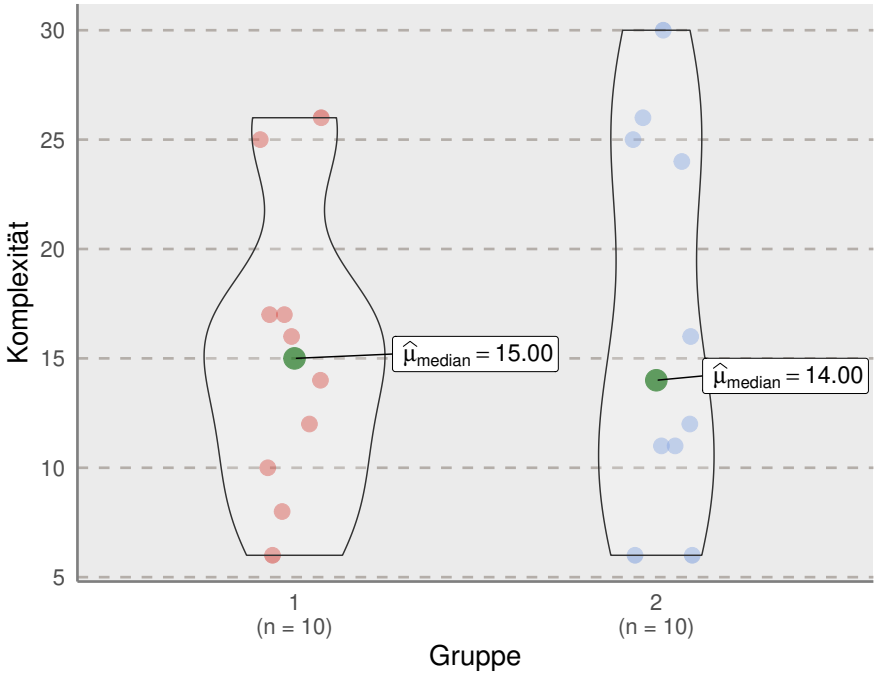


Abbildung 4.19: Vergleich des ASQ-Faktors der kognitiven Komplexität (1: „Sehr leicht“, 7: „Sehr schwer“) im Vergleich zwischen Gruppe 1 und 2 (*nicht* zwischen Dashboard und SEE).

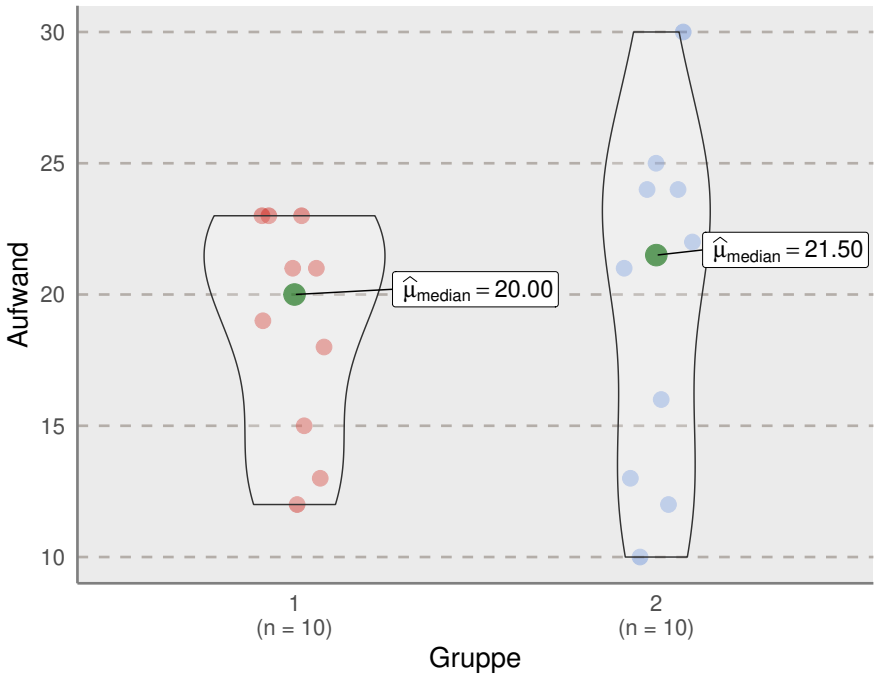


Abbildung 4.20: Vergleich des ASQ-Faktors des nötigen Aufwands (1: „Sehr leicht“, 7: „Sehr schwer“) im Vergleich zwischen Gruppe 1 und 2 (*nicht* zwischen Dashboard und SEE).

Wert fällt und somit unsere Anforderungen an das Signifikanzniveau erfüllen würde. Insgesamt haben wir hier also keine Vergleiche, in denen das Dashboard besser abschneidet, aber drei, bei denen SEE sich besser eignet. Hiervon sind jedoch alle drei von der Kategorie *Aufwand*, bei der Komplexität gab es nirgendwo signifikanten Unterschiede — dies deckt sich größtenteils mit der Beobachtung aus Unterabschnitt 4.7.3, wo wir in den Aufgaben 1, 2, 4 und 5 eine signifikant kürzere Bearbeitungsdauer in SEE festgestellt hatten.

Wir können hier wegen der drei signifikanten Unterschiede zwar tendenziell die Nullhypothese H_{d_0} *verwerfen* und somit über die alternative Hypothese H_{d_1} davon ausgehen, dass der über den ASQ gemessene Aufwand mit SEE geringer als beim Dashboard war, aber die Nullhypothese H_{e_0}, die diese Aussage über die Komplexität trifft, können wir *nicht verwerfen*.

4.7.5 *Der Einfluss von Erfahrung*

An dieser Stelle wollen wir den Zusammenhang zwischen bestimmten unabhängigen Variablen des demographischen Fragebogens und abhängigen Variablen der Aufgaben testen, hauptsächlich den Einfluss von Erfahrung mit einem jeweiligen System auf die drei abhängigen Variablen Korrektheit, Zeit und Usability. Somit vergleichen wir hier jeweils eine ordinalskalierte, diskrete Variable mit einer kontinuierlichen. In Situationen mit wenigen möglichen Werten auf der Ordinalskala (weniger als fünf oder sechs), wie in unserem Fall, schlägt Khamis [Kha08, S. 158] vor, den *Kendall-Tau-Korrelationskoeffizienten* zu verwenden, um eine Korrelation zu messen. Für den Vergleich des Alters verwenden wir (ebenfalls auf Empfehlung dieser Quelle) den Pearson-Korrelationskoeffizienten. Die Erfahrung mit Videospielen lassen wir bei dieser Analyse aus, da zu viele Antworten identisch waren (siehe 4.7f). Das Gleiche gilt für die generelle Programmiererfahrung (siehe 4.7c).

Kendall-Tau-Korrelationskoeffizient (τ_b): *Ein rangbasierter Korrelationskoeffizient, der zwei ordinalskalierte oder intervallskalierte Variablen auf einen Zusammenhang in Form einer ordinalen Assoziation testet. Der Wertebereich des Koeffizienten liegt dabei bei $[-1;1]$, wobei ein positiver Wert eine positive Korrelation, 0 keine Korrelation und ein negativer Wert eine negative Korrelation impliziert. Die Ausprägung des Betrags entspricht dabei dem Ausmaß der Korrelation.*

	SEE-Aufgabe	**Dashboard-Aufgabe**
Korrektheit	$\tau_b \approx -0,01$ ($p \approx 0,97$)	$\tau_b \approx 0,07$ ($p \approx 0,72$)
Zeit	$\tau_b \approx -0,13$ ($p \approx 0,47$)	$\tau_b \approx 0,09$ ($p \approx 0,61$)
SUS	$\tau_b \approx 0,02$ ($p \approx 0,92$)	$\tau_b \approx -0,22$ ($p \approx 0,22$)
ASQ (Aufwand)	$\tau_b \approx -0,11$ ($p \approx 0,54$)	$\tau_b \approx -0,18$ ($p \approx 0,32$)
ASQ (Komplexität)	$\tau_b \approx 0,01$ ($p \approx 0,95$)	$\tau_b \approx 0,01$ ($p \approx 0,97$)

Tabelle 4.2: Korrelation zwischen Erfahrung mit SEE und den abhängigen Variablen, berechnet durch den Kendall-Tau-Korrelationskoeffizienten.

ERFAHRUNG MIT SEE In Tabelle 4.2 können jeweils die Korrelationen zwischen der Erfahrung mit SEE und der entsprechenden Variable

(unterschieden nach System) gesehen werden. Verwenden wir wie bisher unser Signifikanzniveau von $\alpha = 0,05$ gibt es hier nirgendwo auch nur annähernd eine signifikante Korrelation — daraus schließen wir, dass die bisherige Erfahrung mit SEE wohl keinen erkennbaren Einfluss auf die abhängigen Variablen hatte.

	SEE-Aufgabe	**Dashboard-Aufgabe**
Korrektheit	$\tau_b \approx 0,24$ ($p \approx 0,21$)	$\tau_b \approx 0,22$ ($p \approx 0,24$)
Zeit	$\tau_b \approx -0,14$ ($p \approx 0,44$)	$\tau_b \approx -0,35$ ($p \approx 0,05$)
SUS	$\tau_b \approx -0,21$ ($p \approx 0,26$)	$\tau_b \approx 0,09$ ($p \approx 0,63$)
ASQ (Aufwand)	$\tau_b \approx 0,01$ ($p \approx 0,97$)	$\tau_b \approx -0,4$ ($p \approx 0,03$)
ASQ (Komplexität)	$\tau_b \approx -0,25$ ($p \approx 0,19$)	$\tau_b \approx -0,27$ ($p \approx 0,15$)

Tabelle 4.3: Korrelation zwischen Erfahrung mit dem Axivion-Dashboard und den abhängigen Variablen, berechnet durch den Kendall-Tau-Korrelationskoeffizienten.

