freenet GROUP

Nutzung abstrakter Datentypen zur Verbesserung der Codequalität

Thesis zur Erlangung des akademischen Grades

Bachelor of Science

Autor:	Noah Peeters
Adresse:	Wilmersdorfer Straße 1
	21502 Geesthacht
E-Mail:	noah@noahpeeters.de
Studiengang:	Angewandte Informatik (B.Sc.)
Zenturie:	A17a
Matrikelnummer:	8252
Gutachter:	Prof. Dr.-Ing. Johannes Brauer
Zweitgutachter:	Prof. Dr. Joachim Sauer
Betreuung:	B.Sc. Kay Butter
Unternehmen:	freenet.de GmbH

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

Tabellenverzeichnis

1 Einführung

Die Qualität von Software ist eine wichtige Eigenschaft, insbesondere wenn die Software in kritischen Bereichen eingesetzt wird.[1] So hat zum Beispiel ein Softwarefehler zum Absturz der Rakete Ariane 5 geführt.[2] Ereignisse wie dieses haben wiederum insgesamt zu einem erhöhten Bewusstsein für Softwarequalität geführt.[3] Die Qualität ist jedoch auch für alle wirtschaftlichen Unternehmen von Interesse, da diese für den Gewinn, der durch die Software erreicht wird, ausschlaggebend sein kann.[4]

Um eine hohe Qualität zu gewährleisten, werden verschiedene Methoden eingesetzt. Eine häufig verwendete Methode ist das Durchführen von Tests, um Fehler und Qualitätsmängel festzustellen.[5] In dieser Arbeit wird die Möglichkeit untersucht, die Qualität durch das Nutzen von Abstraktion zu verbessern.

Abstraktion ist in der Informatik und in Software Engineering ein mächtiges Konzept, das es ermöglicht Komplexität zu bewältigen. Dabei kann Abstraktion auf verschiedenen Ebenen angewendet werden.[6] Um Abstraktion zu erreichen, können zwei unterschiedliche Methoden eingesetzt werden. Zunächst gibt es die Abstraktion durch Parametrisierung. Dabei wird die Implementierung so umgesetzt, dass sie mit verschiedenen Eingabedaten verwendet und somit wiederbenutzt werden kann. Ein Beispiel hierfür ist eine Sortierfunktion, dessen Parameter eine Liste ist. Durch die Parametrisierung kann die Funktion nun in verschiedenen Kontexten verwendet werden, solange es das Ziel ist, eine beliebige Liste zu sortieren.[7] Die zweite Methode ist die Abstraktion durch Spezifikation. In diesem Fall wird ausschließlich das Ein- und Ausgabeverhalten definiert, welches von einer nicht weiter spezifizierten Implementierung umgesetzt wird. Wie dieses Verhalten umgesetzt ist, wird nach außen nicht definiert. Dies ermöglicht die Verwendung, ohne die konkrete Implementierung zu kennen.[8]

Mithilfe dieser Methoden können verschiedene Arten von Abstraktion erreicht werden. So können sowohl Verhalten als auch Daten und deren Kombination abstrahiert werden.[9] In dieser

[1] Jiantao Pan. „Software testing“. In: *Dependable Embedded Systems* 5 (1999), S. 2006.

[2] Jacques-Louis Lions u. a. „Flight 501 failure“. In: *Report by the Inquiry Board* 190 (1996).

[3] Manfred Broy, Florian Deißenböck und Markus Pizka. „A holistic approach to software quality at work“. In: *Proc. 3rd world congress for software quality (3WCSQ)*. 2005.

[4] Khaled El Emam. *The ROI from software quality*. CRC press, Juni 2005. DOI: 10.1201/9781420031201.

[5] Maneela Tuteja, Gaurav Dubey u. a. „A research study on importance of testing and quality assurance in software development life cycle (SDLC) models“. In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.3 (2012), S. 251–257.

[6] Gregor Kiczales. „Towards a new model of abstraction in software engineering“. In: *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*. IEEE. 1991, S. 127–128. DOI: 10.1109/IWOOOS.1991.183036.

[7] Barbara Liskov, John Guttag u. a. *Abstraction and specification in program development*. Bd. 180. MIT press Cambridge, 1986, S. 14.

[8] Ebd., S. 15.

[9] Ebd., S. 17.

Arbeit werden *abstrakte Datentypen* als eine Form der Abstraktion von Daten durch Spezifikation untersucht.

Abstrakte Datentypen wurden zunächst von Liskov und Zilles 1974 eingeführt.[10] Seitdem sind abstrakte Datentypen ein großes Forschungsthema. Mit dem Suchbegriff `"abstract data types"` werden auf Google Scholar etwa 130.000 Ergebnisse gefunden.[11] Außerdem wurde die Arbeit von Liskov und Zilles laut Google Scholar 1.133-mal zitiert.[12]

Diese Arbeit untersucht einen Zusammenhang zwischen der Nutzung abstrakter Datentypen und Codequalität. Dabei ist die Kernfrage der Arbeit, wie sich die Nutzung abstrakter Datentypen auf die Codequalität konkret auswirkt und insbesondere ob eine Qualitätssteigerung erreicht werden kann.

Das Vorgehen der Arbeit ist in drei Schritte untergliedert. Zunächst werden die theoretischen Grundlagen vorgestellt. Dabei werden abstrakte Datentypen und Codequalität als die beiden wesentlichen Kernaspekte des Themas definiert und beschrieben. Dafür wird auf bestehende Literatur zurückgegriffen und diese Arbeit in deren Kontext gesetzt. Anschließend werden mithilfe von Interviews Daten erhoben. Für diese Interviews wird zunächst die Methodik begründet und beschrieben. Die so erhobenen Daten werden dann strukturiert aufgearbeitet. Abschließend werden die erhobenen Daten und die theoretischen Grundlagen verwendet, um die Kernfrage dieser Arbeit zu beantworten.

[10] Barbara Liskov und Stephen Zilles. „Programming with Abstract Data Types“. en. In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Santa Monica, California, USA: Association for Computing Machinery, 1974, S. 50–59. ISBN: 9781450378840. DOI: 10.1145/800233.807045.

[11] Google Scholar. *Google Scholar search for „abstract data types“*. 1. Feb. 2021. URL: https://scholar.google.com/scholar?q=%22abstract+data+types%22 (besucht am 01. 02. 2021).

[12] Google Scholar. *Google Scholar search for „Programming with abstract data types“*. 1. Feb. 2021. URL: https://scholar.google.com/scholar?q=Programming+with+abstract+data+types (besucht am 01. 02. 2021).

2 Abstrakte Datentypen

Abstrakte Datentypen stellen neben der Codequalität den ersten Kernaspekt der Fragestellung und damit auch dieser Arbeit dar. Daher ist eine genaue Definition abstrakter Datentypen notwendig.

Die erste Verwendung des Begriffes *abstrakter Datentyp* findet sich in der Arbeit „Programming with Abstract Data Types“ von Barbara Liskov und Stephen Zills aus 1974, in der der Begriff definiert wurde. Der Anlass dafür war die Entwicklung der Programmiersprache CLU durch Liskov.[13]

Insbesondere in diesem Kapitel ist die Unterscheidung zwischen abstrakten Datentypen als mathematisches Modell und der Nutzung in der Programmierung wichtig. Die Unterscheidung ist in der gesamten Arbeit relevant, wobei der Fokus der Arbeit und insbesondere der späteren Abschnitte überwiegend auf der Anwendung liegt.

2.1 Abstrakte Datentypen als mathematisches Modell

In dieser Arbeit wird die Definition aus der bereits erwähnten Arbeit von Liskov und Zilles verwendet:[14]

> Ein abstrakter Datentyp ist ein Datentyp, der vollständig durch die verfügbaren Operationen charakterisiert ist.

Die Definition kann in drei Kernaspekte aufgeteilt werden. Zunächst muss geklärt werden, worum es sich bei einem Datentyp handelt. Der zweite Aspekt sind die verfügbaren Operationen. Der letzte Aspekt ist die Frage, was es bedeutet durch Operation charakterisiert zu sein. Diese drei Aspekte werden im Folgenden weiter definiert und ausgeführt, um anschließend in einem zweiten Schritt den Transfer zur Nutzung in der Programmierung zu ermöglichen.

2.1.1 Datentyp

Für den Begriff Datentyp gibt es keine allgemeingültige Definition, die in der Literatur einheitlich verwendet wird.[15] Historisch wurde der Begriff erst benötigt, als nutzerdefinierte Datentypen zusätzlich zu den vordefinierten Datentypen einer Programmiersprache (engl. build-in data types) verwendet wurden. Daher gibt es historisch bedingt verschiedene Definitionen.[16]

[13] Liskov und Zilles, „Programming with Abstract Data Types“.

[14] Ebd.

[15] Johannes Brauer. „Typen, Objekte, Klassen: Teil 1 – Grundlagen“. de. In: *Arbeitspapiere der Nordakademie* (Jan. 2009).

[16] D. L. Parnas, John E. Shore und David Weiss. „Abstract Types Defined as Classes of Variables“. In: *SIGPLAN Not.* 11.SI (März 1976), S. 149–154. ISSN: 0362-1340. DOI: 10.1145/942574.807133.

Die verschiedenen Ansätze der Definitionen können klassifiziert werden. So kann der Begriff über die möglichen Werte, die Repräsentation durch Teildatentypen, das Verhalten oder eine Kombination dieser definiert werden.[17] Dazu kommt, dass der Datentypbegriff häufig eng an die Programmiersprache gekoppelt und von dieser kaum trennbar ist.[18]

Daher wird an dieser Stelle zunächst auf eine genaue Definition für Datentypen verzichtet. Im Verlauf der Arbeit wird die Verwendung abstrakter Datentypen in einer konkreten Programmiersprache weiter ausgeführt. Dabei wird auf den Datentypbegriff erneut eingegangen und im Kontext dieser Programmiersprache betrachtet.

2.1.2 Operationen

Operationen *eines Datentyps* können zunächst als Funktionen betrachtet werden, die mit dem Datentyp in Verbindung stehen. Um dies zu konkretisieren, können die Operationen in vier verschiedene Kategorien eingeteilt werden.[19]

Zunächst gibt es erzeugenden Operationen. Diese bilden beliebige andere Werte auf einen Wert aus der Menge des Datentyps ab und sind damit in der Lage, Werte des Datentyps zu erstellen. Auf diese Werte können dann die anderen Operationsarten angewendet werden.

Seien

W die Menge aller beliebigen Werte,
A die Menge aller Werte des betrachteten Datentyps,
$\overline{W} = W \setminus A$ und
$X^* = \bigcup_{i \in \mathbb{N}_0} X^i$ die kleenesche Hülle der Menge.[20]

Dann gilt für die Definitionsmenge $D_{\text{erzeugend}}$ und die Zielmenge $Z_{\text{erzeugend}}$:

$$D_{\text{erzeugend}} \subseteq \left(\overline{W}\right)^* \qquad Z_{\text{erzeugend}} \subseteq A$$

Die zweite Kategorie bilden die produzierenden Operationen. Diese bilden einen existierenden Wert des Datentyps zusammen mit beliebigen anderen Werten auf einen neuen Wert des Datentyps ab. Formal betrachtet gilt für die Definitionsmenge und die Zielmenge:

$$D_{\text{produzierend}} \subseteq A \times W^* \qquad Z_{\text{produzierend}} \subseteq A$$

[17] Parnas, Shore und Weiss, „Abstract Types Defined as Classes of Variables“.
[18] Johannes Brauer. „Typen, Objekte, Klassen: Teil 2 – Sichtweisen auf Typen“. de. In: *Arbeitspapiere der Nordakademie* (Jan. 2010).
[19] Michael Kart. „Teaching Type Design Using Transition Diagrams and Sports Scoreboards“. en. In: *J. Comput. Sci. Coll.* 31.4 (Apr. 2016), S. 28–35. ISSN: 1937-4771.
[20] Martin Hofmann und Martin Lange. *Automatentheorie und Logik*. Springer-Verlag, 2011.

Die dritte Kategorie bilden die observierenden Operationen. Diese bilden einen existierenden Wert des Datentyps zusammen mit beliebigen anderen Werten auf einen beliebigen anderen Wert ab. Für die Definitionsmenge und die Zielmenge gilt damit:

$$D_{\text{observierend}} \subseteq A \times W^* \qquad\qquad Z_{\text{observierend}} \subseteq \overline{W}$$

Die letzte Kategorie bilden die mutierenden Operationen. Diese verändern einen existierenden Wert des Datentyps. Damit gilt, wie auch schon bei den produzierenden Operationen:

$$D_{\text{mutierend}} \subseteq A \times W^* \qquad\qquad Z_{\text{mutierend}} \subseteq A$$

Semantisch unterscheiden sich mutierende von den produzierenden Operationen, indem das Ergebnis als derselbe aber veränderte Wert betrachtet werden kann, während bei produzierenden Operationen das Ergebnis als neuer Wert betrachtet wird. So wäre bezogen auf ganze Zahlen die Fakultät eine produzierende Operation, während inkrementieren eine mutierende Operation ist. Mathematisch ist diese Differenzierung zunächst nicht notwendig. Bei der Nutzung in Programmiersprachen, welche einen veränderbaren Zustand unterstützen, ist diese Differenzierung jedoch sinnvoll.[21]

Neben der Forderung der Syntax der Operationen, also insbesondere der Spezifizierung der Anzahl und Art der Parameter, wird bei abstrakten Datentypen in der Regel auch die Semantik definiert. Als typische Beispiele werden hierbei eine Queue (auch FIFO = First-In-First-Out oder Warteschlange) und ein Stack (auch LIFO = Last-In-First-Out oder Stapel) betrachtet. Beide Datenstrukturen haben eine Operation zum Hinzufügen und eine Operation zum Entfernen eines Elements. Der große Unterschied liegt jedoch in der Semantik der Operationen. Erst dadurch wird ersichtlich, dass die Reihenfolge des Entfernens umgekehrt ist.[22] Um die Semantik zu definieren, gibt es verschiedene Ansätze. Eine Möglichkeit stellen algebraische Axiome dar. Dabei werden algebraische Formeln verwendet, um die semantische Beziehung zwischen den Operationen zu formalisieren. Ein solches Axiom kann folgendermaßen aussehen:[23]

$$\forall a, b : \text{operation1}(\text{operation2}(a, b)) = a$$

Neben der Syntax und der Semantik werden teilweise weitere Eigenschaften von Operationen definiert. Diese können unter anderem die Komplexität und insbesondere das Laufzeitverhalten in Bezug auf Rechenaufwand und Speicherbedarf sein. Diese Eigenschaften sind jedoch nicht

[21] Kart, „Teaching Type Design Using Transition Diagrams and Sports Scoreboards“.

[22] John Guttag. „Abstract Data Types and the Development of Data Structures“. In: *Commun. ACM* 20.6 (Juni 1977), S. 396–404. ISSN: 0001-0782. DOI: 10.1145/359605.359618.

[23] John V. Guttag und James J. Horning. „The algebraic specification of abstract data types“. In: *Acta informatica* 10.1 (1978), S. 27–52.

zwingend notwendig für die Definition abstrakter Datentypen und werden in der weiteren Arbeit nicht betrachtet.[24]

2.1.3 Charakterisieren

Mit *charakterisieren* ist gemeint, dass ein abstrakter Datentyp vollständig definiert werden kann, indem die Operationen definiert werden. Das bedeutet, es ist ausreichend, diese Operationen mit den zuvor genannten Eigenschaften zu definieren. Anschließend ist es möglich, den abstrakten Datentyp zu verwenden. Sofern ausreichend viele Operationen definiert sind, ist die Nutzung ohne Einschränkungen möglich.[25] Für die theoretische Verwendung ist eine Implementierung beziehungsweise Wissen über die konkrete Implementierung dieser Operationen nicht notwendig. Für die praktische Nutzung ist es natürlich dennoch unabdingbar, dass mindestens eine konkrete Implementierung verfügbar ist. Aus dieser Definition abstrakter Datentypen folgt, dass diese Datentypen ohne Wissen über die Implementierung verwendet werden können. Das bedeutet, dass anschließend die Implementierung beliebig verändert werden kann, solange die Semantik und gegebenenfalls weitere definierte Eigenschaften wie das Laufzeitverhalten weiter erhalten bleiben.

Typischerweise werden als Beispiele für abstrakte Datentypen Sammeldatenstrukturen wie Stacks oder Listen verwendet. Diese lassen sich gut über die Operationen definieren. Darüber hinaus lassen sich weitere Eigenschaften wie die Laufzeitkomplexität von `O(1)` definieren, die für diese Datenstruktur ausreichend ist.[26]

Im weitesten Sinne können alle vordefinierten Datentypen einer Programmiersprache als abstrakte Datentypen betrachtet werden. Bereits Liskov und Zills gaben an, „Abstract types are intended to be very much like the built-in types provided by a programming language.“[27]

Betrachtet man beispielhaft ganze Zahlen, die in vielen Programmiersprachen als vordefinierter Datentyp unter Namen wie `int`, `integer` oder `number` verfügbar sind, erfüllen diese in der Regel die Eigenschaften abstrakter Datentypen. Es werden alle charakterisierenden Operationen definiert. Dazu gehören unter anderem erzeugende Operationen wie das direkte Initialisieren mit einem Wert, produzierende Operationen wie die Addition und die Multiplikation, observierende Operationen wie die Prüfung auf Gleichheit und mutierende Operationen wie das Inkrementieren des Wertes. Die konkrete Implementierung ist bei der Nutzung irrelevant und kann ohne Auswirkungen auf die Programme verändert werden.