ERFAHRUNG MIT DEM DASHBOARD Die Korrelationen zwischen der Erfahrung mit dem Dashboard und den abhängigen Variablen, getrennt nach System, sind in Tabelle 4.3 gelistet. Hier haben wir nun eine signifikante Korrelation: Und zwar korreliert der subjektiv empfundene Aufwand negativ ($\tau_b \approx -0,4$) mit der Erfahrung, die ein Teilnehmer mit dem Dashboard hat. Das bedeutet, je erfahrener ein Teilnehmer mit dem Dashboard ist, desto geringer bewertet dieser den empfundenen Aufwand. Passend dazu gibt es bei dem tatsächlichen Aufwand, gemessen durch die Zeit, *fast* einen signifikanten Zusammenhang ($\tau_b \approx -0,35$).

Dies kann mehrere Ursachen haben. Die wahrscheinlichste Möglichkeit besteht wohl darin, dass die erfahrenen Nutzer des Dashboards dieses wegen ihrer Erfahrung einfach schneller bedienen können und sich mit den Features bereits gut auskennen und daher auch die Aufgabe als weniger aufwändig bewerten. Eine weitere Möglichkeit ist, dass die Erfahrung mit dem Dashboard wiederum stark mit einem anderen Faktor zusammenhängt[24], der wiederum diese höhere Bewertung des Aufwands erklärt.

ERFAHRUNG IN GRÖSSEREN SOFTWAREPROJEKTEN Die Korrelationen sieht man in Tabelle 4.4, dort fällt an einer Stelle ein signifikanter Zusammenhang auf: Es existiert nämlich eine negative Korrelation zwischen der benötigten Zeit in den Dashboard-Aufgaben und der bisherigen Erfahrung in größeren Softwareprojekten ($\tau_b \approx -0,37$). Das bedeutet, Teilnehmer, die mehr Erfahrung mit größeren Softwareprojekten haben, können — vielleicht, weil sie die Struktur komplexerer Projekte daher in traditionellen, gewohnten Tools besser erfassen können — die Aufgaben mit dem Dashboard im Vergleich zu anderen schneller lösen.

24 Dies ist z. B. bei dem Schulabschluss der Fall, welcher signifikant mit der Dashboard-Erfahrung korreliert ($\tau_b \approx 0,48$; $p \approx 0,01$).

	SEE-Aufgabe	Dashboard-Aufgabe
Korrektheit	$\tau_b \approx 0,1$ ($p \approx 0,59$)	$\tau_b \approx 0,21$ ($p \approx 0,26$)
Zeit	$\tau_b \approx -0,15$ ($p \approx 0,41$)	$\tau_b \approx -0,37$ ($p \approx 0,04$)
SUS	$\tau_b \approx -0,06$ ($p \approx 0,73$)	$\tau_b \approx 0,17$ ($p \approx 0,34$)
ASQ (Aufwand)	$\tau_b \approx 0,07$ ($p \approx 0,7$)	$\tau_b \approx 0,3$ ($p \approx 0,1$)
ASQ (Komplexität)	$\tau_b \approx -0,24$ ($p \approx 0,2$)	$\tau_b \approx -0,32$ ($p \approx 0,08$)

Tabelle 4.4: Korrelation zwischen Erfahrung in größeren Softwareprojekten und den abhängigen Variablen, berechnet durch den Kendall-Tau-Korrelationskoeffizienten.

	SEE-Aufgabe	Dashboard-Aufgabe
Korrektheit	$\tau_b \approx 0,22$ ($p \approx 0,25$)	$\tau_b \approx -0,01$ ($p \approx 0,94$)
Zeit	$\tau_b \approx -0,23$ ($p \approx 0,21$)	$\tau_b \approx -0,18$ ($p \approx 0,33$)
SUS	$\tau_b \approx -0,52$ ($p \approx 0,005$)	$\tau_b \approx -0,01$ ($p \approx 0,97$)
ASQ (Aufwand)	$\tau_b \approx -0,35$ ($p \approx 0,06$)	$\tau_b \approx 0,17$ ($p \approx 0,37$)
ASQ (Komplexität)	$\tau_b \approx 0,1$ ($p \approx 0,6$)	$\tau_b \approx -0,57$ ($p \approx 0,002$)

Tabelle 4.5: Korrelation zwischen Hochschulabschluss und den abhängigen Variablen, berechnet durch den Kendall-Tau-Korrelationskoeffizienten.

SCHULABSCHLUSS Die Korrelationen in Zusammenhang mit dem erreichten Hochschulabschluss sind in Tabelle 4.5 aufgelistet. Es fallen zwei signifikante Zusammenhänge auf: Erstens existiert ein sehr signifikanter negativer Zusammenhang zwischen dem erreichten Abschluss und dem SUS für SEE ($\tau_b \approx -0,52$) und zweitens existiert ebenfalls ein sehr signifikanter negativer Zusammenhang zwischen dem Abschluss und dem Komplexitätsfaktor des ASQ ($\tau_b \approx -0,57$). Das bedeutet, je höher der erreichte Hochschulabschluss eines Teilnehmers ist, desto schlechter wird im Schnitt der SUS von SEE und desto besser[25] der Komplexitätsfaktor im ASQ für das Dashboard bewertet.

Eine mögliche Erklärung wäre dafür z. B., dass mit einem höheren Schulabschluss auch ein höheres Alter ($\tau_b \approx 0,44; p \approx 0,02$) und somit eher eine Erfahrung mit konventioneller Software einhergeht — dies mag bei der eher unkonventionellen Bedienung von SEE und der eher konventionellen Bedienung des Dashboards eine Rolle spielen.

ALTER Schließlich untersuchen wir noch die das Alter betreffenden Korrelationen, diese sind in Tabelle 4.6 sichtbar. Dort fällt an zwei Stellen ein signifikanter Zusammenhang auf: Je älter ein Teilnehmer ist, desto schlechter bewertet er im Schnitt die Usability (über den SUS) von SEE ($r \approx -0,5$) und desto weniger komplex empfindet er im Schnitt die Aufgaben (über die ASQ-Komplexität) des Dashboards

25 Zur Erinnerung, ein höherer ASQ-Wert bedeutet eine höhere Komplexität bzw. einen höheren Aufwand.

	SEE-Aufgabe	Dashboard-Aufgabe
Korrektheit	$r \approx -0,1$ ($p \approx 0,67$)	$r \approx 0,15$ ($p \approx 0,52$)
Zeit	$r \approx -0,22$ ($p \approx 0,35$)	$r \approx 0,3$ ($p \approx 0,2$)
SUS	$r \approx -0,5$ ($p \approx 0,02$)	$r \approx -0,16$ ($p \approx 0,49$)
ASQ (Aufwand)	$r \approx -0,27$ ($p \approx 0,24$)	$r \approx 0,19$ ($p \approx 0,42$)
ASQ (Komplexität)	$r \approx 0,31$ ($p \approx 0,19$)	$r \approx -0,54$ ($p \approx 0,01$)

Tabelle 4.6: Korrelation zwischen Alter und den abhängigen Variablen, berechnet durch den Pearson-Korrelationskoeffizienten.

($r \approx -0,54$). Dies ist das gleiche Ergebnis wie eben bei der Betrachtung des Schulabschlusses und kann daher die gleichen Gründe haben.

4.7.6 *Kommentare der Teilnehmer*

Die Teilnehmer konnten jeweils nach der SUS-Evaluation für das Axivion-Dashboard und SEE Kommentare hinterlassen, um spezifische Anmerkungen zu diesen Systemen zu machen. Diese wollen wir in diesem Abschnitt auswerten. Der Originaltext der Kommentare kann in `Dashboard_SEE.csv` nachgelesen werden.

Es gab folgende Vorschläge für das **Axivion-Dashboard** (Die Kommentare wurden als Feedback an die Axivion-Mitarbeiter weitergegeben):

- Es sollte im Dashboard eine Spalte für die Anzahl an Codezeilen geben. (**10x**)
- Das Dashboard sollte sich die letzte Ordnerposition merken, da man sonst zum Wurzelverzeichnis zurückspringt. (**2x**)
- Es sollten die absoluten Code-Smell-Zahlen ins Verhältnis zur Anzahl der Code-Zeilen gesetzt werden, um kritische Bereiche zu erkennen. (**2x**)
- Code-Smell-Typen sollten farblich unterschieden werden. (**1x**)
- Es sollte eine „horizontale“ Sortierung geben, mit der man sich pro Zeile die häufigste Code-Smell-Art anzeigen lassen kann. (**1x**)

Folgende Vorschläge wurden für **SEE** abgeschickt (in Unterpunkten sind diesmal mögliche Lösungen der Probleme angegeben):

- Das Ablesen der Code-Smell-Zahlen bzw. Dateinamen ist schwierig, da diese oft von anderen Dingen überlagert werden. (**7x**)
  - Hier wäre es wohl sinnvoll, beim Fokussieren auf einen Knoten alle anderen Zahlen/Texte benachbarter Knoten auszublenden. Ebenso sollte nicht gleichzeitig der Dateiname und die Code-Smell-Anzahl angezeigt werden.

- Es sollte eine Suchfunktion für Dateien/Ordner geben. (**5x**)
  - Eine solche Funktion wurde bereits von mir in SEE implementiert, ich wollte die Erklärung für die beiden Programme aber nicht zu lang werden lassen, deswegen habe ich dort jeweils die Suchfunktion nicht erwähnt. Rückblickend wäre das die zusätzliche Länge aber wohl wert gewesen — wird nochmal eine Studie dieser Art durchgeführt, sollte also die Suchfunktion mit erwähnt werden.
- Die Code-Smell-Zahlen sollten dem Nutzer immer zugewandt sein. (**5x**)
  - Dies ist leicht zu bewerkstelligen. In einer ursprünglichen Version war dies auch bereits der Fall, wurde dann aber von mir fälschlicherweise als zu ablenkend im Vergleich zur dann verwendeten Version eingeschätzt.
- Dadurch, dass die Würfel komplett weiß sind, ist die Größe schwerer abschätzbar und einzelne Dateien sind schwerer voneinander zu trennen. (**2x**)
  - Hier scheint es in SEE ein Problem mit dem Shading zu geben.
- Die Animationen wirken teilweise störend. (**1x**)
  - Die Animationen lassen sich bereits deaktivieren, diese Funktionalität wurde im Rahmen der Evaluation aber den Teilnehmenden nicht gezeigt.
- Es sollte nicht nur einen Reset-Knopf für die Position der CodeCity, sondern auch für die Position des Spielers geben. (**1x**)
  - Dies würde sich ziemlich schnell in SEE umsetzen lassen.
- Eine Art „Karte“ der Tastenbelegung, die man jederzeit aufrufen kann, wäre hilfreich. (**1x**)
  - Das klingt wegen der vielen Tastenbelegungen in SEE nach einer guten Idee.

4.8 THREATS TO VALIDITY

Wir wollen nun einige so genannte „Threats to Validity“ (deutsch „Bedrohungen der Validität“), d.h. mögliche Probleme in der Umsetzung, untersuchen. Dabei unterscheiden wir zwischen *internen* (Unterabschnitt 4.8.1) und *externen* (Unterabschnitt 4.8.2) Threats to Validity — bei ersteren geht es darum, ob tatsächlich ein kausaler Zusammenhang zwischen der Verwendung von SEE bzw. dem Dashboard und den beobachteten Werten besteht und bei letzteren geht es darum, ob sich unser Ergebnis auch auf andere Situationen verallgemeinern lässt.

Wir gehen dabei grob die von Campbell, Stanley und Gage [CSG63, S. 5–6] aufgestellten Bedrohungen[26] durch und beantworten für die bei dieser Studie relevanten, ob und wie diese adressiert wurden.

4.8.1 *Interne Validität*

Fangen wir mit den internen Bedrohungen der Validität an. Hier wird in Frage gestellt, ob die unabhängige Variable (in unserem Fall also, ob das Dashboard oder SEE verwendet wird) wirklich in einem Kausalzusammenhang mit der abhängigen Variable (Korrektheit, Zeit, Usability) steht.

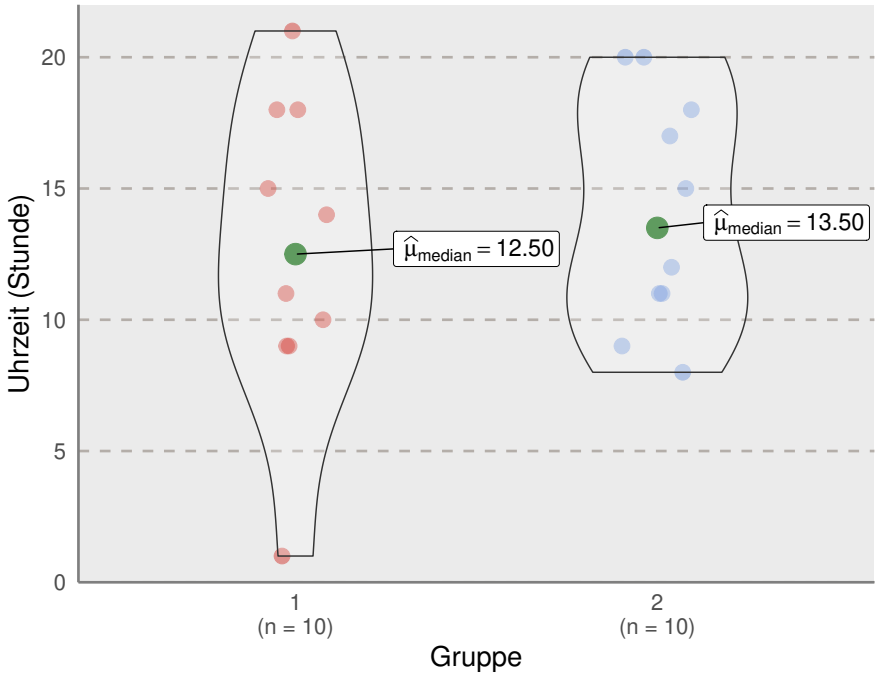


Abbildung 4.21: Verteilung der Stunde, zu dem die Teilnehmer mit der Studie angefangen haben.

HISTORIE Die asynchrone Natur der Studie macht zeitliche Effekte auf die Ergebnisse unwahrscheinlicher, da keine Uhrzeit und kein Tag der Bearbeitung vorgeschrieben war. Ein Vergleich zwischen beiden Gruppen (siehe Abbildung 4.21) zeigt bis auf einen nächtlichen Ausreißer auch keine erkennbaren Unterschiede.

SELEKTION Während die Einteilung in die beiden Gruppen zwar zufällig war, kann es natürlich auch eine zufällig ungleiche Zuordnung gegeben haben — diese Frage haben wir hauptsächlich in Unterabschnitt 4.7.1 geklärt und konnten unter Verwendung des Mann-Whitney-

26 Wir benutzen dabei teilweise die in der Vorlesung *Softwaretechnik* von Prof. Dr. Rainer Koschke verwendeten Übersetzungen der englischen Begriffe.

U-Test keine signifikanten Unterschiede erkennen. Da eine manuelle Betrachtung von z. B. Abbildung 4.9 und Abbildung 4.10 jedoch schon einige Unterschiede aufzeigt, können wir Selektionseffekte hier nicht ausschließen.

WIEDERHOLTES TESTEN Um etwa Lerneffekte zu vermeiden[27], haben wir eine randomisierte Zuweisung zu den zwei Reihenfolgen umgesetzt. Es hätte natürlich immer noch sein können, dass z. B. die Gruppe mit dem Dashboard zuerst besser die Projektstruktur lernen konnte als die mit SEE — in Unterabschnitt 4.7.2 wäre das auch durchaus ein möglicher Grund für den signifikanten Unterschied, in den anderen untersuchten Bereichen gab es jedoch keinen solchen Unterschied zwischen den beiden Gruppen.

SÄTTIGUNG Möglicherweise ändert sich der Zustand der Teilnehmer nach einer bestimmten Anzahl Aufgaben, falls diese z. B. ermüden oder keine Lust mehr auf den Rest der Studie haben und diese möglichst schnell hinter sich bringen wollen. Die hohe durchschnittliche Korrektheit der letzten Aufgabe (siehe Abbildung 4.12f) widerspricht dieser Vermutung, doch selbst wenn die „Sättigung" der Teilnehmer eine Rolle spielen sollte, wird dies ebenfalls durch die randomisierte Aufteilung in die zwei Gruppen adressiert.

Eine Bedrohung der Validität, welche wir nicht ausschließen können, ist, dass sich die Sättigungseffekte unterschiedlich auf SEE bzw. das Dashboard auswirken können. Bei SEE handelt es sich bspw. um eine vergleichsweise eher unkonventionelle Art der Softwarevisualisierung — man könnte spekulieren, dass Teilnehmer bei SEE schneller als beim Dashboard überfordert sind und die Geduld verlieren, wodurch z. B. Teilnehmer der Gruppe, in der SEE zuerst an der Reihe war, besser abschneiden könnten als die bereits durch das Dashboard „vorbelastete" andere Gruppe. Da in dieser Hinsicht keine zu großen Unterschiede in den Gruppen beobachtet werden konnte, wirkt das weiterhin eher unwahrscheinlich, kann jedoch wie gesagt nicht ausgeschlossen werden.

EXPERIMENTATOREINFLUSS Da mich viele Teilnehmer persönlich kennen und wissen, dass ich meinen SEE-Teil mit dem Dashboard-Teil vergleichen will, um meine Implementierung zu evaluieren, wäre es möglich, dass manche davon SEE bewusst oder unbewusst z. B. besser bewerten als das Dashboard. Dies wirkt jedoch eher unwahrscheinlich oder zumindest scheint es keine bedenklichen Auswirkungen gehabt zu haben, da das Dashboard letztendlich besser bewertet wurde (siehe Unterabschnitt 4.7.4). Es besteht also höchstens die Möglichkeit, dass bestimmte Teilnehmer explizit ein negatives Ergebnis dieser Studie

27 Es könnte ja sein, dass nach den ersten drei Aufgaben (bevor gewechselt wird) die Projektstruktur von den Teilnehmenden bereits gut verinnerlicht wurde, was die verbleibenden Aufgaben leichter macht.

bewirken wollten, ich hoffe aber aus mehreren Gründen, dass dies nicht der Fall ist.