Vordefinierte Datentypen werden in der restlichen Arbeit nicht weiter als abstrakte Datentypen im engeren Sinne in Bezug auf das Thema betrachtet. Dies würde über den Rahmen dieser

[24] Manoochchr Azmoodeh. *Abstract data types and algorithms*. Palgrave Macmillan UK, 1990. DOI: 10.1007/978-1-349-21151-7.

[25] Liskov, Guttag u. a., *Abstraction and specification in program development*, S. 76.

[26] Guttag, „Abstract Data Types and the Development of Data Structures“.

[27] Liskov und Zilles, „Programming with Abstract Data Types“.

Arbeit hinausgehen. Daher sind im Folgenden mit abstrakten Datentypen nur die Datenstrukturen gemeint, die von Nutzern der Programmiersprache definiert wurden.

2.1.4 Beispiel Stack

Nun kann die bereits angesprochene Datenstruktur Stack als abstrakter Datentyp definiert werden. In Listing 1 ist eine mögliche Definition zu finden. Zu den charakterisierenden Operationen gehört eine erzeugende, zwei observierende und zwei mutierende Operation. Die Semantik wird mithilfe von algebraischen Gleichungen definiert. Zu beachten ist, dass die Kommentare (eingeleitet mit `//`) nicht Teil der Definition sind und lediglich einem besseren Verständnis dienen. Weitere Eigenschaften wie die Laufzeitkomplexität sind nicht definiert.

```
// Definition des abstrakten Datentyps
Stack<Element>
    // Definition der charakterisierenden Operationen
    // erzeugenden Operation, um einen leeren Stack zu erstellen
    emptyStack: () → Stack
    // observierende Operation, um zu ermitteln, ob der Stack leer ist
    isEmpty: Stack → boolean
    // observierende Operation, um das oberste Element zu ermitteln
    top: Stack → Element
    // mutierende Operation, um das oberste Element zu entfernen
    pop: Stack → Stack
    // mutierende Operation, um ein Element oben auf den Stack zu legen
    push: Stack × Element → Stack

    // Definition der Semantik
    ∀s ∈ Stack, e ∈ Element:
        isEmpty(emptystack) = true
        isEmpty(push(s, e)) = false
        top(push(s, e)) = e
        pop(push(s, e)) = s
```

Listing 1: Definition des abstrakten Datentyps `Stack` *(Quelle: Eigener Quelltext nach Guttag, „Abstract Data Types and the Development of Data Structures“)*

2.2 Abstrakte Datentypen in der Programmierung verwenden

Betrachtet man nun die Nutzung abstrakter Datentypen in Programmiersprachen, muss der Transfer von dem mathematischen Modell zu den Möglichkeiten in der Programmierung, die die Programmiersprachen bereitstellen, erfolgen. Da unterschiedliche Programmiersprachen mit

verschiedenen Typsystemen andere Möglichkeiten haben, wird nun eine Sprachauswahl für die weitere Arbeit festgelegt. Dies ermöglicht die parallele theoretische und praktische Betrachtung.

Für die Wahl der Programmiersprache müssen zwei Bedingungen erfüllt sein. Zunächst muss die Sprache die Option bieten, entsprechende Datentypen zu definieren. Das bedeutet, das Typsystem muss es ermöglichen, eigene Datentypen mit Operationen zu definieren und die Implementierung zu verbergen. Dies ist jedoch bei den meisten Sprachen zu einem ausreichenden Grad gegeben.

Die zweite Bedingung ist rein praktisch für die empirische Untersuchung im Verlauf der Arbeit notwendig. Für die empirische Untersuchung werden Probanden benötigt, die die gewählte Programmiersprache bereits kennen. Durch die Wahl einer bekannten Programmiersprache wird die Akquise der Probanden vereinfacht. Daher wird eine verbreitete Programmiersprache gewählt.

Insbesondere aufgrund der zweiten Bedingung wird in der weiteren Arbeit die Programmiersprache *Java* verwendet. Java ist eine weit verbreitetet Programmiersprache, sodass ausreichend Experten zur Verfügung stehen.[28] Im Folgenden wird gezeigt, wie sich das mathematische Konzept der abstrakten Datentypen auf die Programmiersprache Java beziehungsweise dessen Typsystem übertragen lässt.

Wie nun ausführlich erläutert wurde, handelt es sich bei abstrakten Datentypen nicht um eine Implementierungsstrategie, sondern zunächst um ein mathematisches Modell. Um abstrakte Datentypen in Java zu verwenden, müssen alle Eigenschaften, die im Abschnitt 2.1 analysiert wurden, in die Programmierung transferiert werden. Damit kann anschließend ein Datentyp nach den theoretischen Regeln definiert und dann in Java Quelltext umgesetzt werden. Für den vollständigen Transfer muss der Datentyp an sich beziehungsweise die möglichen Werte des Datentyps, die charakteristischen Operationen sowie die Eigenschaften dieser Operationen in der Programmiersprache Java abgebildet werden.

2.2.1 Abbilden des Datentyps

Java kennt verschiedene Konstrukte, um eigene Datentypen zu definieren. Konkret sind dies *Interfaces*, *Klassen* und *Enumerationen* als spezielle Form von Klassen.

Zunächst kann für einen abstrakten Datentyp eine Klasse definiert werden. In der Klasse können dann alle charakterisierenden Operationen definiert werden. In diesem Fall findet im Quelltext keine strikte Trennung von Schnittstelle und Implementierung statt. Durch das Konzept der Kapselung ist es jedoch möglich, die Implementierung nach außen zu verbergen. In Java wird dies durch den Einsatz von Zugriffsmodifikatoren (engl. access modifiers) – insbesondere `private` – erreicht. Dadurch kann festgelegt werden, welche Teile der Implementierung nach außen als

[28] Stack Overflow Ltd. *Stack Overflow Annual Developer Survey*. Feb. 2020. URL: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages (besucht am 20. 12. 2020).

Schnittstelle sichtbar sind und welche nicht. Hiermit können insbesondere Instanzvariablen verborgen werden. In Java ist es üblich, Instanzvariablen als `private` zu deklarieren und nur mithilfe von Get- und gegebenenfalls Set-Methoden von außen erreichbar zu machen.[29] Diese werden hier weggelassen, um im Sinne abstrakter Datentypen nur die charakterisierenden Operationen nach außen verfügbar zu machen. Alle weiteren Implementierungsdetails können in der Klasse verborgen bleiben.

Die zweite Möglichkeit stellen Enumerationen (*enum*) dar. Enumerationen sind in Java eine spezielle Form von Klassen, die mithilfe vordefinierter Konstanten nur einen fest umrissenen Wertebereich abbilden.[30] In Bezug auf die Nutzung von Enumerationen als Datentyp für die Umsetzung abstrakter Datentypen verhalten sich Enumerationen genauso wie Klassen.

Die letzte Möglichkeit stellt das Definieren eines Interface dar. Interfaces können in Java benutzt werden, um eine Schnittstelle zu definieren, die anschließend von konkreten Klassen oder Enumerationen implementiert werden muss. Dies erlaubt die strikte Trennung der Schnittstelle und der Implementierung.[31] Für Nutzer des Interface ist dabei die konkrete Implementierung irrelevant. Lediglich die angebotenen Operationen des Interface müssen bekannt sein, um dieses zu nutzen. Durch die Möglichkeit, mehrere, unabhängige Implementierungen bereitzustellen, kann die Implementierung zur Laufzeit ausgetauscht werden.

Darüber hinaus gibt es seit Java 8 die Möglichkeit, abstrakte Klassen und Interfaces mit einer Standardimplementierung zu definieren.[32] Dies stellt eine Kombination der ersten und der letzten Möglichkeit dar. Auf der einen Seite wird – wie bei einfachen Interfaces – eine konkrete Implementierung in Form einer Klasse oder einer Enumeration benötigt. Auf der anderen Seite beinhaltet diese bereits Teile der konkreten Implementierung, sodass keine strikte Trennung von Interface und Implementierung gegeben ist.

2.2.2 Abbilden der Operationen

Der wesentliche Teil der Definition abstrakter Datentypen sind die Operationen. Java bietet hierfür die drei Konstrukte *Instanzmethoden*, *statische Methoden* und *Konstruktoren*.

Instanzmethoden haben immer den impliziten Parameter `this`, welcher die aktuelle Instanz des abstrakten Datentyps referenziert. Für die Implementierung von produzierenden, observierenden und mutierenden Operationen wird jeweils eine Instanz als Argument benötigt, weshalb Instanzmethoden für deren Implementierung geeignet sind. Bei mutierenden Opera-

[29] Ken Arnold, James Gosling und David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[30] Ebd.

[31] Paolo A.G. Sivilotti und Matthew Lang. „Interfaces First (and Foremost) with Java“. en. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE '10. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 2010, S. 515–519. ISBN: 9781450300063. DOI: 10.1145/1734263.1734436.

[32] Oracle. *What's New in JDK 8*. 19. Apr. 2018. URL: https://www.oracle.com/java/technologies/javase/8-whats-new.html (besucht am 31. 01. 2021).

tionen muss dabei der veränderte Wert nicht als Rückgabewert zurückgegeben werden, wie es bei der theoretischen Betrachtung der Fall war. In Java kann stattdessen die Instanz selbst verändert werden.

Für erzeugende Operationen können statische Methoden verwendet werden, da bei diesen keine Instanz als Parameter benötigt wird. Darüber hinaus können statische Methoden theoretisch auch für die restlichen Operationsarten verwendet werden, indem explizit eine Instanz des abstrakten Datentyps als Parameter übergeben wird.

Konstruktoren können ebenfalls für erzeugende Operationen verwendet werden, da diese keine Instanz als Parameter benötigen, sondern eine Instanz als Ergebnis zurückgeben. Außerdem lassen sich auch produzierende Operationen mit Konstruktoren abbilden, indem auch hier eine Instanz des abstrakten Datentyps als expliziter Parameter übergeben wird.

Wird der abstrakte Datentyp mithilfe eines Interface abgebildet, ist es in Java nicht möglich, erzeugende Operationen in dem Interface zu definieren. In einem Interface kann in Java kein Konstruktor definiert werden.[33] Statische Methoden können seit Java 8 auch in Interfaces definiert werden.[34] Diese verhalten sich jedoch anders als statische Methoden, wie sie für einen abstrakten Datentyp benötigt werden. Instanzmethoden in einem Interface definieren zusammen eine minimale Schnittstelle, welche von einer Klasse implementiert werden muss. Statische Methoden hingegen müssen im Interface implementiert sein und werden direkt auf dem Interface aufgerufen. Ein Aufruf könnte zum Beispiel folgendermaßen aussehen: `MeinInterface.meineStatischeMethode()`. Die so definierten Methoden gehören nicht mit zu der minimalen Schnittstelle, die implementiert werden muss und eignen sich damit auch nicht für das Abbilden von erzeugenden Operationen. Dies ist eine Einschränkung der Programmiersprache Java. Typsysteme anderer Programmiersprachen wie zum Beispiel in *Swift* können das gewünschte Verhalten abbilden.[35]

2.2.3 Abbildung der Eigenschaften von Operationen

Die Syntax der Operationen und damit im Speziellen der Name sowie die Anzahl und die Typen der Parameter können wie zuvor beschrieben in Java Quelltext abgebildet und vom Java Compiler überprüft werden. Die weiteren Eigenschaften lassen sich jedoch nur eingeschränkt definieren und werden nicht vom Java Compiler überprüft. Stattdessen können diese Eigenschaften nur in Form von Kommentaren und Javadoc, der Benennung von Methoden oder externer Dokumentation abgebildet werden.

[33] Sivilotti und Lang, „Interfaces First (and Foremost) with Java“.

[34] Oracle, *What's New in JDK 8*.

[35] Dimitri Racordon und Didier Buchs. „Featherweight swift: A core calculus for swift's type system“. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, S. 140–154.

2.2.4 Beispiel Stack

Basierend auf der Definition des abstrakten Datentyps Stack aus Abschnitt 2.1.4 und den dargestellten Möglichkeiten, abstrakte Datentypen in Java abzubilden, kann dieser Transfer nun exemplarisch durchgeführt werden. Dies wird im Folgenden einmal mithilfe einer konkreten Klasse und einmal mithilfe eines Interface gezeigt. Eine Implementierung mit einer Enumeration als Datentyp ist nicht möglich, da es sich nicht um einen kleinen, diskreten Wertebereich handelt.

Eine mögliche Klassendefinition ist in Listing 2 dargestellt. Zunächst wird eine Instanzvariable `elements` definiert. Da die konkrete Implementierung und damit auch die Abbildung des Zustandes im Speicher nicht Teil des abstrakten Datentyps ist, wird die Instanzvariable `private` definiert. Ferner werden auch keine Get- und Set-Methoden definiert, da diese nicht zu den charakterisierenden Operationen eines Stacks gehören und damit nicht gebraucht werden. Die erzeugende Operationen `emptyStack` kann als Konstruktor ohne Parameter abgebildet werden. Alle weiteren Operationen werden als Instanzmethoden umgesetzt. Zu beachten ist dabei, dass die mutierende Operation `pop` nun direkt die Instanz modifizieren kann und nicht mehr wie im mathematischen Modell eine veränderte Instanz zurückgeben muss. Daher kann `pop` nun den entfernten Wert zurückgeben. Bei allen Methoden wurde die Implementierung mit Auslassungszeichen (`[...]`) weggelassen, da diese hier nicht relevant ist.

```
public class Stack<E> {
    private List<E> elements;

    public Stack() { [...] }
    public boolean isEmpty() { [...] }
    public E top() { [...] }
    public E pop() { [...] }
    public void push(E element) { [...] }
}
```

Listing 2: `Stack` als Klasse *(Quelle: Eigener Quelltext)*

Eine mögliche Interfacedefinition ist in Listing 3 dargestellt. Im Gegensatz zu der Klassendefinition werden beim Interface keine Instanzvariablen definiert. Des Weiteren ist es nicht möglich, die erzeugende Operationen `emptyStack` abzubilden. Die restlichen Operationen werden wie bei der Klassendefinition umgesetzt. Das Interface ist dabei lediglich die Schnittstellendefinition, weshalb keine Implementierung der Methoden definiert wird. Stattdessen muss eine Klasse definiert werden, welche das Interface implementiert.

Als nächster Schritt kann nun die Semantik der einzelnen Operationen beschrieben werden, indem zum Beispiel Kommentare an die Methoden geschrieben werden. Die einfachste Methode

```java
public interface IStack<E> {
    public boolean isEmpty();
    public E top();
    public E pop();
    public void push(E element);
}
```

Listing 3: `Stack` als Interface *(Quelle: Eigener Quelltext)*

ist die Beschreibung der Semantik mithilfe natürlicher Sprache, welche aber aufgrund von Fehlinterpretation zu Missverständnissen führen kann.[36]

[36] F. Chantree u. a. „Identifying Nocuous Ambiguities in Natural Language Requirements“. In: *14th IEEE International Requirements Engineering Conference (RE'06)*. 2006, S. 59–68. DOI: 10.1109/RE.2006.31.

3 Codequalität

Neben dem generellen Aspekt der Verwendung abstrakter Datentypen stellt sich die spezielle Frage nach einer Verbesserung der Codequalität durch deren Nutzung. Das Messen von Codequalität ist eine nicht triviale Problemstellung. Daher wurden einige Modelle entwickelt, mit dem Ziel, Softwaresysteme hinsichtlich der Qualität im Allgemeinen und auch der Codequalität im Speziellen zu bewerten.[37]

3.1 Modell zur Messbarkeit von Codequalität

In dieser Arbeit wird der ISO-Standard 25010 mit dem Titel „System and software quality models“ verwendet werden. ISO 25010 definiert dabei Software[produkt]qualität[38] als die Qualität des gesamten Softwareproduktes,[39] also der Summe der Programme, Prozeduren, Daten und Dokumente des finalen Produktes sowie aller Zwischenprodukte.[40]

Der Standard definiert ein Vorgehensmodell, welches in dieser Arbeit verwendet wird. Zunächst müssen die Ziele der Qualitätsuntersuchung festgelegt werden. Dann muss eine Teilmenge der von dem Modell angebotenen Charakteristiken ausgewählt werden, sodass mit diesen die zuvor definierten Ziele erreicht werden können. Dabei ist zu beachten, dass es aufgrund des Umfangs nicht praktikabel ist, alle Teilcharakteristiken zu untersuchen. Anschließend kann das System anhand der gewählten Charakteristiken beurteilt werden.[41]

ISO 25010 stellt zwei verschiedene Modelle mit unterschiedlichen Anwendungsbereichen zur Verfügung. Zunächst gibt es das sogenannte *quality in use model*, welches dazu dient, die Qualität von Software während der Benutzung zu bewerten. Das zweite Modell ist das *Produktqualitätsmodell* (engl. *product quality model*), welches die Qualität des Produktes unabhängig von dem Benutzungskontextes betrachtet. Stattdessen wird die Qualität des Produktes isoliert betrachtet.[42]

Diese Arbeit hat das Ziel, die Codequalität zu untersuchen und verwendet daher das Produktqualitätsmodell. Diese Zielvorgabe bestimmt die Auswahl der Charakteristiken und lässt insbesondere jene aus, die sich auf andere Bereiche beziehen.