SONSTIGE STÖRFAKTOREN Es wäre denkbar, dass manche Ordner übersichtlicher in SEE und andere übersichtlicher im Dashboard sind. Um dieses Problem zu addressieren, wird in den beiden Gruppen durch die Änderung der Reihenfolge der beiden Programme auch bestimmt, welche der Ordner mit welchem Programm untersucht wird, daher wurde jeder der Ordner von 50% der Teilnehmern mit SEE und von den anderen 50% mit dem Dashboard bearbeitet.

Außerdem gibt es noch ein paar relevante Unterschiede zwischen SEE und dem Dashboard, die teilweise auch schon in Abschnitt 4.1 erwähnt wurden:

- Einer der größten relevanten Unterschiede liegt darin, dass im Dashboard Dateien erst separat geöffnet werden müssen, um an die maximale Zeilennummer zu kommen, während die Größe in SEE zumindest visuell abgeschätzt werden kann. Dies ist vermutlich auch einer der Gründe für den signifikanten Unterschied in der Geschwindigkeit (siehe Unterabschnitt 4.7.3) bei den Aufgaben, die das Finden der größten Dateien erfordern.
- Im Dashboard lässt sich die exakte Zeilennummer ermitteln, während dies in SEE (zumindest im Rahmen der Evaluation) nicht möglich war. Hierdurch ließe sich eventuell der signifikante Unterschied bei der Korrektheit in Unterabschnitt 4.7.2 erklären.
- Im Dashboard werden die Code-Smell-Zahlen direkt angezeigt, während in SEE erst über den entsprechenden Knoten gehovert werden muss. Dies könnte dem Dashboard einen leichten zeitlichen Vorteil verschaffen.
- Im Dashboard lassen sich Dateinamen kopieren, während diese in SEE abgeschrieben werden müssen. Dies könnte ebenfalls einen leichten zeitlichen Vorteil für das Dashboard bedeuten.

4.8.2 *Externe Validität*

An dieser Stelle befassen wir uns mit einigen Bedrohungen der externen Validität, also der Frage, ob diese Studie repräsentativ für die Allgemeinheit[28] ist.

REPRÄSENTANZ Wenn man sich die Ergebnisse des Fragebogens in Unterabschnitt 4.7.1 ansieht, fällt z. B. als Problem auf, dass 100% der Teilnehmer männlich sind, obwohl der Frauenanteil in der Industrie eher bei ca. 17% liegt [Bit19]. Außerdem haben wir einen ungewöhnlich

28 Wobei mit „die Allgemeinheit“ hier speziell mögliche Nutzer von SEE bzw. dem Dashboard, also hauptsächlich Personen im IT-Bereich gemeint sind.

hohen Anteil an Studenten, die unter Umständen auch nicht repräsentativ für die Allgemeinheit sind. Von den tatsächlich angestellten Softwaremitarbeitern (die wahrscheinlich eher die Nutzerbasis von SEE und dem Dashboard repräsentieren) waren die meisten Angestellte von AXIVION, dort ist also die Frage, ob AXIVION-Mitarbeiter repräsentativ für IT-Arbeiter in diesem Bereich sind. Schließlich kommt noch hinzu, dass die bestehende Erfahrung mit dem Dashboard- und SEE auch nicht dem normalen Verhältnis der zu erwartenden Nutzer entspricht (siehe 4.7d und 4.7e). Besonders der hohe Anteil an SEE-Entwicklern, was dem Einbinden von Teilnehmenden des Bachelorprojekts SEA geschuldet war, dürfte unrepräsentativ sein.

REAKTIVITÄT Laut diesem nach einem auch *Hawthorne-Effekt*[29] genannten Phänomen [Lan57] verhalten sich Personen unter Beobachtung oft anders als in realen Situationen. Da hier eine Online-Studie ohne direkte Beobachtung durchgeführt wurde, dürfte dieser Effekt etwas geringer ausfallen, doch auch in Online-Fragebögen ohne aktiven menschlichen Beobachter kann dieser Effekt auftreten [Eva+10]. Wir können daher nicht ausschließen, dass die Repräsentativität dieser Studie durch einen solchen Effekt vermindert wurde.

4.9 ZWISCHENFAZIT

Wir wollten durch diese Studie unsere Forschungsfrage „Eignen sich Code-Cities besser als traditionelle tabellarische Darstellungsformen, um Code-Smells übersichtlich zu visualisieren?“ beantworten. Dies ist unter Verwendung der Ergebnisse aus Abschnitt 4.7 nicht eindeutig machbar, wir müssen eine etwas differenziertere Antwort als nur „Ja“ / „Nein“ geben. Im Punkt der Geschwindigkeit und des empfundenen Aufwands war SEE zwar signifikant besser als das Dashboard, aber sowohl in der Korrektheit der Antworten als auch in der Usability, gemessen durch den SUS, war das Dashboard signifikant besser. Hieraus lässt sich die Vermutung aufstellen, dass sich SEE eher für eine schnelle, übersichtliche Ansicht eignet, während das Dashboard besser für detaillierte Analysen geeignet sein könnte, in denen exakte Zahlen benötigt werden.

Das Verbesserungspotenzial von SEE bzw. der Code-Smell-Anzeige[30] zeigt sich vor allem auch unter Betrachtung des SUS-Scores von Unterabschnitt 4.7.4. Abgesehen davon, dass dieser signifikant unterhalb

29 Benannt nach einer von Roethlisberger u. a. [Roe+39] durchgeführten Studie in der Hawthorne-Fabrik, in welcher sich die Arbeiter unter Beobachtung produktiver verhalten haben, was fälschlicherweise als Effekt einer veränderten Beleuchtung interpretiert wurde.

30 Da der SUS nur nach einem System, also SEE, fragt, können wir hier nicht wissen, ob z. B. die Steuerung von SEE oder die Code-Smell-Anzeige die Bewertung am meisten beeinflusst hat. Die Kommentare in Unterabschnitt 4.7.6 legen eine Kombination aus beidem nah.

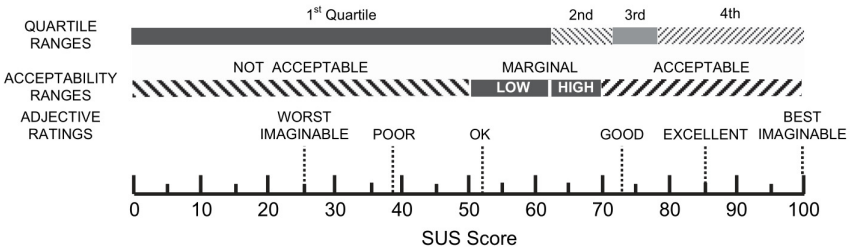


Abbildung 4.22: Ein von Bangor, Kortum und Miller [BKM08, S. 592] erstelltes Diagramm, welches den SUS-Score in Akzeptanzbereiche einteilt und Adjektive zuweist.

des SUS-Scores des Dashboards liegt, ist dieser unter 70, was laut Bangor, Kortum und Miller [BKM08, S. 592] zwischen „akzeptabel“ und „inakzeptabel“ liegt — laut der gleichen Quelle sind Produkte mit einem Score unterhalb von 70 „candidates for increased scrutiny and continued improvement and should be judged to be marginal at best“. Dementsprechend sollte, zusätzlich zu den in Unterabschnitt 4.7.6 angesprochenen Verbesserungsmöglichkeiten, besonders an der Usability von SEE gearbeitet werden — wenn sich die von Lewis und Sauro [LS09] herausgestellten zwei Faktoren auf eine ähnliche Weise interpretieren lassen, ist die *Learnability* (also mit welcher Leichtigkeit das System erlernt werden kann) bei SEE wenigstens in einem akzeptablen Bereich, wenn auch niedriger als die beim Dashboard.

Außerdem sorgte insbesondere ein höheres Alter bzw. ein höherer Schulabschluss zu einem schlechteren Ergebnis des SUS in SEE und einer geringeren empfundenen Komplexität bei den Aufgaben des Dashboards. Dies könnte implizieren, dass sich Teilnehmer mit mehr Erfahrung in der Bedienung von Software eher mit dem Interface des Dashboards gut zurechtfinden, da dieses dem von anderer Software ähnelt und gewöhnte Konzepte verwendet. SEE hingegen hatte, wie bereits angesprochen, eine eher unkonventionelle Bedienung und war daher unter Umständen weniger gut benutzbar, insbesondere für Teilnehmer, die wegen ihrer Erfahrung bereits mit dem Dashboard sehr gut zurechtkamen. Um dieses Problem zu beheben, kann es z. B. sinnvoll sein, ein besseres Hilfesystem einzuführen, welches die Bedienkonzepte von SEE anschaulicher erklärt.

Die tatsächliche Antwort auf die Forschungsfrage (mit den oben genannten Vorbehalten) lautet also: „Code-Cities eignen sich eher in Situationen, in denen ein schneller Überblick geschaffen werden soll, tabellarische Darstellungsformen hingegen besser für Detailanalysen.“

5

AUSBLICK

In diesem Teil schließen wir die Bachelorarbeit ab, indem wir auf mögliche Einschränkungen und Verbesserungsmöglichkeiten zu sprechen kommen, zukünftig umsetzbare Ideen vorstellen und die Arbeit als Ganzes mit einem Fazit beenden.

5.1 EINSCHRÄNKUNGEN

In diesem Teil geht es hauptsächlich um Limitationen der Ergebnisse.

5.1.1 *Implementierung*

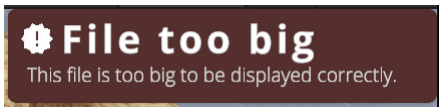


Abbildung 5.1: Fehlermeldung, die beim Laden zu großer Dateien in die Code-Windows auftritt.

BUG IM TEXTMESHPRO Es existiert ein Bug im `TextMeshPro`, bei dem die Textdarstellung ab einer gewissen Größe und unter Verwendung von Markierungen und einigen anderen Aspekten des Rich-Text-Systems (siehe Abschnitt 3.2) nicht mehr funktioniert. Der Entwickler wurde darauf angesprochen und ist sich des Problems bewusst[1], wird dieses aber wohl nicht in nächster Zeit beheben. Aktuell wird beim Öffnen von zu großen Dateien[2] eine Fehlermeldung angezeigt (siehe Abbildung 5.1).

Der Entwickler des `TextMeshPros` schlägt vor, das Text-GameObject (also den entsprechenden Teil des Code-Windows) in mehrere kleine Teile aufzuspalten. Dies würde das Problem lösen und würde zusätzlich die Performance verbessern, würde aber einen erheblichen Aufwand bedeuten, da der Code des Code-Windows stark von der Annahme abhängt, dass nur ein `TextMeshPro` existiert. Eine weitere Möglichkeit wäre unter Umständen, insgesamt zwei `TextMeshPros` zu haben: Eines

1 Der entsprechende Forumsthread mit mehr Informationen: https://forum.unity.com/threads/indexoutofboundsexception-when-using-u-or-mark-with-vertices-count-bigger-than-65535.1124647/ (letzter Abruf: 15.09.2021)

2 Das sind solche mit mehr als $\frac{2^{16}}{4} - 2$ Zeichen, wobei das Rich-Text-System (also z. B. Syntax-Highlighting und Markieren von Code-Smells) ebenfalls an diesem Limit zehrt. Beim Testen trat dieser Fehler ungefähr bei Dateien mit einer Zeilenanzahl von 2000 auf.

für die Darstellung des reinen Quelltextes ohne Markierungen (dort tritt der Fehler ja nicht auf) und eines als *Overlay*, welches nur die Markierungen enthält und über dem Text halbtransparent darüberliegt.

ÜBERLAGERUNGEN Ein weiterer Punkt, der von den Teilnehmern angesprochen wurde (siehe Unterabschnitt 4.7.6), ist der, dass das Ablesen von Code-Smell-Zahlen bzw. Dateinamen oft wegen der gegenseitigen Überlagerungen schwierig war. Wie dort angesprochen wurde, wäre es hier vermutlich eine gute Idee, beim Hovern über einen Knoten nicht nur die aktuellen Informationen zu vergrößern, sondern alle benachbarten Texte auszublenden.

NUTZERINTERFACE ZUR LAUFZEIT Kurz nachdem ich die Implementierung fertiggestellt hatte, wurde in SEE eine größere Änderung eines anderen Studenten für seine Bachelorarbeit integriert — mit dieser war es möglich, die Einstellungen für die Code-City, die bis dahin nur über den Unity-Inspektor möglich waren, auch im Spiel selbst über ein dediziertes Nutzerinterface zu tätigen. Hier wäre es sinnvoll, die Konfigurationsmöglichkeiten der beiden Visualisierungen (siehe Abschnitt 3.4) in diese GUI zu überführen. Außerdem ist der Abruf der Code-City-Metriken aktuell darauf ausgelegt, im Unity-Editor ausgeführt zu werden, nicht während des Spiels, z. B. erscheint nicht wie bei den Code-Windows (siehe Abschnitt 3.2) eine entsprechende Benachrichtigung über den Ladevorgang.

5.1.2 *Evaluation*

Die „Threats to Validity“ haben wir bereits in Abschnitt 4.8 ausführlich diskutiert, ein dort bereits angesprochener Punkt sollte hier aber nochmal besonders hervorgehoben werden: Es ist gut möglich, dass der gemessene signifikante Unterschied zugunsten SEEs sowohl bei der Zeit (siehe Unterabschnitt 4.7.3) als auch bei der empfundenen Komplexität (siehe Unterabschnitt 4.7.4) komplett verschwinden würde, wenn das Axivion-Dashboard eine Spalte zum Sortieren nach der Anzahl der Codezeilen hätte. Es gibt jedoch auch einen entgegengesetzt wirkenden Effekt, nämlich, dass die Dateinamen bei SEE abgetippt werden mussten[3], während diese beim Dashboard einfach kopiert werden konnten. Genauso gab es bei SEE keine exakten Angaben zu der Anzahl an Codezeilen, beim Dashboard hingegen schon — dies wäre eine mögliche Erklärung für das signifikant bessere Ergebnis beim Dashboard für die Korrektheit (siehe Unterabschnitt 4.7.2).

Untersucht man alleine die Aufgaben 3 und 6, bei denen dieser Effekt nicht auftreten konnte (die in ihrer Aussagekraft z. B. wegen der vergleichsweise kurzen Bearbeitungsdauer allerdings auch beschränkt

3 Bei Setups mit nur einem Monitor musste vermutlich zusätzlich auch noch zwischen den beiden Fenstern hin- und hergewechselt werden, während abgeschrieben wurde.

sind), würde man keinen signifikanten Unterschied zwischen dem Dashboard und SEE erkennen und könnte keine der fünf Hypothesen aus Abschnitt 4.2 verwerfen. Es wäre also vermutlich lohnenswert, ein weiteres Experiment durchzuführen, welches unter Verwendung entweder eines anderen Tools mit Sortiermöglichkeit für Codezeilen oder anderen Aufgabetypen[4] versucht, die Ergebnisse dieser Studie zu replizieren.

5.2 WEITERE IDEEN

An dieser Stelle wollen wir noch stichpunktartig einige Ideen für die Implementierung durchgehen, durch die die in dieser Arbeit erstellten Grundlagen ausgebaut werden können.

- Klone könnten in einer Side-By-Side-Ansicht dargestellt werden.[5]
- Eine Auflistung von Code-Smells pro Komponente, ähnlich wie in Abbildung 1.4, wäre möglich.
- Rechtfertigungen für Code-Smells (im AXIVION-Dashboard „*Justification*“ genannt) lassen sich zwar mit der Datenabrufkomponente aus Abschnitt 2.1 abrufen, es fehlt aber noch eine Möglichkeit, diese anzuzeigen oder zu erstellen.
- Ebenso kann das Ausblenden von Code-Smells (in Abbildung 1.4 „*Suppress*“ genannt) hilfreich sein, da oft nicht alle relevant sind.
- Momentan können aus dem Dashboard alle Metriken und Code-Smell-Daten abgerufen werden, nur die GXL-Datei mit den Informationen über die Knoten und Kanten selbst sowie die Quelldateien müssen noch lokal existieren. Praktisch wäre es, diese letzten beiden ebenfalls vom Dashboard abrufen zu können — dies hätte besonders im kollaborativen Kontext praktische Implikationen, denn wenn z. B. ein Mitarbeiter den anderen eine bestimmte Version zeigen will, müsste dieser lediglich eine URL (oder ein Datum) senden, welche diese dann in SEE eingeben, anstatt GXL-, CSV- und Quellcode untereinander austauschen zu müssen. Während sich Knoten- und Kantendaten prinzipiell bereits über die API des Dashboards abrufen lassen, ist dies bei dem Quellcode (der für die Code-Windows benötigt wird) noch nicht der Fall.

4 Dabei muss natürlich darauf geachtet werden, dass eine geeignete und vergleichbare Aufgabe gewählt wird, z. B. hätte eine Aufgabe der Form „Finde die Klassen mit den meisten Code-Smells“ wegen der Sortierfunktion des Dashboards einen großen Nachteil für SEE.

5 Als Teil der Darstellung in den Code-Windows (siehe Unterabschnitt 3.2.2) werden Klone zwar markiert, jedoch wird immer nur ein Klonfragment angezeigt, nicht mehrere nebeneinander im Vergleich.

- Eine weitere Idee wäre das automatische Anwenden von Codeänderungen, die das Problem beheben würden. Da das Dashboard jedoch diese Vorschläge nicht anbietet, wäre das deutlich komplizierter.