[37] John Estdale und Elli Georgiadou. „Applying the ISO/IEC 25010 Quality Models to Software Product“. In: *Systems, Software and Services Process Improvement*. Hrsg. von Xabier Larrucea u. a. Cham: Springer International Publishing, 2018, S. 492–503. ISBN: 978-3-319-97925-0.

[38] ISO Central Secretary. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. en. Standard ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization, 2011, S. 19.

[39] Ebd., S. 17.

[40] Ebd., S. 19.

[41] Ebd., S. 5.

[42] Ebd., S. 5.

Das in Abbildung 1 dargestellte Produktqualitätsmodell definiert insgesamt acht Charakteristiken, welche jeweils in bis zu sechs Teilcharakteristiken aufgeteilt sind. Von den insgesamt 31 Teilcharakteristiken werden im Folgenden die für diese Arbeit relevanten ausgewählt. In einem ersten Schritt werden die unwesentlichen Charakteristiken ausgeschlossen. Anschließend werden die relevanten Teilcharakteristiken ermittelt.

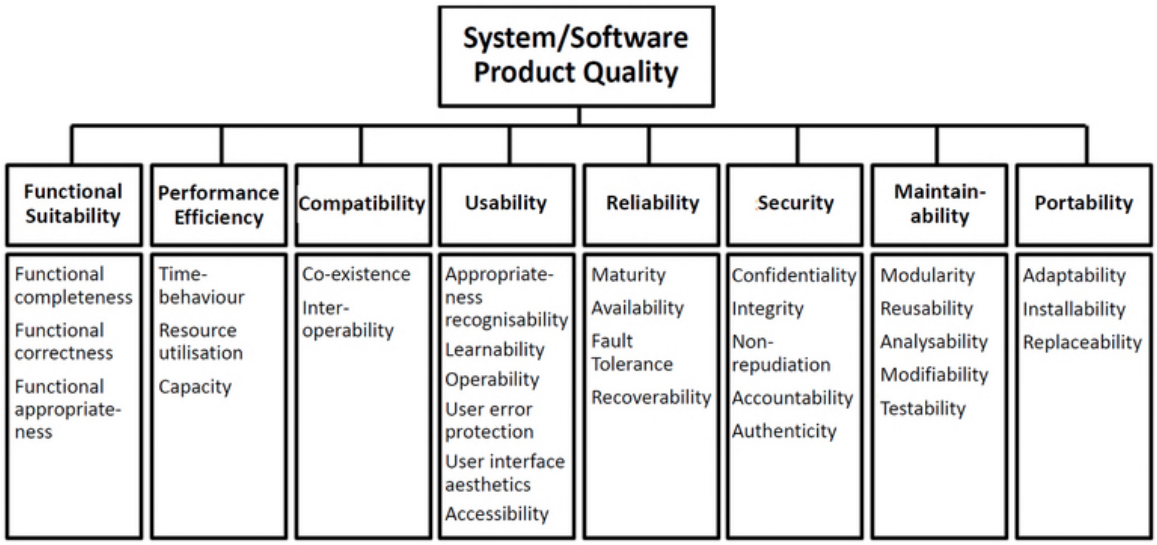

Abbildung 1: ISO 25010 Produktqualitätsmodell *(Quelle: ISO Central Secretary,* ISO/IEC 25010:2011*)*

Die Charakteristiken *Kompatibilität*, *Benutzbarkeit*, *Zuverlässigkeit* und *Übertragbarkeit* beziehen sich vor allem auf den Einsatz der Software und werden daher aufgrund des Fokus auf Codequalität nicht weiter betrachtet. Des Weiteren werden die beiden Charakteristiken *Effizienz* und *Sicherheit* nicht untersucht. Diese können auf die Codequalität bezogen werden, werden aber aufgrund des Umfangs ebenfalls nicht weiter betrachtet.

Die beiden Charakteristiken *Funktionalität* und *Wartbarkeit* lassen sich unmittelbar auf den Quelltext beziehen und werden daher als die relevantesten der acht Charakteristiken analysiert.

Die *Funktionalität* ist in drei Teilcharakteristiken aufgeteilt, von denen in dieser Arbeit zwei untersucht werden. *Funktionale Vollständigkeit* betrachtet den Abdeckungsgrad der spezifizierten Anforderungen und Nutzeranforderungen. *Funktionale Korrektheit* bewertet, ob die angebotene Funktionalität die richtigen Ergebnisse mit der geforderten Präzision liefert. Die *Angemessenheit der Funktionalität* wird nicht analysiert, da diese nur im Kontext der Nutzung des Softwaresystems betrachtet werden kann.[43]

Die *Wartbarkeit* beschreibt die Modifizierbarkeit von Teilen des Systems durch die bestimmten Instandhalter.[44] Da hier die Qualität des Quelltextes untersucht wird, reduziert sich diese

[43]ISO Central Secretary, *ISO/IEC 25010:2011*, S. 11.
[44]Ebd., S. 14.

Gruppe auf die Softwareentwickler, welche den Quelltext anpassen. Die Wartbarkeit ist in fünf Teilcharakteristiken aufgeteilt, von denen in dieser Arbeit drei untersucht werden. *Modularität* bewertet, wie Änderungen isoliert an einer Komponente durchgeführt werden können, sodass Änderungen an anderen Komponenten minimiert werden können. *Analysierbarkeit* betrachtet die Lesbarkeit des Quelltextes. Dabei bezieht sich dies auf verschiedene Aspekte wie das Finden von Fehlern oder das Identifizieren von Teilen, die für eine Änderung modifiziert werden müssen. *Modifizierbarkeit* betrachtet, wie effektiv und effizient diese Änderungen durchgeführt werden können, ohne dass die bestehende Qualität beeinträchtigt wird.[45] Nicht untersucht werden die beiden Teilcharakteristiken *Testbarkeit* und *Wiederverwendbarkeit*. Beide würden den Umfang der Analyse unverhältnismäßig vergrößern.

Die statische Codeanalyse stellt eine relativ einfache Methode zum Testen der Codequalität von Software dar. Sie sollte in der Softwareentwicklung automatisiert angewendet werden und ein Bestandteil des Entwicklungsprozesses sein. Sie ermöglicht die frühzeitige Erkennung von Codequalitätsproblemen.[46] In dieser Arbeit wird für die Bewertung von Codequalität ebenfalls eine statische Codeanalyse durchgeführt.

3.2 Relevanz von Codequalität

Codequalität ist zunächst ein Teilbereich der Softwarequalität und muss daher auch im Kontext der Gesamtqualität betrachtet werden. Wird lediglich die Codequalität betrachtet, kann damit alleine noch keine Aussage über die Qualität der Software getroffen werden.[47] Dennoch stellt die Codequalität einen sehr relevanten Teilaspekt dar, der – wie in Abschnitt 3.1 gezeigt – insbesondere im Kontext der Wartbarkeit bereits eine hohe Aussagekraft hat. Können Qualitätsprobleme am Quelltext rechtzeitig erkannt oder verhindert werden, bietet sich die Möglichkeit auf diese bereits zu einem frühen Zeitpunkt des Projektes zu reagieren.[48] Insbesondere machen Wartungskosten einen hohen Anteil der Gesamtkosten einer Software während ihres Lebenszyklus aus, sodass die frühe Erkennung oder Verhinderung dieser Probleme beträchtliche wirtschaftliche Vorteile mit sich bringen kann.[49]

[45] Ebd., S. 14.

[46] R. Plösch u. a. „A Method for Continuous Code Quality Management Using Static Analysis“. In: *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE. 2010, S. 370–375. DOI: 10.1109/QUATIC.2010.68.

[47] Estdale und Georgiadou, „Applying the ISO/IEC 25010 Quality Models to Software Product“, S. 1.

[48] Plösch u. a., „A Method for Continuous Code Quality Management Using Static Analysis“.

[49] El Emam, *The ROI from software quality*.

4 Beschreibung der Methodik

Um die Fragestellung der Arbeit beantworten zu können, werden Experteninterviews durchgeführt. Diese gehören zu den qualitativen Forschungsmethoden, sodass exploratorisch gearbeitet werden kann. Bei einer quantitativen Methode müsste standardisiert vorgegangen werden, indem zunächst Hypothesen mit entsprechendem Vorwissen aufgestellt werden.[50]

Die Interviews haben das Ziel, Daten zu erheben, die anschließend ausgewertet werden können. Im Folgenden wird zunächst das Basisprojekt für die Interviews beschrieben. Anschließend wird das Interviewdesign detailliert beschrieben, um die Nachvollziehbarkeit und Reproduzierbarkeit der Ergebnisse zu gewährleisten.[51]

4.1 Beschreibung des Basisprojektes

Für die Interviews dient als Basis ein Projekt, welches jedem Probanden bereitgestellt wird und auf das sich verschiedene Aufgaben beziehen. Dabei wird das Projekt mit zwei unterschiedlichen Ansätzen implementiert. Die eine Implementierung, welche im Folgenden als *ADT* (**a**bstrakte **D**aten**t**ypen) bezeichnet wird, verwendet nach der in Abschnitt 2.2 beschriebene Methode abstrakte Datentypen. Die zweite Variante, welche im Folgenden als *Classic* bezeichnet wird, nutzt diese nicht. Stattdessen sollen die Datenstrukturen nicht wie bei abstrakten Datentypen über die Operationen definiert sein, sondern über die Implementierung an sich. Dafür wird die interne Implementierung nicht verborgen, sondern ist nach außen sichtbar.

Trotz der unterschiedlichen Datenstrukturen verhalten sich die Projekte inhaltlich identisch und sind – soweit möglich – gleich implementiert. Dies vereinfacht die anschließende Analyse der Ergebnisse und erlaubt es, die Ursachen für Unterschiede besser ermitteln zu können. Beide Projekte werden aus den in Abschnitt 2.2 erläuterten Gründen in Java implementiert.

Das Projekt soll ein in sich abgeschlossenes und ausführbares Programm sein, damit darüber hinaus kein Kontextwissen notwendig ist und die Ergebnisse möglichst isoliert betrachtet werden können. Für die Wahl des Projektes sind insbesondere drei Kriterien relevant.

Zunächst darf der Umfang des Projektes nicht zu groß sein. Es soll möglich sein, das Interview in einem zeitlichen Rahmen von einer Stunde durchführen zu können. Aufgrund dieser Vorgabe haben die Probanden die Möglichkeit, das Projekt vollständig zu überblicken. Zudem wird die anschließende Auswertung der Ergebnisse vereinfacht, da es weniger Ursachen für Unterschiede in den Ergebnissen geben kann.

Zweitens soll die Datenstruktur im Vergleich zum Umfang des Projektes komplex sein, um das Ziel zu erreichen, zwei verschiedene Implementierungen der Datenstrukturen untersuchen zu

[50] Dominic Lindner. „Forschungsdesigns der Wirtschaftsinformatik“. In: *Forschungsdesigns der Wirtschaftsinformatik*. Springer, 2020, S. 19–44.
[51] Ebd.

können. Dies ist nur dann möglich, wenn das Projekt ausreichende Möglichkeiten bietet, sich in den beiden Implementierungen zu unterscheiden.

Zuletzt muss das Projekt insofern erweiterbar sein, dass sinnvolle Änderungen durchgeführt werden können. Dies ist notwendig, da sich einige der in Abschnitt 3.1 festgelegten Kriterien auf das Modifizieren von Software und Quelltext beziehen.

Bei der Simulation *Conways Spiel des Lebens* (engl. Conway's Game of Life) handelt es sich um einen zweidimensionalen zellulären Automaten mit einem unbegrenzt großen Gitter aus Zellen. Jede Zelle hat einen der beiden Zustände *lebendig* und *tot*. Die Regeln der Simulation sind sehr einfach und werden bei jeder Iteration parallel für alle Zellen ausgeführt. Jede lebendige Zelle mit weniger als zwei oder mehr als drei lebendigen Nachbarn stirbt. Jede tote Zelle mit genau drei lebendigen Nachbarn wird lebendig. Alle anderen Zellen verändern sich nicht. Nachbarn sind dabei alle Zellen, die eine Zelle vertikal, horizontal oder diagonal direkt berührt. Damit hat jede Zelle in dem unbegrenzt großen Gitter genau acht Nachbarn. In Abbildung 2 sind alle Nachbarn der Zelle x grün markiert.[52]

Die Simulation erfüllt alle Kriterien. Zunächst ist der Umfang des Projektes gering. Dies liegt insbesondere daran, dass es nur zwei Regeln gibt, sodass nur sehr wenig Logik benötigt wird. Die Datenstruktur ist mit einem unbegrenzt großen Feld, auf dem die Regeln parallel angewendet werden, ausreichend komplex und die Simulation bietet ausreichende Erweiterungsmöglichkeiten. Die genauen geforderten Änderungen werden im Abschnitt 4.2.2 beschrieben.

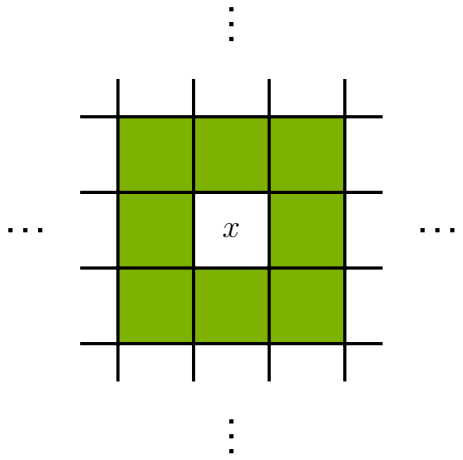

Abbildung 2: Nachbarn einer Zelle in Conways Spiel des Lebens *(Quelle: Eigene Darstellung)*

4.1.1 Implementierungsbeschreibung

Im Folgenden wird die Implementierung der beiden Versionen des Basisprojektes beschrieben und begründet. Dabei liegt der Fokus auf den Unterschieden zwischen den beiden Implementierungen.

Zunächst beinhalten beide Implementierungen eine Klasse `App` als Einstiegspunkt. Außerdem enthalten beide Implementierungen eine Klasse `BoardController`, welche die Regeln der Simulation enthält und im Weiteren als Controller bezeichnet wird. Der Controller hat drei Methoden. Einen Konstruktor, welcher ein Zellengitter mit einer deterministischen Startkonfiguration der

[52] Martin Gardner. „The fantastic combinations of John Conway's new solitaire game "life"“. In: *Scientific American* 223 (1970), S. 120–123.

lebendigen Zellen erstellt. Die zweite Methode führt einen Schritt der Simulation nach den beschriebenen Regeln durch. Die letzte Methode gibt den kleinen quadratischen Bereich zwischen den Zellen an den Koordinaten $(0, 0)$ und $(9, 9)$ des aktuellen Zellengitterzustandes aus.

Alle drei Methoden sind weitestgehend identisch implementiert und unterscheiden sich lediglich, wenn ein Unterschied in den Schnittstellen der Datenstrukturen dies erfordert. Auf diese Weise kann das Ziel erreicht werden, die beiden Implementierungen möglichst ähnlich zu gestalten.

Da wie in Abschnitt 3.1 erläutert, die Testbarkeit nicht untersucht wird, beinhaltet das Basisprojekt keine Tests. Diese würden den Quelltext vergrößern, sodass die Probanden mehr verstehen und gegebenenfalls verändern müssten, ohne dass dies einen Mehrwert für die anschließende Auswertung bietet.

4.1.2 Implementierungsbeschreibung – Classic

In der Classic Implementierung ist die Datenstruktur mit zwei Klassen abgebildet. Es existiert eine Klasse für die Zellposition (`CellPosition`) und eine Klasse für das Zellengitter (`Board`).

Die Zellposition speichert eine x- und eine y-Koordinate, um eine Position im Zellengitter eindeutig identifizieren zu können. Die Schnittstelle der Klasse hat fünf Methoden. Mithilfe eines Konstruktors, der die beiden Koordinaten als Parameter benötigt, kann eine neue Instanz erstellt werden. Für die beiden Koordinaten existiert jeweils eine Get-Methode. Darüber hinaus sind die beiden Java-Standardmethoden `equals` und `hashCode` jeweils durch die beiden Koordinaten definiert.