5.3 ABSCHLUSSFAZIT

Wir haben in Kapitel 1 die Motivation und Forschungsfrage beschrieben, die „Eignen sich Code-Cities besser als traditionelle tabellarische Darstellungsformen, um Code-Smells übersichtlich zu visualisieren?“ lautete. Um diese beantworten zu können, haben wir in Kapitel 2 den Abruf der Code-Smell-Daten vom Axivion-Dashboard und dessen Visualisierung in den Code-Cities und Code-Windows geplant und anschließend in Kapitel 3 die Umsetzung dieser drei Teile in SEE beschrieben. Nachdem wir nun also eine Grundlage zur Beantwortung der Forschungsfrage geschaffen haben, konnten wir uns in Kapitel 4 um die Auswertung der Visualisierung in den Code-Cities kümmern, indem wir eine Vergleichsstudie zwischen SEE und dem Axivion-Dashboard mit $n = 20$ Teilnehmern durchgeführt haben. Dabei haben wir mit einem Signifikanzniveau von $\alpha = 0,05$ in den Aspekten „Korrektheit“ (Anteil korrekter Antworten) und „Usability“ (durch den SUS) ein besseres Ergebnis beim Dashboard, bei den Aspekten „benötigte Zeit“ und „empfundener Aufwand“ hingegen ein besseres Ergebnis bei SEE. Der Aspekt „empfundene Komplexität“ ergab keinen signifikanten Unterschied. Weiterhin fanden wir heraus, dass bestehende Erfahrung mit dem jeweiligen System einen wesentlich stärkeren Einfluss auf die Bearbeitung der Aufgaben beim Dashboard als bei SEE hatte, insbesondere beim empfundenen Aufwand. Das Alter bzw. der Schulabschluss eines Teilnehmers hatte ebenso einen signifikanten Einfluss, ältere Teilnehmer hatten hierbei die Usability von SEE eher schlechter und die Komplexität des Dashboards eher geringer eingeschätzt. Abschließend haben wir in Kapitel 5 Einschränkungen der Implementierung und Evaluation sowie weitere Ideen zum Ausbau der in dieser Bachelorarbeit erstellten Grundlagen gegeben.

Die Forschungsfrage haben wir beantwortet mit „Code-Cities eignen sich eher in Situationen, in denen ein schneller Überblick geschaffen werden soll, tabellarische Darstellungsformen hingegen besser für Detailanalysen.“ Es existiert meiner Meinung nach auf jeden Fall viel Potenzial weiterzuforschen: Es wurden lohnenswerte Verbesserungsvorschläge zu SEE genannt, die z. B. die Usability verbessern könnten, aber auch eine Verbesserung von SEE im Hinblick auf Detailanalysen scheint sinnvoll zu sein — auch zukünftige Evaluationen, welche die in diesem Abschnitt angesprochenen Einschränkungen ausbessern, würden bei der Entwicklung zukünftiger Verbesserungen im Gebiet der Visualisierung von Code-Smells in Code-Cities helfen.

A

GLOSSAR

Axivion-Dashboard Ein kommerzielles Produkt als Teil der Axivion-Suite, das Informationen zu Code-Smells von Softwareprojekten in einem Web-Interface zusammenträgt. 5–12, 17–20, 26, 27, 29–31, 33, 34, 37, 40, 45, 51–54, 63, 64, 70, 72, 80–82, 84, 89, 91, 92, 95

Abhängigkeitsgraph Ein in SEE visualisierter Graph aus typisierten Knoten und Kanten, die zusätzliche Attribute (z. B. Metriken) enthalten können. Die Typen können dabei beliebig sein, wodurch SEE gut generalisierbar ist. 2, 8, 9, 20, 21, 24

Architektur-Verletzung Eine Inkonsistenz zwischen Architekturmodell und Quellcode. 6, 19, 40–42, 45, 89, *siehe* Code-Smell

Argumente der Maxima (argmax) Stellen einer Funktion, an der die Funktionswerte maximiert werden. 42

Blattknoten Ein Knoten in einem Baum, welcher keine weiteren Kindknoten besitzt. 8, 10, 22, 24, 27

Code-City In der Code-City-Metapher werden Softwarekomponenten durch Gebäude in einer Stadt repräsentiert, wobei die Eigenschaften dieser Gebäude verschiedene Metriken der Software ausdrücken können — z. B. könnte die Höhe eines Gebäudes der Anzahl der Codezeilen entsprechen. 2–5, 7–10, 16, 19, 20, 22, 23, 27, 29–31, 33, 34, 73, 77, 78, 80, 82, 86, 89, 91

Code-Smell „Bestimmte Strukturen im Code [...], die es nahelegen [...], dass ein Refactoring angebracht ist“ [Fow20]. 3–10, 12, 16–24, 26, 27, 29, 30, 33, 34, 40, 41, 56, 62, 72, 73, 76, 77, 79–84, 90, 91

Code-Window Eine Darstellung von Quellcode in SEE, die der Darstellung in IDEs sehr ähnelt. 3, 5, 7, 9, 10, 15–18, 26, 27, 30, 31, 34, 79–82, 91, 93

Convenience-Sampling Eine Auswahlstrategie für Teilnehmer einer Studie, bei der diese nach Einfachheit (statt z. B. per Zufall) gewählt werden. 33

Ebene Eine Ebene in einem Graphen enthält alle Knoten, die einen bestimmten Abstand zum Wurzelknoten haben. Die oberste Ebene enthält dabei alleine den Wurzelknoten, die nachfolgende Ebene alle direkten Kinder des Wurzelknotens, usw. 9, 24, 31

Empirische Standardabweichung (s) Die Standardabweichung einer Stichprobe, welche Variable X misst. Wird wie folgt berechnet (n ist die Anzahl der Datenpunkte):

$$\frac{1}{n-1} \cdot \sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2}$$

50, 57–59, 61–65

Erosion Synonym zu Code-Smells. 6, 8–10, 22–25, 27, 46, 91, *siehe* Code-Smell

Fisher-Pitman-Randomisierungstest Ein Permutationstest, welcher überprüft, ob zwei unabhängige Stichproben aus der gleichen Population stammen. Dabei werden keine Anforderungen an die Populationen gemacht. 58

GameObject Ein Objekt in Unity, das als Zusammenhalt von einzelnen Komponenten fungiert und z. B. eine Position, ein Aussehen und ein Verhalten haben kann. 9, 16, 79, 85

Gottklasse Eine Klasse, welche zu groß ist oder für zu viele Dinge zuständig ist, was wiederum die Lesbarkeit bzw. die Übersicht über den Quellcode mindert. 40

Innerer Knoten Ein Knoten in einem Baum, welcher Kindknoten besitzt. 8–10, 22, 27

Issue Eine konkrete Ausprägung eines Code-Smells im Axivion-Dashboard. 12, 15, 27, 30, 31, 84

Issue-Hotspot Stelle im Projekt, an der überproportional viele Issues existieren. 31

Kendall-Tau-Korrelationskoeffizient (τ_b) Ein rangbasierter Korrelationskoeffizient, der zwei ordinalskalierte oder intervallskalierte Variablen auf einen Zusammenhang in Form einer ordinalen Assoziation testet. Der Wertebereich des Koeffizienten liegt dabei bei $[-1; 1]$, wobei ein positiver Wert eine positive Korrelation, 0 keine Korrelation und ein negativer Wert eine negative Korrelation impliziert. Die Ausprägung des Betrags entspricht dabei dem Ausmaß der Korrelation. 69–71, 95

Klon Zwei unterschiedliche Stellen im Quellcode, die ähnlichen Code enthalten. 2, 6, 8, 18, 19, 41, 81, 91, *siehe* Code-Smell

Likert-Skala Eine Skala, die in der Psychometrie verwendet wird, in welcher eine Antwort auf einer linearen Skala von „Ich stimme zu“ bis „Ich stimme nicht zu“ gegeben wird. 35, 37, 38

Mann-Whitney-U-Test Ein Test, welcher zwei unabhängige Stichproben mit Ordinalskalen vergleicht, um Unterschiede zu finden. Die Stichproben müssen aus Populationen stammen, die die gleiche Verteilung haben. 49, 50, 52, 54, 57–59, 61, 62, 64, 65, 74

Metrik-Verletzung Ein Metrikwert ist außerhalb eines definierten Bereiches für eine Codemetrik. 6, 7, 19, 41–43, 46, 89, *siehe* Code-Smell

`MonoBehaviour` Ein bestimmter Klassen-Typ, den alle Unity-Skripte haben müssen. Diese dienen dann als Komponente eines GameObjects und werden (vereinfacht gesagt) während des Spiels ausgeführt. 16, 85

Pearson-Korrelationskoeffizient (r) Hiermit kann eine lineare Korrelation zwischen zwei Variablen X und Y gemessen werden:

$$r = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y}$$

Cov ist hierbei die Kovarianz und σ die Standardabweichung. 41, 69, 71, 72, 89, 95

Post-Study Ein Fragebogen, welcher erst nach allen Aufgaben abgefragt wird, also am Ende der Studie. 32, 34, 37–39, 63

Post-Task Ein Fragebogen, welcher nach jeder einzelnen Aufgabe abgefragt wird. 32, 34, 37, 48, 63, 65

Shader In Unity sind Shader auf der GPU laufende, in der C-ähnlichen Sprache HLSL geschriebene Programme, die bestimmen, wie Objekte des Spiels in Bezug auf einzelne Pixel dargestellt werden. 23

Singleton Ein Software-Entwurfsmuster, in dem eine nach diesem Muster entworfene Klasse nur eine Instanz besitzen darf. 16

Stil-Verletzung Andere Arten von Problemen, hauptsächlich die Verletzung von Code-Konventionen. 6, 19, 30, 40–43, 46, 48, 89, *siehe* Code-Smell

Symmetrische Gruppe ($\mathfrak{S}$) Die Gruppe aller Permutationen einer Menge X, wobei diese Menge in unserer Notation als Index geschrieben wird, also z. B. $\mathfrak{S}_X$. 42

Syntax-Highlighting Das farbliche Hervorheben von Stellen im Quellcode basierend auf der Syntax der Programmiersprache. 3, 7, 16, 18, 79

t-Test Ein Test, welcher die Mittelwerte zweier unabhängiger Stichproben mit Intervallskalen vergleicht, um Unterschiede zu finden. Die Stichproben müssen aus normalverteilten Populationen mit annähernd gleichen Varianzen stammen. 49

`TextMeshPro` Eine Komponente in Unity in Form eines `MonoBehaviours`, das Text auf vielfältige Weisen darstellen kann, z. B. in verschiedenen Farben und Schriftarten. 16, 17, 19, 79

Token Ein einzelnes Element einer Programmiersprache, z. B. in C# ein Schlüsselwort wie `new`, oder der Name einer Variable. 16, 17

toter Code Eine Definition im Quellcode, die an keiner anderen Stelle im Programm genutzt wird. 6, 19, 41, *siehe* Code-Smell

Unity Eine Spiele-Engine, mit der man unter Verwendung der Programmiersprache C# 2D- und 3D-Spiele entwickeln kann. 2, 9, 11, 12, 15, 16, 23, 80, 84, 85, 89

Unity-Inspektor Ein Teil des Editors der Spiele-Engine *Unity*, welcher einem Nutzer die Konfiguration von Spiele-Objekten ermöglicht. Wird in SEE z. B. für die Konfiguration der Code-Cities verwendet. 9, 15, 16, 26, 27, 80, 91

Usability Beschreibt, wie gut ein (insbesondere Software-)System benutzt werden kann. 32, 35, 37–39, 49, 64, 69, 71, 74, 77, 78, 82

zyklische Abhängigkeit Mehrere unterschiedliche Stellen im Quellcode, die im Aufrufgraphen gegenseitig voneinander abhängen. 6, 19, 41, *siehe* Code-Smell

B

AKRONYME

API Eine Software-Schnittstelle, die bereitgestellt wird, um Zugriff von anderer Software zu ermöglichen. 5, 9, 11, 12, 14, 15, 17, 20, 29, 34, 81, 87

ASQ Ein post-task Fragebogen bestehend aus 3 Fragen. 32, 33, 37, 45, 46, 48, 49, 63, 65–69, 71, 93, *siehe* Post-Task

CSV Ein simples Dateiformat für tabellarische Daten, in denen Spalten mit einem Komma und Reihen mit einem Zeilenumbruch getrennt werden.. 20, 27, 44, 81, 92

GXL Ein Dateiformat für Graphen, wird z. B. in SEE zur Repräsentation von Projektgraphen verwendet. 2, 8, 10, 20, 27, 81

IDE Software, die zur Programmentwicklung verwendet wird, wie etwa *Eclipse* oder *JetBrains IntelliJ*. 3–5, 7, 27, 83, 91

JSON Ein menschenlesbares Datenformat, das z. B. gerne in REST-APIs verwendet wird. 11, 12, 14, 15

LOD Ein Mechanismus in 3D-Anwendungen, bei dem der Detailgrad eines Objekts reduziert wird (oder dieses sogar komplett ausgeblendet wird), je weiter der Spieler davon entfernt ist. 23, 24

NASA-TLX Ein post-task Fragebogen bestehend aus 6 Fragen, entwickelt von der NASA. 35, *siehe* Post-Task

PSSUQ Ein post-study Fragebogen bestehend aus 16 Fragen. 38, *siehe* Post-Study

QUIS Ein kostenpflichtiger post-study Fragebogen, der je nach Version 41 oder 122 Fragen beinhaltet. 38, *siehe* Post-Study

REST Eine im Web oft verwendete Art von API, in der meist über HTTP-Anfragen an bestimmte Endpunkte Informationen abgerufen oder modifiziert werden können. 5, 11, 12, 29, 87

SEA Ein Bachelorprojekt der Universität Bremen, in welchem das Softwareprojekt SEE weiterentwickelt wurde. 2, 33, 48, 77

SEE Eine interaktive Visualisierung von Software, welche die *Code-City*-Metapher verwendet und einen kollaborativen Multiplayer

über verschiedene Plattformen[1] hinweg ermöglicht. 2, 3, 5–11, 15–17, 19–21, 24–27, 29–35, 37, 40–43, 45, 46, 48, 49, 51–65, 68–78, 80–83, 86, 87, 89, 91–93, 95

SEQ Ein post-task Fragebogen bestehend aus einer einzelnen Frage. 37, 48, *siehe* Post-Task

SMEQ Ein post-task Fragebogen bestehend aus einer einzelnen Frage, die eine Antwort von 0–150 erlaubt. 35–37, 92, *siehe* Post-Task

SUMI Ein kostenpflichtiger post-study Fragebogen bestehend aus 50 Fragen. 38, *siehe* Post-Study

SUS Ein verbreiteter post-study Fragebogen bestehend aus 10 Fragen. 32, 34, 38, 39, 46, 49, 63, 64, 71, 72, 77, 78, 82, 89, 92, 93, 95, *siehe* Post-Study

UME Ein unkonventioneller post-task Fragebogen, in welchem die Schwierigkeit einzelner Aufgaben zueinander ins Verhältnis gesetzt werden sollen. 35, *siehe* Post-Task

XPath Eine Sprache, mit der sich Knoten in einem XML-Dokument (z. B. aus dem HTML-Dokument einer Website) herausfiltern lassen. 15

1 Neben Desktop- und Touch-Umgebungen noch Virtual Reality (z. B. *Valve Index*) und Augmented Reality (z. B. *Microsoft HoloLens*)

C

ANGEHÄNGTE DATEIEN

Bezeichnung	Beschreibung
`SEE.zip`	Die gebaute Executable von SEE, die auch in der Evaluation den Teilnehmern gegeben wurde. Funktioniert nur auf Windows-Rechnern und eignet sich nur zum Testen der Visualisierung in den Code-Cities.
`DashboardUseCases.txt`	Eine anonymisierte Auflistung der Antworten der AXIVION-Mitarbeiter auf mögliche Anwendungsfälle für das AXIVION-Dashboard.
`findCorrelation.py`	Ein Python-Skript, das die Pearson-Korrelationskoeffizienten zwischen den Architektur-Verletzungen, Metrik-Verletzungen und Stil-Verletzungen berechnet. Die Kommentare zu Beginn der Datei sind zu beachten.
`Common-Eval.xlsx`	Die Studie als XLSForm. Kann direkt z. B. in KoBoToolbox (`https://www.kobotoolbox.org`) importiert werden oder in einem Tabellenverarbeitungsprogramm betrachtet werden. Es handelt sich um eine Kombination aus der SEE→Dashboard und Dashboard→SEE-Version — diese beiden werden durch das unsichtbare Feld mit dem Label „HIER AUFHÖREN“ getrennt.
`Dashboard_SEE.csv`	Enthält die Rohdaten der Teilnehmer. `dsus` bzw. `ssus` beschreibt die SUS-Fragen für das AXIVION-Dashboard bzw. SEE, `dexpl` und `sexpl` sind jeweils Verbesserungsvorschläge. Am Ende wurden Felder mit `_c`-Suffixen angefügt, diese bezeichnen die Korrektheit der Antwort für die jeweilige Frage mit `T` für korrekte und `F` für falsche Antworten.
`SEE-Solution.txt`	Enthält die Lösungen für die SEE-Aufgaben der Evaluation. Diese wurden manuell unter Verwendung des Unity-Editors ermittelt.
`Dashboard-Solution.txt`	Enthält die Lösungen für die Dashboard-Aufgaben der Evaluation. Diese wurden automatisch unter Verwendung des `findDash.py`-Skriptes ermittelt.

Bezeichnung	Beschreibung
`findDash.py`	Ein Python-Skript, das die Lösungen für die Dashboard-Aufgaben berechnet, indem die Code-Smell-Zahlen für die relevanten Dateien der Top-Level-Ordner berechnet werden. Die Kommentare zu Beginn der Datei sind zu beachten.
`calc_results.R`	Ein R-Skript, das alle relevanten Ergebnisse der Auswertung berechnet und ausgibt. Es werden ebenfalls alle Diagramme des Abschnitts als PDF-Dateien im gleichen Ordner erzeugt.

D

ABBILDUNGSVERZEICHNIS

E

TABELLENVERZEICHNIS

F

LITERATUR

[Bit19] Bitkom Research. *IT-Fachkräfte: Nur jeder siebte Bewerber ist weiblich*. 6. März 2019. URL: http://web.archive.org/web/20210812014239/https://www.bitkom.org/Presse/Presseinformation/IT-Fachkraefte-Nur-jeder-siebte-Bewerber-ist-weiblich (besucht am 09. 09. 2021). Archivierte Version: http://web.archive.org/web/20210812014239/https://www.bitkom.org/Presse/Presseinformation/IT-Fachkraefte-Nur-jeder-siebte-Bewerber-ist-weiblich.

[BKM08] Aaron Bangor, Philip T. Kortum und James T. Miller. „An Empirical Evaluation of the System Usability Scale“. In: *International Journal of Human–Computer Interaction* 24.6 (29. Juli 2008), S. 574–594. ISSN: 1044-7318. DOI: 10.1080/10447310802205776.

[Bor+08] „Testmethoden für Kardinaldaten“. In: *Kurzgefasste Statistik für die klinische Forschung: Leitfaden für die verteilungsfreie Analyse kleiner Stichproben*. Hrsg. von Jürgen Bortz u. a. Springer-Lehrbuch. Berlin, Heidelberg: Springer, 2008, S. 227–255. ISBN: 978-3-540-75738-2. DOI: 10.1007/978-3-540-75738-2_4.