Das Board verwaltet den Zustand des Zellengitters. Dazu ist eine Instanzvariable für ein `HashSet` deklariert, welches die Menge der lebendigen Zellen darstellt. Um das Board zu verwenden, sind in der Schnittstelle vier Methoden definiert. Zunächst kann das Board mit einem Konstruktor, welcher ein `HashSet` als Parameter entgegennimmt, initialisiert werden. Mithilfe der Instanzmethode `getAliveCells` kann auf die modifizierbare Menge der lebendigen Zellen zugegriffen werden. Die Instanzmethode `getActivePositions` ermittelt die Zellpositionen aller lebendigen Zellen und deren Nachbarn. Da das Zellengitter unbegrenzt groß ist, muss bestimmbar sein, in welchem Ausschnitt aktuell lebendige Zellen sind. Nur diese Zellen können sich durch die Regeln der Simulation in dem Zellengitter verändern. Die Methode `getNeighboursOfPosition` berechnet die Zellposition aller Nachbarn einer gegebenen Zellposition. Diese wird für die Anwendung der Regeln benötigt.

In Anhang A.B in Abbildung 9 sind in einem UML Klassendiagramm[53] die nun definierten Klassen sowie die Klassen `BoardController` und `App` dargestellt. Die gesamte Implementierung ist in Anhang A.A.A im Java-Paket `de.noahpeeters.gameoflife.classic` zu finden.

4.1.3 Implementierungsbeschreibung – ADT

Für die ADT Implementierung wird zunächst der abstrakte Datentyp für die beiden Datentypen Zellengitter und Zellposition definiert. Dabei wird wie in Abschnitt 2.1 vorgegangen.

Die Definition der Zellposition (`CellPosition`) ist in Listing 4 dargestellt. Die Zellposition hat nur zwei charakterisierende Operation. Mit der erzeugenden Operation kann eine Zellposition mit den entsprechenden x- und y-Koordinaten erstellt werden. Mit der produzierenden Operation `neighbours` wird die Menge der Nachbarn zurückgegeben. Die Richtung, in welche die Nachbarn angeordnet sind, wird nicht benötigt und kann daher weggelassen werden. Die einzige semantische Einschränkung ist, dass die Operation `neighbours` symmetrisch sein muss: Eine Zelle muss selbst Nachbar aller ihrer Nachbarn sein.

```
// Definition des abstrakten Datentyps für eine Zellposition
CellPosition
    // Definition der charakterisierenden Operationen
    // erzeugende Operation, um eine Zellposition zu erstellen
    create: ℤ × ℤ → CellPosition
    // produzierende Operation, um die Nachbarn zu erhalten
    neighbours: CellPosition → CellPosition*

    // Definition der Semantik
    ∀p ∈ CellPosition:
        // symmetrisch von neighbours
        ∀p' in neighbours(p): p ∈ neighbours(p')
```

Listing 4: Definition des abstrakten Datentyps `CellPosition` *(Quelle: Eigener Quelltext)*

Die Definition des Zellengitters (`InfiniteCellGrid`) ist in Listing 5 dargestellt. Es existiert eine erzeugende Operation, um ein Zellengitter ohne lebendige Zellen zu erstellen. Die observierende Operation `isAlive` kann genutzt werden, um zu überprüfen, ob eine Zelle lebendig ist. Mit den mutierenden Operationen `markAsAlive` und `markAsDead` wird der Zustand einer Zelle in dem Zellengitter verändert. Die produzierende Operation `createCopy` erstellt eine Kopie des Zellengitters. Im Kontext des mathematischen Modells ist diese Operation nicht notwendig, da es keinen veränderbaren Zustand gibt. Für die Übertragung in Java-Quelltext wird diese

[53] ISO Central Secretary. *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*. en. Standard ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization, 2005.

Operation jedoch benötigt. Andere Implementierungen, die ohne diese Operation auskommen, sind auch denkbar. Für das hier gewählte Vorgehen ist diese Operation jedoch notwendig und wird der Vollständigkeit halber auch in der Definition des abstrakten Datentyps aufgeführt. Die Operation `getActivePositions` ermittelt wie die gleichnamige Methode aus der Classic Implementierung die Zellpositionen aller lebendigen Zellen und deren Nachbarn.

```
// Definition des abstrakten Datentyps für ein InfiniteCellGrid
InfiniteCellGrid
    // Definition der charakterisierenden Operationen
    // erzeugende Operation, um ein InfiniteCellGrid zu erstellen
    emptyGrid: () → InfiniteCellGrid
    // observierende Operation zum Prüfen, ob eine Zelle lebendig ist
    isAlive: InfiniteCellGrid × CellPosition → boolean
    // mutierende Operation, um eine Zelle als lebendig zu markieren
    markAsAlive: InfiniteCellGrid × CellPosition → InfiniteCellGrid
    // mutierende Operation, um eine Zelle als tot zu markieren
    markAsDead: InfiniteCellGrid × CellPosition → InfiniteCellGrid
    // observierende Operation, um lebendige Zellposition
    // und deren Nachbarn zu erhalten
    getActivePositions: InfiniteCellGrid → CellPosition*
    // produzierene Operation, um eine Kopie anzulegen
    createCopy: InfiniteCellGrid → InfiniteCellGrid

    // Definition der Semantik
    ∀g ∈ InfiniteCellGrid, p ∈ CellPosition:
        isAlive(emptyGrid, p) = false
        isAlive(markAsAlive(g, p), p) = true
        isAlive(markAsDead(g, p), p) = false
        createCopy(g) = g
        getActivePositions(g) = { p′ ∈ CellPosition |
            isAlive(g, p′) ∨
            ∃p″ ∈ CellPosition: isAlive(g, p″) ∧ p′ ∈ neighbours(p″)
        }
```

Listing 5: Definition des abstrakten Datentyps `InfiniteCellGrid` *(Quelle: Eigener Quelltext)*

Die so definierten Datenstrukturen können jetzt nach der in Abschnitt 2.2 beschriebenen Methode in Java-Quelltext transformiert werden. Dafür existieren die in 2.2.1 vorgestellten Methoden. Das Nutzen von Enumerationen ist keine Option, da der Wertebereich der beiden Datentypen nicht sinnvoll als eine Liste diskreter Werte dargestellt werden kann. Die beiden anderen Optionen Klasse und Interface mit einer Klasse können beide verwendet werden. Um eine größere Differenzierung zwischen den beiden Implementierungen zu erreichen, wird für jeden abstrakten Datentyp ein Interface und eine Klasse definiert. Die Operationen werden nach

dem Verfahren aus Abschnitt 2.2.2 in Java implementiert. Es wird dabei auf eine semantische Dokumentation im Quelltext verzichtet.

In Listing 6 sind die so entstehenden Interfaces für die Zellposition und das Zellengitter dargestellt. Wie in Abschnitt 2.2.2 ausgeführt, können die erzeugenden Operationen nicht in den Interfaces dargestellt werden und fehlen hier entsprechend.

```
public interface CellPosition {
    Set<CellPosition> getNeighbours();
}

public interface InfiniteCellGrid {
    boolean isAlive(CellPosition position);
    void markAsAlive(CellPosition position);
    void markAsDead(CellPosition position);
    Set<CellPosition> getActivePositions();
    InfiniteCellGrid createCopy();
}
```

Listing 6: Implementierung mit Interfaces *(Quelle: Eigener Quelltext)*

Zusätzlich zu den Interfaces sind die Klassen `CartesianCellPosition` und `Board` definiert, die jeweils das entsprechende Interface und zusätzlich die erzeugende Operation implementieren. In Anhang A.B in Abbildung 8 sind in einem UML Klassendiagramm[54] die nun definierten Klassen und Schnittstellen dargestellt. Die gesamte Implementierung ist in Anhang A.A.A im Java-Paket `de.noahpeeters.gameoflife.adt` zu finden.

4.2 Interviewdesign

Das Interviewdesign hat einen maßgeblichen Einfluss auf die Ergebnisse und wird daher im Folgenden detailliert beschrieben.[55] Zunächst wird der Aufbau des Interviews und insbesondere die Wahl der eingesetzten Methoden begründet. Anschließend wird beschrieben, wie die Auswahl der inhaltlichen Aufgaben und der Probanden erfolgt, sodass die Ziele der Interviews erreicht werden können.

4.2.1 Aufbau

Während der Interviews werden drei verschiedene Arten von Informationen gesammelt, welche anschließend ausgewertet werden können. Diese Informationsarten geben den Aufbau der Interviews vor.

[54] ISO Central Secretary, *ISO/IEC 19501:2005*.
[55] Lindner, „Forschungsdesigns der Wirtschaftsinformatik“.

„Thinking Aloud“ Methode und Vorgehen Die Interviews werden mit der Methode *Thinking Aloud* durchgeführt. Die Probanden werden gebeten, während der Bearbeitung der Aufgaben ständig zu beschreiben, was sie machen und warum sie sich so entschieden haben. Dieser Methode ermöglicht es, einen Einblick in die Beweggründe und Probleme zu erhalten. Diese Informationen können anschließend in der Auswertung analysiert werden. Der Nachteil der Methode ist, dass keine Zeitmessungen möglich sind, da das Beschreiben der Beweggründe diese verfälschen würden. Daher können bei der Auswertung keine zuverlässigen Aussagen über die Effizienz gemacht werden. Der vergleichsweise hohe Aufwand der Thinking Aloud Methode ist vertretbar, da nicht sehr viele Interviews durchgeführt werden.[56]

Während der Bearbeitung der Aufgaben werden die Probanden beobachtet. Dabei wird protokolliert, wie die Probanden vorgehen und sich verhalten. Es wird insbesondere darauf geachtet, wie der Quelltext gelesen und gegebenenfalls verändert wird. Die so gesammelten Informationen über das Verhalten werden zusammen mit den Aussagen der Thinking Aloud Methode in der Auswertung analysiert und interpretiert.

Bewertung durch die Probanden Die Probanden sollen verschiedene Aussagen auf einer Likert-Skala von eins bis fünf bewerten.[57] Dabei steht eins für *ich stimme gar nicht zu* und fünf für *ich stimme voll zu*. Es wurde eine fünfstufige Bewertungsskala gewählt, da diese in diesem Anwendungsfall die besten Ergebnisse liefert. So haben Skalen mit weniger als vier Stufen eine geringere Zuverlässigkeit und Aussagekraft. Auf der anderen Seite bieten Skalen mit sieben oder mehr Stufen nur geringe Verbesserungen gegenüber Skalen mit weniger Stufen. Skalen mit mehr Stufen können aufgrund der Schwierigkeit die einzelnen Stufen zu differenzieren zu neuen Messfehlern führen.[58]

Wenn eine Aussage in Bezug auf beide Implementierungen bewertet werden soll, dann werden die Probanden zunächst die Aufgabe für beide Implementierungen bearbeiten und anschließend die Aussagen jeweils getrennt bewerten. Dieses Vorgehen wurde gewählt, da man bei Bewertungsskalen immer implizit eine Referenz benötigt und das vorherige Kennenlernen aller zu bewertenden Objekte diese Voraussetzung schafft.[59] Dieses Vorgehen zwingt die Probanden, die beiden Implementierungen erneut zu reflektieren, sodass weitere Daten für die Thinking Aloud Methode gesammelt werden können. Die Ergebnisse der Bewertungsskalen werden ordinalskaliert betrachtet. Aufgrund der Erhebung in Form einer fünfstufigen Bewertungsskala ist dies die

[56] Clayton Lewis. *Using the"thinking-aloud"method in cognitive interface design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

[57] Rensis Likert. „A technique for the measurement of attitudes.“ In: *Archives of psychology* (1932).

[58] Luis M Lozano, Eduardo García-Cueto und José Muñiz. „Effect of the number of response categories on the reliability and validity of rating scales“. In: *Methodology* 4.2 (2008), S. 73–79.

[59] Jon A Krosnick, Allyson L Holbrook und Penny S Visser. „Optimizing brief assessments in research on the psychology of aging: A pragmatic approach to self-report measurement“. In: *When I'm* 64 (2006), S. 231.

genaueste Betrachtung.[60] In der Auswertung können dann die Bewertungen der Probanden mit den Ergebnissen der Thinking Aloud Methode und den Beobachtungen kombiniert werden.

Finaler Quelltext Die Probanden werden an verschiedenen Stellen den Quelltext des Basisprojektes modifizieren. Der so entstehende Quelltext stellt die dritte Informationsart dar. Der Quelltext kann zum Beispiel Aufschluss über die Korrektheit der Änderungen geben.

4.2.2 Inhaltliche Aufgaben

Zunächst sollen die Probanden ihre Kenntnisse selbst einschätzen. Dazu gibt es jeweils eine Frage zu den *Java*-Kenntnissen und eine zu den Vorkenntnissen bezüglich *Conways Spiel des Lebens*. Diese werden in der Auswertung verwendet, um Unterschiede zwischen den einzelnen Ergebnissen der Probanden erklären zu können. So könnte zum Beispiel mehr Vorwissen des fachlichen Kontextes eine Modifikation vereinfachen.

Auch wenn die Probanden Conways Spiel des Lebens bereits aus anderen Kontexten kennen, sind jedem der Probanden die beiden konkreten Implementierungen des Basisprojektes nicht bekannt. Der zuvor beschriebene Quelltext wird den Probanden erstmalig zu Beginn des Interviews zur Verfügung gestellt. Der Vorteil ist, dass die Probanden nicht durch Vorwissen beeinflusst werden. Der Nachteil ist, dass mit der verwendeten Methode keine Aussage über bekannten Quelltext möglich ist, sondern nur das Vorwissen des fachlichen Kontextes betrachtet werden kann.

Anschließend erhalten die Probanden drei Aufgaben. Diese sollen jeweils für beide Implementierungen durchgeführt werden. Jeder Proband soll immer mit derselben Implementierung beginnen, wobei die eine Hälfte der Probanden immer mit der ADT Implementierung (im Folgenden *ADT-Gruppe*) und die andere Hälfte immer mit der Classic Implementierung (im Folgenden *Classic-Gruppe*) beginnt. Daher wird eine gerade Anzahl an Probanden an den Interviews teilnehmen, um die Auswertung im Anschluss zu vereinfachen.

Aufgabe 1 In der ersten Aufgabe sollen sich die Probanden für einen ersten Einblick den gesamten Quelltext ansehen. Hierbei ist vor allem interessant, wie die Probanden das erste Mal auf den Quelltext reagieren. Außerdem wird so gewährleistet, dass die Probanden bereits ein erstes Verständnis vom Aufbau des Projektes haben, damit die anschließenden Aufgaben sinnvoll durchgeführt werden können. Anschließend sollen die Probanden bewerten, wie einfach der Quelltext zu lesen ist, wie intuitiv der Quelltext ist und wie einfach er sich vermutlich modifizieren lässt. Dabei zielen die Bewertungsfragen auf die in Tabelle 1 dargestellten Codequalitätskriterien aus Abschnitt 3.1 ab.

[60] David J Hand. „Statistics and the theory of measurement“. In: *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 159.3 (1996), S. 445–473.

Bewertung	Kriterium
Einfach	Analysierbarkeit
Intuitiv	Analysierbarkeit
Modifizierbar	Modifizierbarkeit

Tabelle 1: Bewertungsfragen und Codequalitätskriterien von Aufgabe 1 *(Quelle: Eigene Auflistung)*

Aufgabe 2 In der zweiten Aufgabe sollen die Probanden eine kleine Änderung vornehmen. Das Programm soll die Gesamtzahl der lebendigen Zellen nach jeder Iteration ausgeben. Für diese Modifikation ist keine Anpassung der Datenstruktur im Speicher notwendig. Anschließend sollen die Probanden bewerten, wie einfach die Änderung durchzuführen war, wie einfach die zu modifizierenden Quelltextstellen identifiziert werden konnten und ob die Struktur des bestehenden Quelltextes eine Hilfe oder ein Hindernis darstellte. Dabei zielen die Bewertungsfragen auf die in Tabelle 2 dargestellten Codequalitätskriterien aus Abschnitt 3.1 ab.

Bewertung	Kriterium
Einfach	Modifizierbarkeit
Ort	Analysierbarkeit
Hilfe	Modifizierbarkeit

Tabelle 2: Bewertungsfragen und Codequalitätskriterien von Aufgabe 2 *(Quelle: Eigene Auflistung)*

Aufgabe 3 In der letzten Aufgabe sollen die Probanden eine größere Änderung vornehmen. Das Programm soll ausgeben, wie oft eine vorgegebene Zelle während der gesamten Laufzeit in den durchgeführten Iterationen der Simulation lebendig war. Diese Information ist noch nicht in der Datenstruktur vorhanden, sodass diese angepasst werden muss. Die Aufgabe ist so spezifiziert, dass diese Information für zwei vorgegebene Zellen zu ermitteln ist und ausgegeben werden soll. Die Zellen wurden so gewählt, dass keine der beiden Zellen Teil der Startkonfiguration der Zellen im Basisprojekt ist, sodass ein „Off-By-One Fehler“ möglich ist, ohne dass dieser auffällt.[61] Anschließend sollen die Probanden dieselben Aussagen wie bei Aufgabe 2 bewerten.

Abschließend sollen die Probanden zwei weitere Aussagen bewerten. Zunächst soll bewertet werden, ob die Unterschiede zwischen den Implementierungen deutlich wurden. Dann soll angegeben werden, welche der beiden Implementierungen präferiert wurde. Diese Fragen sollen vor allem für eine nachträgliche Reflexion der beiden Implementierungen sorgen, welche zusätzlich durch die Thinking Aloud Methode ausgewertet werden kann.

[61] Manu Jose und Rupak Majumdar. „Cause Clue Clauses: Error Localization Using Maximum Satisfiability“. In: *SIGPLAN Not.* 46.6 (Juni 2011), S. 437–446. ISSN: 0362-1340. DOI: 10.1145/1993316.1993550.

Die genauen Aufgabenbeschreibungen, wie sie die Probanden während der Interviews als PDF erhalten haben, sind in Anhang A.C zu finden.

4.2.3 Akquise der Probanden

Die Probanden müssen nicht über besonderen Fähigkeiten verfügen. Insbesondere ist es nicht notwendig, dass die Probanden bereits Vorkenntnisse zu den Themen *abstrakte Datentypen*, *Codequalität* oder zu der im Basisprojekt umgesetzten Simulation *Conways Spiel des Lebens* haben. Die einzige Anforderung an die Probanden sind ausreichende Java-Kenntnisse, um an dem Basisprojekt Modifikationen durchführen zu können. Dazu gehören insbesondere die Nutzung von Klassen und Interfaces sowie die Nutzung elementarer und vordefinierter Datenstrukturen in Java wie `HashSets`.

Die Akquise der Probanden wurde unter Informatikstudierenden der Nordakademie durchgeführt, da diese über ausreichende Java-Kenntnisse verfügen und damit die geforderten Voraussetzungen erfüllen.

5 Auswertung der Interviews

Es wurden nach dem in Abschnitt 4 beschriebenen Verfahren sechs Interviews durchgeführt. Die Protokolle der Interviews finden sich im Anhang A.D.A – A.D.F. Dabei haben die Probanden eins bis drei die jeweiligen Aufgaben mit der Classic Implementierung begonnen, während die Probanden vier bis sechs mit der ADT Implementierung begonnen haben. Die finalen Projekte mit allen Änderungen der Probanden sind in Anhang A.A.B zu finden.

In diesem Abschnitt werden die Ergebnisse aufgabenweise analysiert. Dabei werden die Ergebnisse noch nicht auf die in Abschnitt 3.1 definierten Codequalitätsmerkmale bezogen, sondern zunächst davon unabhängig betrachtet und ausgewertet.

Für die Auswertung werden alle der in Abschnitt 4.2.1 beschriebenen Arten von Informationen verwendet. Falls die Probanden eine Quelltextmodifikation durchführen sollten, wird zunächst die Änderung am Quelltext sowie das Endergebnis beschrieben und analysiert. Anschließend wird die Protokollanalyse durchgeführt. Diese Ergebnisse werden dann mit den Bewertungen durch die Probanden abgeglichen. In der Mehrzahl der Fälle lassen sich die Ergebnisse der Protokollanalyse so untermauern. Bei Abweichungen wird auf diese eingegangen.

Zur Visualisierung der Bewertungen durch die Probanden werden Säulendiagramme verwendet. Auf der x-Achse sind die fünf Antwortoptionen eins bis fünf von links nach rechts aufgetragen. Auf der y-Achse ist die absolute Häufigkeit der Bewertungen abgebildet. Wenn in einer Aufgabe eine Aussage für beide Implementierungen getrennt bewertet wurde, sind die entsprechenden Diagramme übereinander angeordnet und in derselben Farbe dargestellt.

5.1 Auswertung der Vorbereitungsphase

Der Vorbereitungsschritt diente vor allem dazu, den Probanden eine kurze Einweisung zu geben und das Basisprojekt einzurichten. In diesem Teil des Interviews wurden noch keine inhaltlichen Aufgaben gestellt. Die Probanden sollten sich als Vorbereitung in zwei Kategorien einschätzen. Die Ergebnisse sind in Abbildung 3 dargestellt.

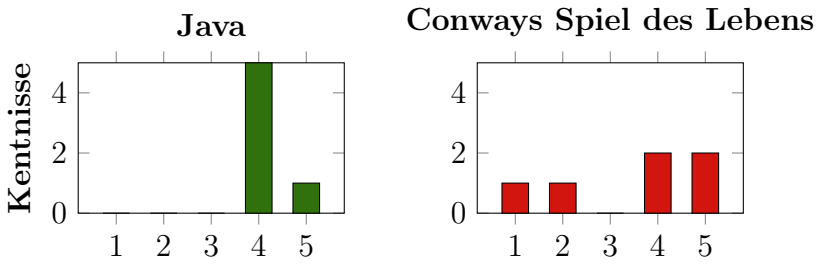

Abbildung 3: Selbsteinschätzung des Vorwissens *(Quelle: Eigene Darstellung)*

Java-Kenntnisse Die Selbsteinschätzung der Java-Kenntnisse hat ergeben, dass alle Probanden ihre Kenntnisse einheitlich gut bewertet haben. Es gibt kaum eine Streuung in der Verteilung. Es können daher keine Aussagen über Unterschiede zwischen verschiedenen Kenntnisständen getroffen werden.

Conways Spiel des Lebens Die Selbsteinschätzung der Vorkenntnisse von Conways Spiel des Lebens sind dagegen über die gesamte Skala verstreut. Vier Probanden war die Simulation bereits bekannt[62]. Die anderen beiden Probanden gaben an, noch kein oder nur ein geringes Vorwissen zu haben[63]. Ein möglicher Einfluss dieser unterschiedlichen Voraussetzungen wird in der Gesamtbewertung untersucht.

5.2 Auswertung Aufgabe 1

In dieser Aufgabe sollten die Probanden den Quelltext das erste Mal lesen. Es wurden noch keine Änderungen gefordert. Die Bewertung der einzelnen Aussagen ist in Abbildung 4 dargestellt. Insgesamt gaben die Probanden an, dass die ADT Implementierung einfacher zu lesen und intuitiver war. Diese Angaben konnten auch durch die Protokollanalyse bestätigt werden. Die vermutete Modifizierbarkeit der ADT Implementierung wurde ebenfalls besser bewertet.

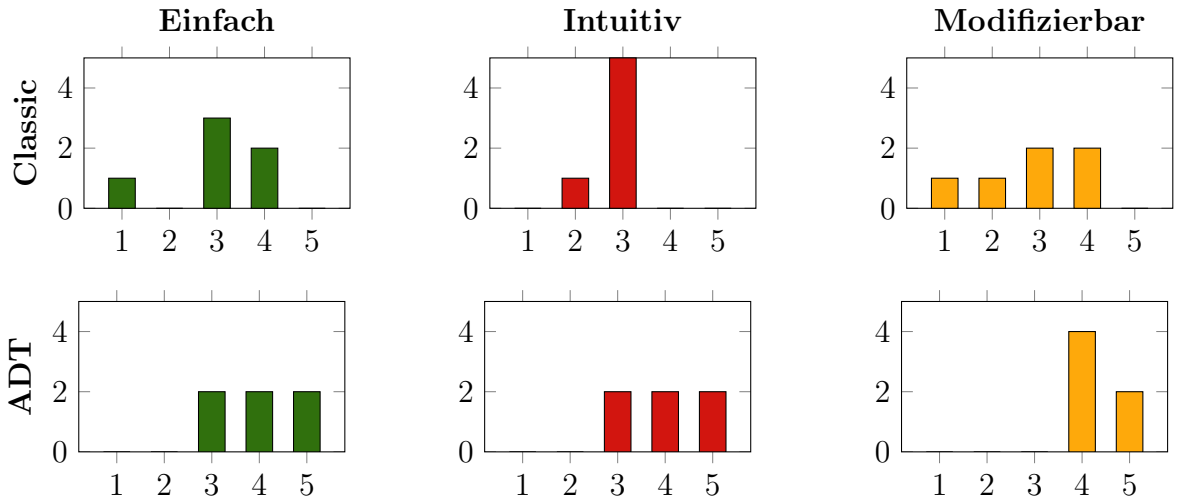

Abbildung 4: Bewertung Aufgabe 1 *(Quelle: Eigene Darstellung)*

Einfach Einige der Probanden merkten während der Bearbeitung der Aufgabe an, dass der ADT Quelltext lesbarer sei[64]. So wurde zum Beispiel konkret angemerkt, dass verknüpfte Methodenaufrufe wie `getAliveCells().add()` in der Classic Implementierung weniger sprechend

[62] Proband 1, 2, 3 und 4
[63] Proband 5 und 6
[64] zum Beispiel Proband 1, 2, 3 und 6

seien als `markAsAlive()`[65]. Die Methode `step` der Classic Implementierung wurde als schwer verständlich wahrgenommen[66]. Dazu passt auch die Bewertung durch die Probanden selbst. Die ADT Implementierung wurde insgesamt als einfacher eingestuft.

Intuitiv Es konnte beobachtet werden, dass die Probanden durch die Benennung der Methoden deren Semantik in der Regel schnell erfassen konnten[67]. Es wurde jedoch auch kritisiert, dass die Benennung nicht immer intuitiv sei beziehungsweise eine bessere Benennung möglich wäre. Die Methode `isAlive` wäre nicht intuitiv, da diese zur Schnittstelle des Boards gehört, sich der Rückgabewert jedoch auf eine Zelle bezöge[68]. Dennoch konnten die Probanden beim ersten Lesen die Semantik bereits korrekt erkennen. Außerdem wurde kritisiert, dass die Benennung der Interfaces und Klassen keine intuitive Zuordnung der Interfaces zu den Klassen zulässt[69]. Die Bewertung der Probanden stimmt mit den Ergebnissen der Protokollanalyse überein. Die ADT Implementierung wurde als intuitiver eingestuft.

Modifizierbar Die vermutete Modifizierbarkeit der ADT Implementierung wurde deutlich besser bewertet als die der Classic Implementierung. Dabei wurde angemerkt, dass die Vorteile vermutlich vor allem bei größeren Projekten zum Tragen kämen[70]. Die Einschätzung, dass sich die ADT Implementierung einfacher modifizieren ließe, kann in den folgenden Abschnitten zur Auswertung der anderen Aufgaben überwiegend bestätigt werden.

5.3 Auswertung Aufgabe 2

In dieser Aufgabe sollten die Probanden zunächst eine kleine Änderung am Quelltext durchführen. Nach jeder Iteration der Simulation sollte die Anzahl der lebendigen Zellen ausgegeben werden. Bei der Bearbeitung der ADT Implementierung haben fünf Probanden – abgesehen von Funktionsbenennungen – dieselbe Lösung erarbeitet: Es wurde eine neue Methode im Board und im Interface angelegt, welche die Anzahl der lebendigen Zellen zurückgibt. Das Ergebnis konnte so direkt im Controller ausgegeben werden. Lediglich ein Proband hat eine andere Lösung entworfen. Dabei wurde die bestehende Methode `getActivePositions` aufgerufen und das Ergebnis nach lebendigen Zellen gefiltert und anschließend gezählt.[71] Nach der semantischen Definition aus Abschnitt 4.1.3 liefert dieses Vorgehen immer das richtige Ergebnis. Die Begrün-

[65] zum Beispiel Proband 2, 4 und 6
[66] zum Beispiel Proband 1
[67] zum Beispiel Proband 2, 3, 4 und 5
[68] Proband 6
[69] Proband 1
[70] zum Beispiel Proband 6
[71] Proband 6

dung des Probanden für diese Lösung war, dass es weniger Aufwand sei als eine Erweiterung der Schnittstelle vorzunehmen.

Bei der Bearbeitung der Classic Implementierung ist ein signifikanter Unterschied zwischen den beiden Gruppen aufgefallen. Alle Probanden der Classic-Gruppe haben die bestehende Methode `getAliveCells` im Controller verwendet, um die Menge aller lebendigen Zellen zu erhalten und daraus die Anzahl zu bestimmen. Die ADT-Gruppe hingegen hat jeweils den Ansatz der ADT Implementierung wiederverwendet und eine diesem Paradigma folgende Änderung durchgeführt. Im Detail bedeutet dies, dass zwei der Probanden eine neue Methode im Board angelegt haben, um die Anzahl der lebendigen Zellen zu ermitteln. Die Begründung der Probanden dafür war die Unabhängigkeit vom internen Zustand.[72] Der letzte Proband hat die bestehenden Methoden verwendet, um die Anzahl wieder im Controller zu ermitteln. Die Begründung des Probanden war die höhere Flexibilität der Methode in der Zukunft.[73]

Hieran ist besonders interessant, dass die zuerst verwendete Implementierung vermutlich eine Auswirkung auf das weitere Vorgehen hat. Dies zeigt, dass die Wahl der verwendeten Implementierung auch Auswirkungen auf andere Teile einer Software haben kann, da die Entwickler die Methoden weiter anwenden. Dies zeigt aber auch, dass die Probanden sich sehr schnell an die ADT Implementierung anpassen konnten und diese auch weiterhin verwendet haben. Über die Langfristigkeit dieser Effekte kann mit den Daten der kurzen Interviews jedoch keine Aussage getroffen werden.

Insgesamt waren alle Probanden in der Lage, die Änderung ohne viel Aufwand fehlerfrei durchzuführen.

Auch in dieser Aufgabe sollten die Probanden wieder drei Aussagen bewerten. Diese Ergebnisse sind in Abbildung 5 dargestellt. Sowohl die Auswertung der Protokolle als auch die Bewertung der Probanden hat ergeben, dass die Classic Implementierung für die Bearbeitung dieser Aufgabe der ADT Implementierung überlegen ist.

[72] Proband 4 und 5
[73] Proband 6

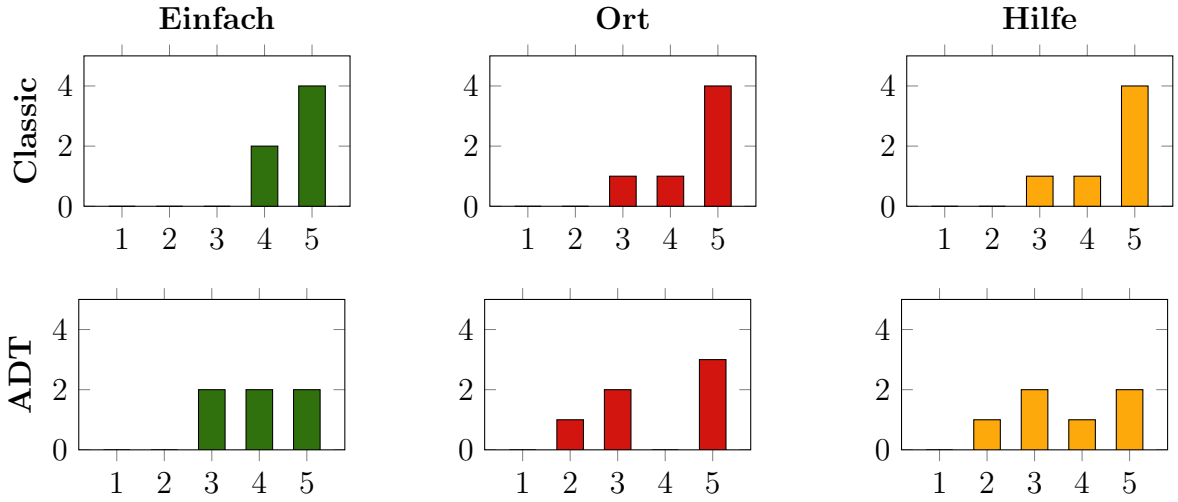

Abbildung 5: Bewertung Aufgabe 2 *(Quelle: Eigene Darstellung)*

Einfach Das größte Hindernis der ADT Implementierung für die Modifizierbarkeit waren die Interfaces, da diese einen höheren Aufwand bei Anpassungen erfordern. So haben mehrere Probanden zunächst versucht bestehende Methoden wiederzuverwenden, konnten aber keine identifizieren, die das gesuchte Verhalten abbilden[74]. Dies spiegelt sich auch in der besseren Bewertung der Classic Implementierung durch die Probanden wider.

Ort Die Verwendung von Interfaces bei der ADT Implementierung erschwerte den Probanden das Identifizieren aller Stellen, die verändert werden mussten. Diese notwendigen Anpassungen wurden daher teilweise zunächst vergessen[75]. Auch die Bewertung durch die Probanden hat dieses Ergebnis bestätigt.

Hilfe Der bestehende Quelltext der ADT Implementierung wurde teilweise als Hindernis wahrgenommen[76]. Dagegen wurde bei der Classic Implementierung der Quelltext als Hilfe wahrgenommen, da bereits sehr ähnlicher Quelltext in der Methode `print` vorhanden ist[77]. Dazu passt auch die Bewertung durch die Probanden selbst. Die Classic Implementierung wurde insgesamt als hilfreicher eingestuft.