[Bri04] Scott Densmore Brian Button. *Why Singletons are Evil*. 25. Mai 2004. URL: https://docs.microsoft.com/en-us/archive/blogs/scottdensmore/why-singletons-are-evil (besucht am 28. 08. 2021). Archivierte Version: http://web.archive.org/web/20210715184717/https://docs.microsoft.com/en-us/archive/blogs/scottdensmore/why-singletons-are-evil.

[Bro96] John Brooke. „SUS: A 'Quick and Dirty' Usability Scale“. In: *Usability Evaluation In Industry*. CRC Press, 1996, S. 189–194. ISBN: 978-0-429-15701-1.

[CDN88] John P. Chin, Virginia A. Diehl und Kent L. Norman. „Development of an instrument measuring user satisfaction of the human-computer interface“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '88. New York, NY, USA: Association for Computing Machinery, 1. Mai 1988, S. 213–218. ISBN: 978-0-201-14237-2. DOI: 10.1145/57167.57203.

[CSG63] Donald Thomas Campbell, Julian C. Stanley und Nathaniel Lees Gage. *Experimental and quasi-experimental designs for research.* Boston, MA, US: Houghton, Mifflin und Company, 1963. ix, 84. ISBN: 0-395-30787-2.

[Eva+10] Rhodri Evans u. a. „Supporting Informed Decision Making for Prostate Specific Antigen (PSA) Testing on the Web: An Online Randomized Controlled Trial“. In: *Journal of Medical Internet Research* 12.3 (6. Aug. 2010), e27. ISSN: 1438-8871. DOI: 10.2196/jmir.1305. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2956331/ (besucht am 09. 09. 2021).

[Fin06] Kraig Finstad. „The System Usability Scale and non-native English speakers“. In: *Journal of Usability Studies* (2006), S. 185–188.

[FL05] Ann Fruhling und Sang Lee. „Assessing the Reliability, Validity and Adaptability of PSSUQ“. In: *AMCIS 2005 Proceedings* (1. Jan. 2005). URL: https://aisel.aisnet.org/amcis2005/378 (besucht am 05. 08. 2021).

[Fow20] Martin Fowler. *Refactoring – Wie Sie das Design bestehender Software verbessern*. 2nd edition. mitp Verlag, 2020. ISBN: 9783958459434. URL: https://learning.oreilly.com/library/view/-/9783958459434/?ar (besucht am 01. 06. 2021).

[FR20] Walter Funk und Barbara Roegele. *Safety4Bikes. Assistenzsystem für mehr Sicherheit von fahrradfahrenden Kindern. Arbeitspaket 1: Usability-Evaluation des Gesamtsystems aus der Anforderungsperspektive von Kindern*. 1. Aug. 2020. URL: https://www.ifes.fau.de/files/2020/09/ifes-mat-3-2020-safety4bikes-arbeitspaket-1-usability-evaluation-1.pdf (besucht am 18. 08. 2021).

[Hot+08] Torsten Hothorn u. a. „Implementing a Class of Permutation Tests: The coin Package“. In: *Journal of Statistical Software, Articles* 28.8 (2008), S. 1–23. ISSN: 1548-7660. DOI: 10.18637/jss.v028.i08.

[HS88] Sandra G. Hart und Lowell E. Staveland. „Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research“. In: *Advances in Psychology*. Hrsg. von Peter A. Hancock und Najmedin Meshkati. Bd. 52. Human Mental Workload. North-Holland, 1988, S. 139–183. DOI: 10.1016/S0166-4115(08)62386-9.

[Kaw20] Yoshifumi Kawai. *UniTask v2 — Zero Allocation async/await for Unity, with Asynchronous LINQ*. 11. Juni 2020. URL: https://neuecc.medium.com/unitask-v2-zero-allocation-async-await-for-unity-with-asynchronous-linq-1aa9c96aa7dd (besucht am 06. 09. 2021). Archivierte Ver-

sion: http://web.archive.org/web/20210120103350/https://neuecc.medium.com/unitask-v2-zero-allocation-async-await-for-unity-with-asynchronous-linq-1aa9c96aa7dd.

[Kha08] Harry Khamis. „Measures of Association: How to Choose?“ In: *Journal of Diagnostic Medical Sonography* 24.3 (1. Mai 2008). Publisher: SAGE Publications Inc STM, S. 155–162. ISSN: 8756-4793. DOI: 10.1177/8756479308317006.

[Kir] Jurek Kirakowski. *How to get the SUMI service*. URL: https://sumi.uxp.ie/about/how_to_get.html (besucht am 17. 08. 2021). Archivierte Version der Seite: https://archive.vn/OWgwG.

[Kir94] Jurek Kirakowski. „The use of questionnaire methods for usability assessment“. In: *Unpublished manuscript. Recuperado el* 12 (1994).

[Lan57] Henry A Landsberger. *Hawthorne revisited: a plea for an open city.* OCLC: 61637839. Ithaca, N.Y.: Cornell University, 1957.

[Lew02] James R. Lewis. „Psychometric Evaluation of the PSSUQ Using Data from Five Years of Usability Studies“. In: *International Journal of Human–Computer Interaction* 14.3 (1. Sep. 2002), S. 463–488. ISSN: 1044-7318. DOI: 10.1080/10447318.2002.9669130.

[Lew18] James R. Lewis. „The System Usability Scale: Past, Present, and Future“. In: *International Journal of Human–Computer Interaction* 34.7 (3. Juli 2018), S. 577–590. ISSN: 1044-7318. DOI: 10.1080/10447318.2018.1455307.

[Lew91] James R. Lewis. „Psychometric evaluation of an after-scenario questionnaire for computer usability studies: the ASQ“. In: *ACM SIGCHI Bulletin* 23.1 (1991), S. 78–81. ISSN: 0736-6906. DOI: 10.1145/122672.122692.

[Lew92] James Lewis. „Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ“. In: Proceedings of the Human Factors Society. Bd. 2. 1992, S. 1259–1263.

[LS09] James R. Lewis und Jeff Sauro. „The Factor Structure of the System Usability Scale“. In: *Human Centered Design*. Hrsg. von Masaaki Kurosu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, S. 94–103. ISBN: 978-3-642-02806-9. DOI: 10.1007/978-3-642-02806-9_12.

[McG03] Mick McGee. „Usability Magnitude Estimation“. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 47.4 (1. Okt. 2003), S. 691–695. ISSN: 2169-5067. DOI: 10.1177/154193120304700406.

[McG18] Monnie McGee. „Case for omitting tied observations in the two-sample t-test and the Wilcoxon-Mann-Whitney Test“. In: *PLOS ONE* 13.7 (24. Juli 2018). Publisher: Public Library of Science, e0200837. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0200837.

[Mic20] Microsoft. *Asynchronous programming with async and await*. 6. Apr. 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (besucht am 26. 08. 2021). Archivierte Version: http://web.archive.org/web/20210514092454/https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/.

[MMP11] Sam Mclellan, Andrew Muddimer und S. Peres. „The Effect of Experience on System Usability Scale Ratings“. In: *Journal of Usability Studies* 7 (30. Nov. 2011).

[Pat21] Indrajeet Patil. „Visualizations with statistical details: The 'ggstatsplot' approach“. In: *Journal of Open Source Software* 6.61 (2021), S. 3167. DOI: 10.21105/joss.03167.

[PPP13] S. Camille Peres, Tri Pham und Ronald Phillips. „Validation of the System Usability Scale (SUS): SUS in the Wild“. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 57.1 (1. Sep. 2013), S. 192–196. ISSN: 2169-5067. DOI: 10.1177/1541931213571043.

[Roe+39] F. J Roethlisberger u. a. *Management and the worker: an account of a research program conducted by the Western Electric Company, Hawthorne Works, Chicago*. OCLC: 271822. Cambridge, Mass.: Harvard University Press, 1939.

[RR] Bernard Rummel und Eva Ruegenhagen. *System Usability Scale – jetzt auch auf Deutsch*. URL: http://web.archive.org/web/20200607035254/https://experience.sap.com/skillup/system-usability-scale-jetzt-auch-auf-deutsch/ (besucht am 17. 08. 2021). (Hier handelt es sich um eine archivierte Version, da die Originalseite mittlerweile nicht mehr verfügbar ist. Der Link zum eigentlichen übersetzten Fragebogen funktioniert jedoch auch im Original: https://experience.sap.com/files/System_Usability_Scale_A4_DE.doc).

[SD09] Jeff Sauro und Joseph S. Dumas. „Comparison of three one-question, post-task usability questionnaires“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 4. Apr. 2009, S. 1599–1608. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518946.

[SL09] Jeff Sauro und James R. Lewis. „Correlations among prototypical usability metrics: evidence for the construct of usability“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 4. Apr. 2009, S. 1609–1618. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518947.

[TT06] Donna Tedesco und Thomas Tullis. „A Comparison of Methods for Eliciting Post-Task Subjective Ratings in Usability Testing“. In: *Usability Professionals Association (UPA)* 2006 (Jan. 2006), S. 1–9.

[Tuk77] John Wilder Tukey. *Exploratory data analysis*. Unter Mitarb. von Internet Archive. Reading, Mass. : Addison-Wesley Pub. Co., 1977. 714 S. ISBN: 978-0-201-07616-5. URL: http://archive.org/details/exploratorydataa00tuke_0 (besucht am 03. 09. 2021).

[ZD85] Fred Zijlstra und L. Doorn. „The Construction of a Scale to Measure Perceived Effort“. In: *Department of Philosophy and Social Sciences* (1985).