5.4 Auswertung Aufgabe 3

In dieser Aufgabe musste eine umfangreichere Quelltextmodifikation durchgeführt werden. Nach jeder Iteration der Simulation sollte die Anzahl der Lebenszyklen einzelner Zellen ausgegeben

[74] zum Beispiel Proband 2 und 4
[75] zum Beispiel Proband 1 und 2
[76] zum Beispiel Proband 1 und 2
[77] zum Beispiel Proband 1, 2 und 3

werden. Es wurde von allen Probanden erkannt, dass die bestehende Datenstruktur nicht ausreichend ist, um die neue Anforderung umzusetzen. Fünf der Probanden haben sich entschieden eine `HashMap<CellPosition, Integer>` zu verwenden, um für jede Zelle zu zählen, wie oft diese lebendig war. Der letzte Proband war sich unsicher, wie die neuen Daten gespeichert werden können. Der Proband hat zunächst nicht erkannt, dass das Erweitern der Zellposition um einen Zähler aufgrund dessen Verwendung kein sinnvoller Lösungsansatz ist. Nach einem entsprechenden Hinweis hat sich auch dieser Proband für eine Map entschieden.[78] Die genaue Implementierung unterscheidet sich jedoch bei den einzelnen Probanden.

Bei der ADT Implementierung haben sich alle Probanden dafür entschieden, die HashMap in dem Board zu speichern. Fünf der Probanden haben anschließend die bestehende Methode `markAsAlive` erweitert, sodass diese den Zähler für die entsprechende Zelle inkrementiert. Der letzte Proband hat hingegen die Schnittstelle um die neue Methode `cellBecameAlive` erweitert, welche das Inkrementieren des Zählers übernimmt.[79]

Bei der Classic Implementierung gab es zwei verschiedene Ansätze. Zwei der Probanden haben wie bei der ADT Implementierung das HashSet im Board gespeichert.[80] Einer dieser Probanden hat eine Get-Methode eingeführt, um eine Referenz auf die neue Datenstruktur zu erhalten, sodass der Zähler im Controller inkrementiert werden kann[81]. Der andere Proband hat das Board um die Methode `reviveCell` erweitert, welche wie die Methode `markAsAlive` in der ADT Implementierung arbeitet.[82]

Die restlichen Probanden haben die HashMap direkt im Controller gespeichert, sodass ein direkter Zugriff möglich ist. Anschließend wurde die Methode `step` erweitert, um den Zähler zu inkrementieren. Einige Probanden haben außerdem den Konstruktor erweitert, sodass auch die Startkonfiguration bei der Zählung berücksichtigt wird.

Einige der finalen Implementierungen enthalten Fehler. Ein Proband hat zunächst eine inkorrekte Lösung erstellt. Bei der Classic Implementierung wurde die `HashMap` mit den Zählern für alle Zellen aufgrund eines Fehlers bei jeder Iteration der Simulation neu angelegt. Durch den Vergleich der Ergebnisse der beiden Implementierungen ist der Fehler aufgefallen und konnte korrigiert werden[83]. Damit haben alle Probanden Implementierungen erstellen können, die für die beiden Testfälle der angegebenen Zellen $(x = 0, y = 0)$ und $(x = 3, y = 2)$ das richtige Ergebnis geliefert haben.

Einige der Lösungen enthalten dennoch Fehler, die in zwei Kategorien eingeteilt werden können. Die erste Kategorie bilden Fehler, die nicht bei den vorgegebenen Zellpositionen, sondern

[78] Proband 5
[79] Proband 1
[80] Proband 1 und 2
[81] Proband 1
[82] Proband 2
[83] Proband 2

bei anderen Eingabeparametern auftreten. Die zweite Kategorie bilden Methoden, bei denen die semantische Definition nicht mit der Implementierung übereinstimmt. Dies kann bei weiteren Modifikationen zu Fehlern führen.[84]

Für die Fehler der ersten Kategorie stellt in diesem Fall die Startkonfiguration der Simulation die Eingabe dar. In dieser Kategorie konnte ein Fehler identifiziert werden. Bei einigen der Implementierungen wird die Startkonfiguration bei der Zählung der lebendigen Zellen nicht mitbetrachtet. Bei einem der Probanden ist dieser Fehler bei beiden Implementierungen aufgetreten, da hier das Board um eine weitere Methode erweitert wurde, die den Zähler inkrementiert. Diese Methode wird nicht für die Startkonfiguration aufgerufen[85]. Der andere Proband hat denselben Fehler in der Classic Implementierung gemacht. In der ADT Implementierung ist das Problem nicht aufgetreten, da hier die Methode `markAsAlive` erweitert wurde, sodass das Board immer in einem konsistenten Zustand ist[86].

In der zweiten Fehlerkategorie konnten zwei Fehler identifiziert werden. In der ADT Implementierung stellt die Schnittstelle die Methode `copy` zur Verfügung. In der aktuellen Implementierung wird das alte Board nur während des Durchführens einer Iteration verwendet und anschließend verworfen. In der Implementierung eines Probanden wird von der neuen Datenstruktur zum Speichern der Anzahl der Lebenszyklen keine Kopie angelegt. Sowohl das Originalboard als auch die neue Boardkopie teilen sich eine Referenz auf dieselben Datenstruktur[87]. Würde nun anschließend das alte Board weiterverwendet werden, kann es zu inkonsistenten Datenzuständen kommen. Dieses Problem wird auch in der Literatur erkannt. Hier wird ausgeführt, dass das Trennen von Schnittstelle und Implementierung diese Gefahr sichtbarer macht.[88] Dies konnte mit den Ergebnissen der Interviews nicht überprüft werden, da nur zwei der Probanden die neue Datenstruktur in der Classic Implementierung im Board angelegt haben[89]. Damit entstand die Problematik bei den restlichen Probanden nicht. Der eine Proband erstellte die Kopie weiterhin im Controller. Auch hier wurde keine Kopie der Datenstruktur angelegt.[90]. Der andere hatte analog zu der ADT Implementierung eine Methode `cloneBoard` definiert, die eine Kopie der neuen Datenstruktur anlegt[91].

Der zweite Fehler tritt bei der ADT Implementierung bei zwei Probanden auf. Beide haben die Methode `markAsAlive` erweitert, um den Zähler der Lebenszyklen der entsprechenden Zelle zu inkrementieren. Dabei wird nicht geprüft, ob die Zelle in der vorherigen Iteration bereits

[84] Baishakhi Ray u. a. „Detecting and characterizing semantic inconsistencies in ported code“. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, S. 367–377.
[85] Proband 1
[86] Proband 5
[87] Proband 5
[88] Sivilotti und Lang, „Interfaces First (and Foremost) with Java“.
[89] Proband 1 und 2
[90] Proband 1
[91] Proband 2

lebendig war. Wird die Methode für eine bereits lebendige Zelle aufgerufen, dann wird der Zähler fälschlicherweise inkrementiert, da die Methode nicht mehr idempotent ist[92]. Aufgrund der aktuellen Struktur des Boards tritt dieser Fehler nicht auf. Dennoch gibt es in der Definition des abstrakten Datentyps und auch in der Benennung der Methode keine Einschränkung für das mehrfache Aufrufen. Damit ist die neue Implementierung nicht mehr kompatibel zu der definierten Schnittstelle des abstrakten Datentyps.

Die Probanden sollten erneut die drei Aussagen der ersten Änderung für die umfangreichere Änderung bewerten. Diese Ergebnisse sind in Abbildung 6 dargestellt. Zwischen den Ergebnissen der Bewertung durch die Probanden und der Protokollanalyse gibt es teilweise signifikante Unterschiede. Insgesamt ergibt die Auswertung, dass die ADT Implementierung der Classic Implementierung überlegen ist.

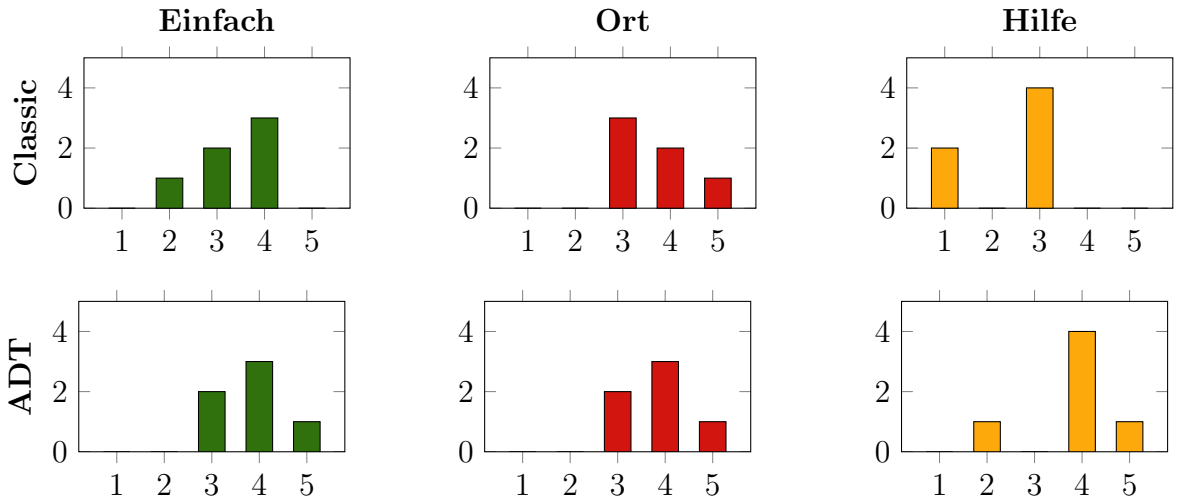

Abbildung 6: Bewertung Aufgabe 3 *(Quelle: Eigene Darstellung)*

Einfach Die Probanden haben sich sehr unterschiedlich zu den Implementierungen geäußert. Auf der einen Seite wurden die Änderungen bei beiden Implementierungen als einfach und intuitiv beschrieben[93]. Auf der anderen Seite wurden die Änderungen an der ADT Implementierung als aufwändiger eingestuft[94]. Dennoch gab es bei der Bewertung der Aussage keine ausgeprägtere Streuung als bei anderen Aussagen. Insgesamt wurde die ADT Implementierung als einfacher zu modifizieren bewertet.

Ort Während der Interviews sind bei der Classic Implementierung Probleme beim Identifizieren der zu ändernden Stellen erkennbar geworden. Die Hälfte der Probanden hat zunächst begonnen

[92] Proband 3 und 4
[93] zum Beispiel Proband 1
[94] zum Beispiel Proband 6

das HashSet im Board oder Controller zu speichern. Während der Durchführung der dafür notwendigen Änderungen sind Hindernisse aufgetreten, sodass der gesamte Ansatz verworfen werden musste und das HashSet in die jeweils andere Klasse verschoben wurde[95]. Bei der ADT Implementierung sind diese Probleme nicht aufgetreten. Bei der Bewertung durch die Probanden ist dieser deutliche Vorteil der ADT Implementierung gegenüber der Classic Implementierung nicht deutlich geworden. Es gab lediglich eine minimale Tendenz zu einer besseren Bewertung für die ADT Implementierung. Diese Tendenz ist aber deutlich weniger ausgeprägt als bei den anderen Aussagen, obwohl hier mit der Protokollanalyse ein sehr großer Unterschied zwischen den beiden Implementierungen festgestellt wurde.

Hilfe Mehrere Probanden empfanden, dass die Struktur der Classic Implementierung die Erweiterung verkompliziert hätte[96]. Drei der Probanden, die die HashMap im Controller gespeichert haben, hätten eine Implementierung im Board bevorzugt, sich jedoch aufgrund der Struktur dagegen entschieden[97]. Zwei dieser Probanden haben den entstandenen Quelltext anschließend als *unschön* beschrieben[98]. Dies wird auch in der Bewertung durch die Probanden deutlich. Die ADT Implementierung wurde als wesentlich besser bewertet.

5.5 Auswertung der Abschlussphase

Abschließend sollten die Probanden noch zwei weitere Aussagen bewerten. Die Ergebnisse sind in Abbildung 7 dargestellt.

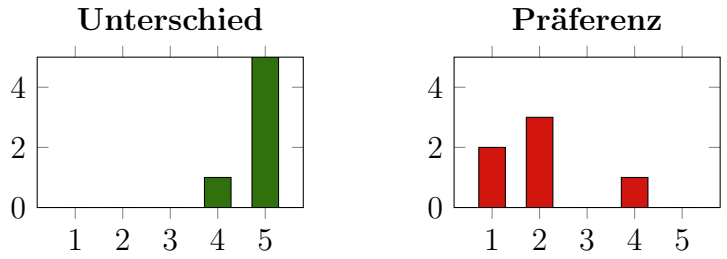

Abbildung 7: Abschlussfragen *(Quelle: Eigene Darstellung)*

Unterschied Die Probanden sollten bewerten, ob die Unterschiede zwischen den beiden Implementierungen deutlich wurden. Bei der Bearbeitung der Aufgaben haben die Probanden bereits sehr früh die wichtigsten Unterschiede erkannt. Insbesondere beim Erklären des Quelltextes in Aufgabe 1 wurden die Unterschiede benannt. Es hat jedoch keiner der Probanden von sich

[95] Proband 2, 3 und 6
[96] zum Beispiel Proband 2 und 3
[97] Proband 3, 4 und 6
[98] Proband 4 und 6

aus die Kerneigenschaft abstrakter Datentypen – also die Definition des Datentyps mithilfe der Operationen – ganz oder teilweise benannt. Die Bewertungen der Probanden stimmen dabei mit den Ergebnissen der Protokollanalyse überein.

Präferenz Die Probanden sollten angeben, welche der beiden Implementierungen sie bevorzugen. Dabei steht eins für die ADT Implementierung und fünf für die Classic Implementierung. Er wurde mehrfach angegeben, dass Prinzipien der Softwareentwicklung wie sprechende Namen, Datenhoheit und das Geheimnisprinzip in der ADT Implementierung besser umgesetzt sind[99]. Außerdem wurde angegeben, dass bei einem größeren Projektumfang die ADT Implementierung geeigneter sei[100]. Dennoch wurde auch hier noch einmal darauf hingewiesen, dass die Interfaces Modifikationen erschwert hätten oder in dem verwendeten Projekt unnötig seien[101]. Dieses Ergebnis wird auch in der Bewertung durch die Probanden sichtbar. Fünf der Probanden bevorzugten die ADT Implementierung. Ein Proband hat angegeben, dass die Classic Implementierung bei kleinen Projekten wie in diesem Fall vermutlich besser geeignet sei. Bei großen Projekten würde auch dieser Proband die ADT Implementierung bevorzugen.[102]

[99] Proband 2 und 3
[100] Proband 2 und 6
[101] Proband 1, 2 und 6
[102] Proband 4

6 Beantwortung der Kernfrage

Die Kernfrage der Arbeit untersucht, ob der Einsatz abstrakter Datentypen die Codequalität verbessert. In Abschnitt 3.1 wurden die Kriterien bestimmt, an denen dies gemessen werden kann. Im Folgenden werden mit den Ergebnissen der durchgeführten Interviews aus Abschnitt 5 diese einzelnen Kriterien analysiert und bewertet.

6.1 Funktionale Vollständigkeit

Die funktionale Vollständigkeit betrachtet, ob die Software die spezifizierten Aufgaben und Nutzerziele erfüllt.[103] Im Fall der Interviews gab es keine Nutzerziele, da es sich nicht um ein Projekt mit Nutzern handelt. Die spezifizierten Aufgaben des Basisprojektes wurden den Probanden in der Aufgabenbeschreibung der Interviews wie in Anhang A.C mitgeteilt. Die geforderten Modifikationen waren nicht umfangreich und ließen sich mit wenigen Zeilen Quelltext umsetzten. Insbesondere durch das direkte Interview und das Reden über die Aufgaben ist das Auslassen von (Teil-)Spezifikationen unwahrscheinlich. Entsprechend haben alle Probanden die geforderten Spezifikationen in beiden Implementierungen vollständig umgesetzt. Daher kann mithilfe der in den Interviews gesammelten Daten weder eine Verbesserung noch eine Verschlechterung der Codequalität in Bezug auf die funktionale Vollständigkeit festgestellt werden.

6.2 Funktionale Korrektheit

Die funktionale Korrektheit betrachtet, ob die Software das richtige Ergebnis mit der geforderten Präzision liefert.[104] In Aufgabe 2 wurden die geforderten Spezifikationen von allen Probanden korrekt implementiert. Dies liegt vermutlich vor allem an dem geringen Umfang der Problemstellung, die sich in wenigen Zeilen Quelltext lösen ließ. Dies spiegelt sich auch in den Bewertungen der Probanden in Abschnitt 5.3 wider.

In Aufgabe 3 war eine umfangreichere Modifikation gefordert. Dabei sind wie in Abschnitt 5.4 beschrieben verschiedene aufgetretene und potenzielle Fehler implementiert worden. In der Classic Implementierung ist insgesamt eine größere Anzahl an fehlerhaften Änderungen vorgenommen worden. Die ADT Implementierung hat hingegen im Quelltext mehrere möglicherweise unerwartete Seiteneffekte, die in der Zukunft bei weiteren Modifikationen die Ursache für neue Fehler sein können.[105]

[103] ISO Central Secretary, *ISO/IEC 25010:2011*, S. 11.
[104] Ebd., S. 11.
[105] Ray u. a., „Detecting and characterizing semantic inconsistencies in ported code“.

6.3 Modularität

Die Modularität betrachtet, ob das Softwaresystem – beziehungsweise durch den Fokus auf Codequalität der Quelltext – in Komponenten aufgeteilt ist, sodass Änderungen möglichst geringe Auswirkungen auf andere Komponenten haben.[106]

Die Modularität von Datentypen kann in zwei Richtungen betrachtet werden: Wie gut lässt sich der Datentyp anpassen, ohne den Rest der Anwendung anpassen zu müssen (siehe Aufgabe 3), und wie gut lässt sich der Rest der Anwendung ändern, ohne dass der Datentyp verändert werden muss (siehe Aufgabe 2). Bei der ersten Richtung ist es notwendig, die charakterisierenden Operationen geeignet zu definieren. Dadurch kann eine größere Flexibilität erreicht werden.[107]

In Aufgabe 2 haben fast alle Probanden neben dem Controller auch das Board angepasst, um die Ausgabe zu implementieren, weil die gewünschte Funktionalität noch nicht Teil der definierten Schnittstelle war. Auch in der Classic Implementierung haben – wie in Abschnitt 5.3 analysiert – die Hälfte der Probanden die Schnittstelle angepasst, obwohl dies nicht notwendig gewesen wäre.

Bei der Bearbeitung der Aufgabe 3 ist ein deutlicherer Unterschied zwischen den beiden Implementierungen sichtbar geworden. In der ADT Implementierung haben fast alle Probanden alle inhaltlichen Änderungen am Board durchgeführt, sodass der Controller nur noch die eigentliche Ausgabe in der Methode `print` durchführen musste. Lediglich ein Proband musste zusätzliche Änderungen am Controller durchführen, wie in Abschnitt 5.4 ausgeführt wurde. In der Classic Implementierung haben vier der Probanden versucht, die Änderung im Board durchzuführen oder hätten dies bevorzugt. Aufgrund der geringen Modularität haben sie sich jedoch dagegen entschieden.

Insgesamt kann damit die Modularität der ADT Implementierung als deutlich besser eingestuft werden.

6.4 Analysierbarkeit

Die Analysierbarkeit umfasst drei verschiedene Aspekte: Das Analysieren der Auswirkungen von Modifikationen, das Finden von Fehlern sowie das Identifizieren der zu modifizierenden Komponenten für eine Änderung.[108] Dabei wird im Folgenden nur der letzte der drei Aspekte weiter ausgeführt, da die anderen beiden Aspekte nicht von den Aufgaben abgedeckt werden.

In der ersten Aufgabe hat sowohl die Protokollanalyse als auch die Auswertung der Bewertungen durch die Probanden ergeben, dass die ADT Implementierung als einfacher zu lesen und intuitiver eingestuft wurde. Bei den beiden Aufgaben zur Modifikation des Quelltextes

[106] ISO Central Secretary, *ISO/IEC 25010:2011*, S. 14.
[107] Liskov, Guttag u. a., *Abstraction and specification in program development*, S. 76.
[108] ISO Central Secretary, *ISO/IEC 25010:2011*, S. 15.

unterscheiden sich die Ergebnisse. Bei der kleinen Änderung von Aufgabe 2, haben sowohl die Protokollanalyse als auch die Bewertung durch die Probanden ergeben, dass die Classic Implementierung tendenziell für die Probanden einfacher zu analysieren war. Bei Aufgabe 3 mit der größeren Änderung waren die Ergebnisse dem entgegengesetzt. Besonders deutlich hat dies die Protokollanalyse ergeben. Die Hälfte der Probanden musste bei der Classic Implementierung einen Großteil der eigenen Anpassungen verwerfen, da festgestellt wurde, dass die zunächst identifizierte Quelltextstelle ungeeignet war.

Die Ergebnisse bestätigen sich auch bei den Auswertungen der Abschlussaufgabe. Die Classic Implementierung wurde von den Probanden bevorzugt, wenn es sich um kleine Projekte oder kleine Änderungen handelt. Dabei scheint die Schwelle, ab der die Vorteile einer ADT Implementierung sichtbar werden, relativ niedrig zu sein. Auch die größere Änderung von Aufgabe 3 war in kurzer Zeit und mit wenigen Zeilen Quelltext in beiden Implementierungen umsetzbar. Dennoch war die Analysierbarkeit der Classic Implementierung bereits signifikant schlechter als die der ADT Implementierung.

6.5 Modifizierbarkeit

Die Modifizierbarkeit betrachtet, wie effektiv und effizient das Softwaresystem verändert werden kann, ohne Fehler zu erzeugen oder die Qualität anderweitig zu beeinträchtigen.[109] Dabei wird hier nicht die Effizienz betrachtet, da darauf explizit bei der Wahl der Methodik wie in Abschnitt 4.2.1 begründet verzichtet wurde.

Die funktionale Korrektheit der Modifikationen wurde bereits in Abschnitt 6.2 untersucht. Diese hat ergeben, dass bei beiden Implementierungen Fehler eingebaut wurden. Keine der beiden Implementierungen war also in der Lage, eine Modifizierbarkeit ohne Qualitätsverlust zu bieten.

Ähnlich wie bei den Ergebnissen der Analysierbarkeit in Abschnitt 6.4, ist die Effektivität der Modifizierbarkeit der ADT Implementierung erst bei der größeren Änderung besser als die der Classic Implementierung. Dabei war die größte Hürde der ADT Implementierung die Verwendung der als unnötig oder störend beschriebenen Interfaces, die zusätzlich zu den Klassen definiert sind. Dies ist eine Anmerkung, die in jeder Aufgabe von verschiedenen Probanden gemacht wurde und sich auch im Vorgehen gezeigt hat. So wurde teilweise zunächst vergessen das Interface anzupassen. Durch die zusätzliche Schnittstellendefinition waren Änderungen aufwändiger. Hier kann als Alternative die in Abschnitt 2.2.1 vorgestellte Methode verwendet werden, abstrakte Datentypen nur als Klasse ohne Interfacedefinition abzubilden. Dabei sollte zunächst jedoch untersucht werden, ob diese Methode andere Nachteile mit sich bringt.

[109] ISO Central Secretary, *ISO/IEC 25010:2011*, S. 15.

6.6 Statische Codeanalyse

Zuletzt wurde eine statische Codeanalyse wie in Abschnitt 3.1 beschrieben durchgeführt. Für die statische Codeanalyse wurde das Intellij IDEA Plugin der OpenSource Software *Sonarlint*[110] mit der Version `4.13.0.24781` eingesetzt. Zunächst wurde die statische Codeanalyse für beide Implementierungen des Basisprojektes durchgeführt, um einen Vergleichswert zu erhalten. Anschließend wurde die Codeanalyse für alle durch die Probanden veränderten Versionen durchgeführt.

In den Basisimplementierungen hat Sonarlint jeweils drei Warnungen generiert. Alle Warnungen beziehen sich auf die Nutzung der Methode `System.out.println()`. Es wurde vorgeschlagen, stattdessen einen *Logger* zu verwenden. Dies ist in richtigen Projekten eine sinnvolle Entscheidung.[111] In diesem Projekt wurde darauf jedoch bewusst verzichtet, da der Fokus des Basisprojektes die Datentypen waren. Weitere Warnungen oder Fehler wurden von Sonarlint nicht gemeldet.

In den veränderten Projekten sind in allen Implementierungen neue Warnungen bezüglich des empfohlenen Loggers dazugekommen, da die Probanden neue Ausgaben hinzufügen sollten. Diese weiteren Warnungen können positiv betrachtet werden, da die Probanden die Projekte einheitlich gehalten haben.

Neben diesen Warnungen hat Sonarlint in der Classic Implementierung bei drei Probanden die Warnung `Unused Import` erzeugt[112]. In diesen drei Implementierungen wurde an jeweils einer Stelle ein Import von `HasMap` oder `Map` gefunden, welcher nicht mehr verwendet wurde. Dies ist nur eine Warnung mit geringer Wichtigkeit, da die Codequalität dadurch nur minimal verschlechtert wird und auch automatisiert von IDEs wie *IntelliJ IDEA* behoben werden kann.[113] Dennoch kann dies ein Symptom für andere Problemen sein. In diesem Fall sind alle unbenutzten Imports entstanden, weil die Probanden sich während der Entwicklung umentscheiden mussten, wo die neue Datenstruktur gespeichert werden soll. Dies wurde bereits in Abschnitt 6.4 analysiert. Es ist auch nachvollziehbar, warum diese Warnungen nur in der Classic Implementierung aufgetreten sind. Bei der Modifikation der ADT Implementierung hat sich keiner der Probanden umentschieden, sodass keine Warnungen aufgetreten sind. Darüber hinaus wurden von Sonarlint keine weiteren Warnungen oder Fehler gefunden.

[110] SonarSource S.A. *Optimize imports*. 2021. URL: https://www.sonarlint.org (besucht am 31. 01. 2021).

[111] C. Zhi u. a. „An Exploratory Study of Logging Configuration Practice in Java“. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, S. 459–469. DOI: 10.1109/ICSME.2019.00079.

[112] Proband 2, 3 und 4

[113] JetBrains s.r.o. *Optimize imports*. 18. Jan. 2021. URL: https://www.jetbrains.com/help/idea/creating-and-optimizing-imports.html#optimize-imports (besucht am 31. 01. 2021).

6.7 Gesamtbewertung

In diesem Abschnitt wird aus den Bewertungen der einzelnen Codequalitätskriterien und der statischen Codeanalyse eine Gesamtbewertung abgeleitet. In Tabelle 3 sind dazu die Ergebnisse aus den vorangegangenen sechs Abschnitten der Einzelbewertungen kurz zusammengefasst.

Qualitätskriterium	Bewertung
Funktionale Vollständigkeit	Keine Aussage
Funktionale Korrektheit	Ausgeglichen
Modularität	ADT Implementierung ist besser
Analysierbarkeit	Bei größeren Projekten ist ADT signifikant besser
Modifizierbarkeit	Bei größeren Projekten ist ADT besser
Statische Codeanalyse	ADT Implementierung ist minimal besser

Tabelle 3: Zusammenfassung der Einzelbewertungen

Aus der Tabelle ergibt sich, dass die ADT Implementierung in allen Einzelpunkten entweder besser ist oder beide Implementierungen vergleichbar sind. Die Vorteile der ADT Implementierung sind dabei unterschiedlich stark ausgeprägt. Am deutlichsten sind diese bei der Analysierbarkeit. Im Weiteren sind Verbesserungen in der Modifizierbarkeit und Modularität sichtbar geworden. Dies sind genau die drei Teilkriterien der Wartbarkeit, die untersucht wurden. Die anderen beiden Kriterien sind Teil der Funktionalitätscharakteristik.[114] Es konnten daher keine wesentlichen Verbesserungen in der Funktionalität festgestellt werden.

Darüber hinaus hat die Classic Implementierung insbesondere bei der Analysierbarkeit und der Modifizierbarkeit bei kleinen Projekten einige Vorteile. Die Schwelle, ab der die ADT Implementierung besser geeignet ist, scheint sehr niedrig zu sein, wie in Abschnitt 6.4 ausgeführt wurde. Zu beachten ist, dass die ADT Implementierung insbesondere bei der funktionalen Korrektheit Probleme aufgezeigt. Aus der Analyse wird nicht ersichtlich, welche Folgen die gefundenen semantischen Fehler haben.

In den Interviews wurden die Probanden nach ihren Vorkenntnissen zu Conways Spiel des Lebens befragt. Während der Auswertung konnte trotz der unterschiedlichen Vorkenntnisse kein Zusammenhang zu den Ergebnissen festgestellt werden.

6.8 Reflexion der Methodik

Insgesamt ist die gewählte Methodik zur Beantwortung der Kernfrage geeignet. Im Verlauf der Arbeit wurden die meisten Teilziele erreicht und ein sinnvolles Ergebnis erarbeitet. Das Nutzen von Experteninterviews hat sich als ein zweckvolles Vorgehen herausgestellt. Auf diese Weise war eine explorative Herangehensweise möglich.

[114] ISO Central Secretary, *ISO/IEC 25010:2011*.

Als Grundlage für die Interviews wurde das Basisprojekt verwendet. Dieses hat es ermöglicht, sehr detaillierte Daten zu erheben. Damit sind jedoch auch Einschränkungen verbunden. Zunächst handelt es sich um einen künstlichen Kontext, der nicht zwingend die Realität abbilden muss. Dafür konnte die notwendige, kontrollierte Umgebung geschaffen werden. Außerdem wurde der Umfang des Projektes gering gewählt, um eine sinnvolle Auswertung zu ermöglichen. Diese Einschränkungen führen dazu, dass sich nicht alle Teilfragen beantworten lassen. Insbesondere sind dies die Auswirkungen auf die funktionale Vollständigkeit sowie einige Teilaspekte der Analysierbarkeit. Die Entscheidung, in der ADT Implementierung Interfaces zu verwenden, hat sich als negativ herausgestellt. Für das ursprüngliche Ziel, stärker differenzierte Unterschiede zu erreichen, war es kaum hilfreich. Die Interfaces stellten eine Hürde für einige Probanden dar und waren nicht notwendig, um abstrakte Datentypen zu untersuchen. Bei der Classic Implementierung war die Voraussetzung, keine abstrakten Datentypen zu verwenden. Dafür gibt es neben dem Verwendeten viele andere Ansätze. Zum Beispiel hätte die Klasse Board weggelassen und die Funktionalität mit in einen großen Controller aufgenommen werden können. Das bedeutet, dass in dieser Arbeit grundsätzlich nur ein Vergleich der beiden verwendeten Implementierungen möglich ist.

Die Interviews wurden nach der Thinking Aloud Methode durchgeführt. Dies hat es ermöglicht, umfangreiche Daten zu erhalten und hat das explorative Vorgehen unterstützt, da so neue Aspekte von den Probanden genannt wurden. Die so erhaltenen Daten konnten eine breitere Datenbasis liefern als die Effizienzermittlung durch Zeitmessung, die ohne Thinking Aloud möglich gewesen wäre.

Abschließend haben die verschiedenen Bewertungsskalen dazu beigetragen, dass weitere Daten gesammelt werden konnten. Zum einen sind viele der Aussagen der Probanden erst während der Bewertung getroffen worden. Zum anderen konnten mit den Bewertungsskalen die Ergebnisse der Protokollanalyse häufig untermauert oder ergänzt werden.

7 Zusammenfassung

In dieser Arbeit wurde betrachtet, welchen Einfluss das Nutzen abstrakter Datentypen auf die Codequalität hat. Dafür wurden zunächst unter Einbindung der existierenden Literatur die Kernaspekte abstrakte Datentypen und Codequalität umfangreich definiert und abgegrenzt. Basierend auf diesen theoretischen Erkenntnissen wurde ein Experteninterview entworfen, welches belastbare Daten zur Beantwortung der Kernfrage geliefert hat. Die Ergebnisse dieser Interviews wurden anschließend ausgewertet. Basierend auf diesen Auswertungen konnten die Auswirkungen auf die Codequalität durch den Einsatz abstrakter Datentypen bewertet werden.

Die Auswertung hat ergeben, dass die Nutzung abstrakter Datentypen die Codequalität vermutlich verbessern kann. Bei besonders kleinen Projekten oder kleinen Änderungen sind die Vorteile abstrakter Datentypen noch nicht erkennbar. Dabei war die Codequalität tendenziell schlechter als bei der klassischen Vergleichsimplementierung. Die notwendige Projektgröße, ab der die Vorteile überwiegen, ist jedoch sehr klein. So konnte bereits bei einem Projekt mit vier Klassen eine Verbesserung der Codequalität festgestellt werden. Dabei sind die Verbesserungen vor allem in der Analysierbarkeit, aber auch in der Modifizierbarkeit und Modularität aufgetreten. Damit wurden insgesamt Verbesserungen in der Wartbarkeit, nicht jedoch in der Funktionalität festgestellt.

Bei der Interpretation und Anwendung dieser Ergebnisse bleibt zu beachten, dass das Nutzen abstrakter Datentypen nur ein Teil der Qualitätssicherstellung sein kann. Zum einen wurde in dieser Arbeit nur ein Teil der von ISO 25010 definierten Kriterien zur Software- und Codequalität untersucht. Auswirkungen auf andere Teilcharakteristiken wurden nicht untersucht. Zum anderen haben auch andere Maßnahmen Einfluss auf die in dieser Arbeit untersuchten Kriterien, die mit angewendet werden müssen.

In dieser Arbeit konnte das Thema noch nicht ausschöpfend untersucht werden, sodass Teilaspekte offengeblieben sind. Diese bieten eine Möglichkeit, in der Zukunft anzusetzen. Ein Schritt kann es sein, die Analyse auf weitere Kriterien der Codequalität wie zum Beispiel die Testbarkeit auszuweiten. Anschließend kann es auch sinnvoll sein, weitere Teilaspekte der Softwarequalität neben der Codequalität einzubeziehen, um ein klareres Gesamtbild zu erhalten. Ein weiterer in dieser Arbeit nicht betrachteter Bereich sind die Langzeitauswirkungen der Nutzung abstrakter Datentypen. Durch die angewendete Methodik war es möglich, den Einfluss auf die aktuelle Codequalität zu untersuchen. Es können jedoch keine Aussagen darüber gemacht werden, wie sich dies auf die Qualität in dem gesamten Lebenszyklus der Software auswirken kann. Insbesondere wurde in Abschnitt 4.2.2 bereits festgestellt, dass keine Aussagen über bekannten Quelltext gemacht werden können. Außerdem konnten die in Abschnitt 6.2 aufgetretenen semantischen Fehler nicht auf deren Auswirkungen untersucht werden, da das Projekt nicht weiterentwickelt wird.

Quellenverzeichnis

Arnold, Ken, James Gosling und David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

Azmoodeh, Manoochchr. *Abstract data types and algorithms*. Palgrave Macmillan UK, 1990. DOI: 10.1007/978-1-349-21151-7.

Brauer, Johannes. „Typen, Objekte, Klassen: Teil 1 – Grundlagen“. de. In: *Arbeitspapiere der Nordakademie* (Jan. 2009).

— „Typen, Objekte, Klassen: Teil 2 – Sichtweisen auf Typen“. de. In: *Arbeitspapiere der Nordakademie* (Jan. 2010).

Broy, Manfred, Florian Deißenböck und Markus Pizka. „A holistic approach to software quality at work“. In: *Proc. 3rd world congress for software quality (3WCSQ)*. 2005.

Chantree, F. u. a. „Identifying Nocuous Ambiguities in Natural Language Requirements“. In: *14th IEEE International Requirements Engineering Conference (RE'06)*. 2006, S. 59–68. DOI: 10.1109/RE.2006.31.

El Emam, Khaled. *The ROI from software quality*. CRC press, Juni 2005. DOI: 10.1201/9781420031201.

Estdale, John und Elli Georgiadou. „Applying the ISO/IEC 25010 Quality Models to Software Product“. In: *Systems, Software and Services Process Improvement*. Hrsg. von Xabier Larrucea u. a. Cham: Springer International Publishing, 2018, S. 492–503. ISBN: 978-3-319-97925-0.

Gardner, Martin. „The fantastic combinations of John Conway's new solitaire game "life"“. In: *Scientific American* 223 (1970), S. 120–123.

Google Scholar. *Google Scholar search for „abstract data types“*. 1. Feb. 2021. URL: https://scholar.google.com/scholar?q=%22abstract+data+types%22 (besucht am 01. 02. 2021).

— *Google Scholar search for „Programming with abstract data types“*. 1. Feb. 2021. URL: https://scholar.google.com/scholar?q=Programming+with+abstract+data+types (besucht am 01. 02. 2021).

Guttag, John. „Abstract Data Types and the Development of Data Structures“. In: *Commun. ACM* 20.6 (Juni 1977), S. 396–404. ISSN: 0001-0782. DOI: 10.1145/359605.359618.

Guttag, John V. und James J. Horning. „The algebraic specification of abstract data types“. In: *Acta informatica* 10.1 (1978), S. 27–52.

Hand, David J. „Statistics and the theory of measurement“. In: *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 159.3 (1996), S. 445–473.

Hofmann, Martin und Martin Lange. *Automatentheorie und Logik*. Springer-Verlag, 2011.

ISO Central Secretary. *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*. en. Standard ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization, 2005.

ISO Central Secretary. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. en. Standard ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization, 2011.

JetBrains s.r.o. *Optimize imports*. 18. Jan. 2021. URL: https://www.jetbrains.com/help/idea/creating-and-optimizing-imports.html#optimize-imports (besucht am 31. 01. 2021).

Jose, Manu und Rupak Majumdar. „Cause Clue Clauses: Error Localization Using Maximum Satisfiability“. In: *SIGPLAN Not.* 46.6 (Juni 2011), S. 437–446. ISSN: 0362-1340. DOI: 10.1145/1993316.1993550.

Kart, Michael. „Teaching Type Design Using Transition Diagrams and Sports Scoreboards“. en. In: *J. Comput. Sci. Coll.* 31.4 (Apr. 2016), S. 28–35. ISSN: 1937-4771.

Kiczales, Gregor. „Towards a new model of abstraction in software engineering“. In: *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*. IEEE. 1991, S. 127–128. DOI: 10.1109/IWOOOS.1991.183036.

Krosnick, Jon A, Allyson L Holbrook und Penny S Visser. „Optimizing brief assessments in research on the psychology of aging: A pragmatic approach to self-report measurement“. In: *When I'm* 64 (2006), S. 231.

Lewis, Clayton. *Using the"thinking-aloud"method in cognitive interface design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

Likert, Rensis. „A technique for the measurement of attitudes.“ In: *Archives of psychology* (1932).

Lindner, Dominic. „Forschungsdesigns der Wirtschaftsinformatik“. In: *Forschungsdesigns der Wirtschaftsinformatik*. Springer, 2020, S. 19–44.

Lions, Jacques-Louis u. a. „Flight 501 failure“. In: *Report by the Inquiry Board* 190 (1996).

Liskov, Barbara, John Guttag u. a. *Abstraction and specification in program development*. Bd. 180. MIT press Cambridge, 1986.

Liskov, Barbara und Stephen Zilles. „Programming with Abstract Data Types“. en. In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Santa Monica, California, USA: Association for Computing Machinery, 1974, S. 50–59. ISBN: 9781450378840. DOI: 10.1145/800233.807045.

Lozano, Luis M, Eduardo García-Cueto und José Muñiz. „Effect of the number of response categories on the reliability and validity of rating scales“. In: *Methodology* 4.2 (2008), S. 73–79.

Oracle. *What's New in JDK 8*. 19. Apr. 2018. URL: https://www.oracle.com/java/technologies/javase/8-whats-new.html (besucht am 31. 01. 2021).

Pan, Jiantao. „Software testing“. In: *Dependable Embedded Systems* 5 (1999), S. 2006.

Parnas, D. L., John E. Shore und David Weiss. „Abstract Types Defined as Classes of Variables“. In: *SIGPLAN Not.* 11.SI (März 1976), S. 149–154. ISSN: 0362-1340. DOI: 10.1145/942574.807133.

Plösch, R. u. a. „A Method for Continuous Code Quality Management Using Static Analysis“. In: *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE. 2010, S. 370–375. DOI: 10.1109/QUATIC.2010.68.

Racordon, Dimitri und Didier Buchs. „Featherweight swift: A core calculus for swift's type system“. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, S. 140–154.

Ray, Baishakhi u. a. „Detecting and characterizing semantic inconsistencies in ported code“. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, S. 367–377.

Sivilotti, Paolo A.G. und Matthew Lang. „Interfaces First (and Foremost) with Java“. en. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE '10. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 2010, S. 515–519. ISBN: 9781450300063. DOI: 10.1145/1734263.1734436.

SonarSource S.A. *Optimize imports*. 2021. URL: https://www.sonarlint.org (besucht am 31. 01. 2021).

Stack Overflow Ltd. *Stack Overflow Annual Developer Survey*. Feb. 2020. URL: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages (besucht am 20. 12. 2020).

Tuteja, Maneela, Gaurav Dubey u. a. „A research study on importance of testing and quality assurance in software development life cycle (SDLC) models“. In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.3 (2012), S. 251–257.

Zhi, C. u. a. „An Exploratory Study of Logging Configuration Practice in Java“. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, S. 459–469. DOI: 10.1109/ICSME.2019.00079.

A Anhang

A.A Digitaler Anhang

Der digitale Anhang beinhaltet alle Teile des Anhangs, die sich nicht beziehungsweise nur sehr schlecht in diese Arbeit einfügen lassen und werden deshalb nur digital zur Verfügung gestellt.

A.A.A Basisprojekt

Das Basisprojekt, wie es die Probanden zu Beginn jedes Interviews erhalten haben, befindet sich im digitalen Anhang im Ordner *Basisprojekt* und besteht aus einem Java Projekt.

A.A.B Projekte nach den Änderungen

Neben dem Basisprojekt befinden sich alle veränderten Versionen des Basisprojektes von den Probanden nach den Interviews im digitalen Anhang. Die veränderten Versionen befinden sich im Ordner *Aenderung/Proband X*

A.B Klassendiagramme

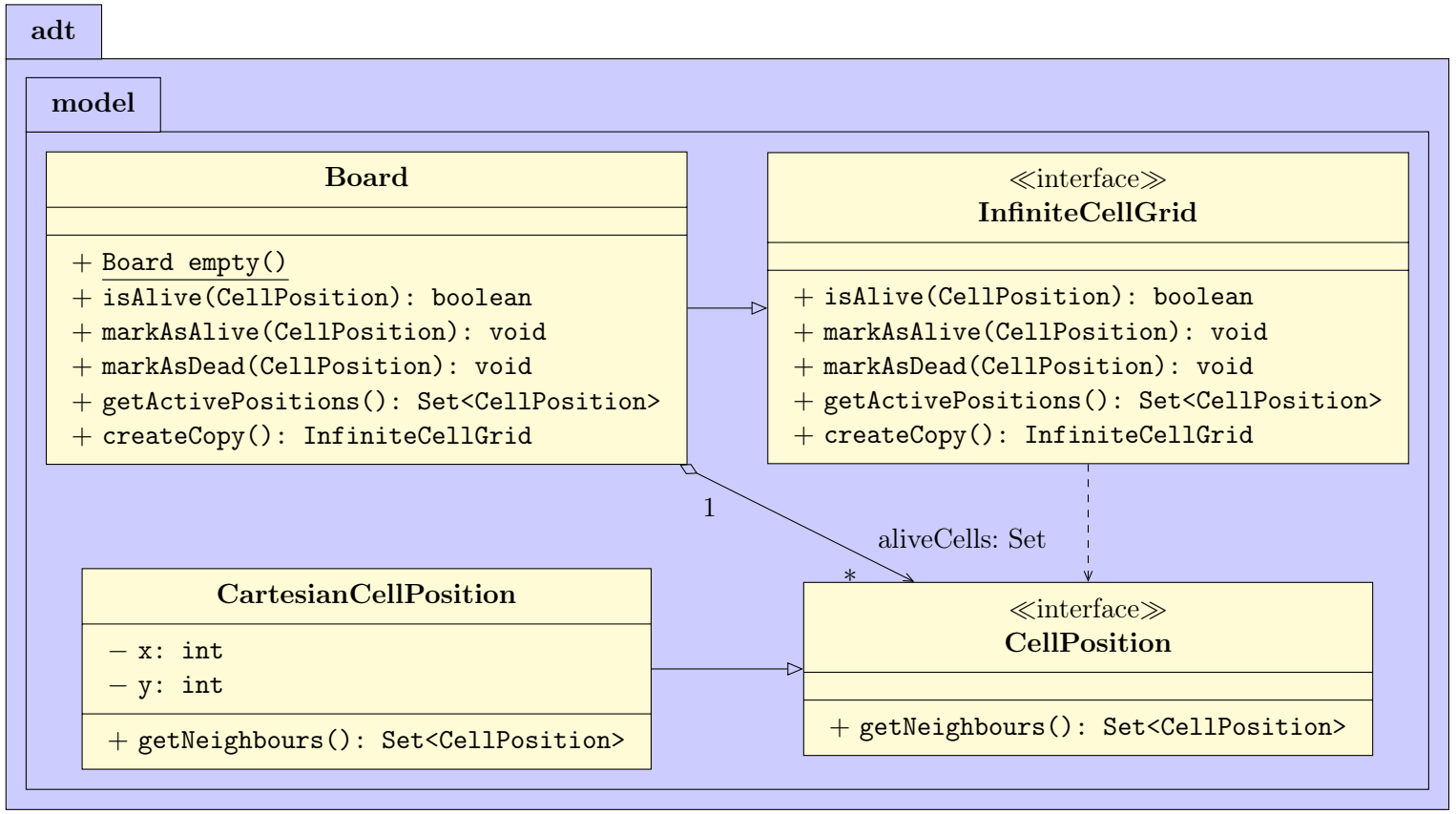

Abbildung 8: UML Klassen Diagramm der ADT Implementierung *(Quelle: Eigene Darstellung)*

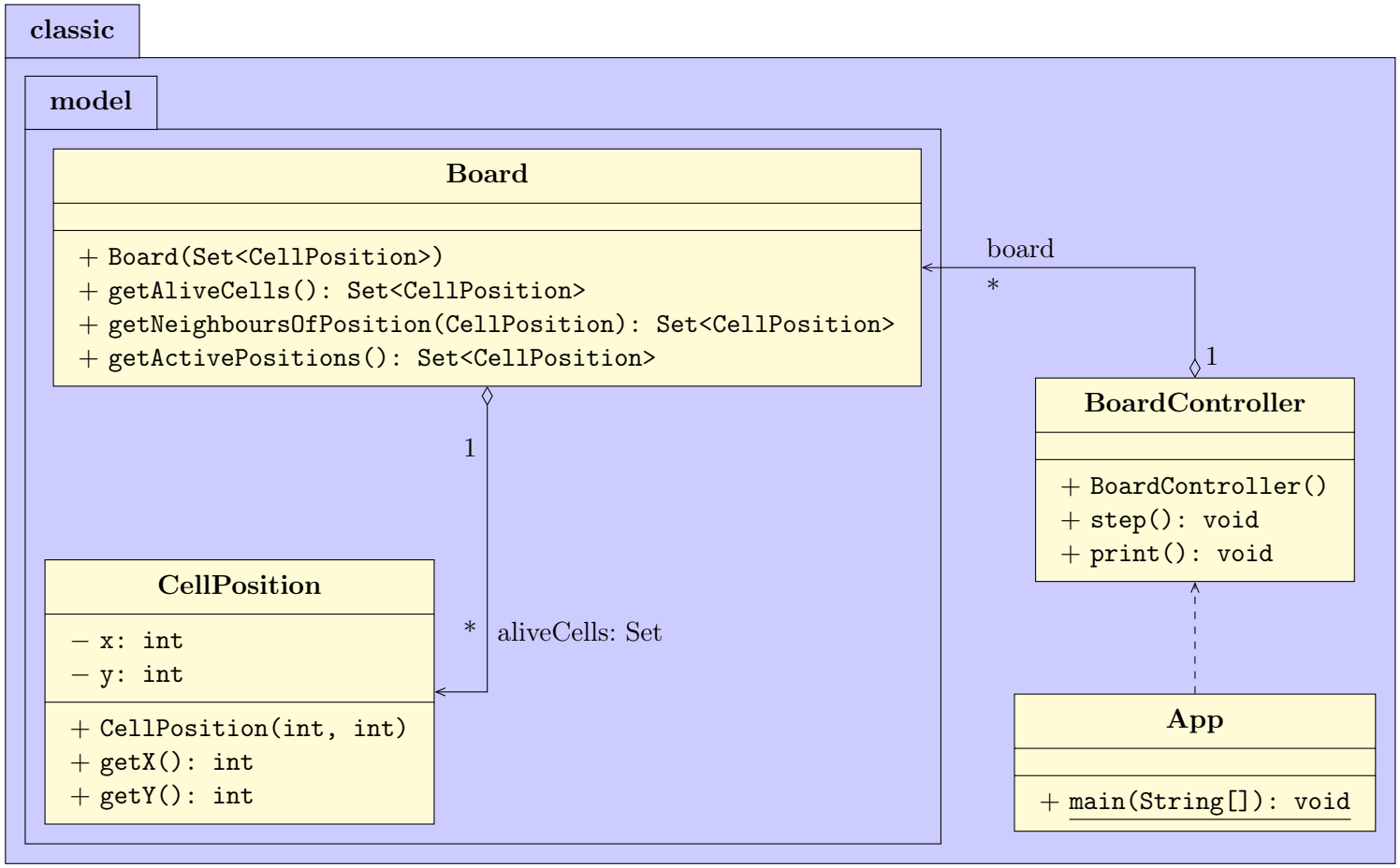

Abbildung 9: UML Klassen Diagramm der Classic Implementierung *(Quelle: Eigene Darstellung)*

A.C Aufgaben für die Experteninterviews

1 Vorbereitung

An mehreren Stellen sollst du kurz auf einer Skala von 1 - 5 verschiedene Aspekte bewerten. Dabei soll 1 für *ich stimme gar nicht zu* und 5 für *ich stimme voll zu* stehen.

Um an den Aufgaben teilzunehmen, solltest du mindestens die Nutzung von `Klassen` und `Interfaces` der Programmiersprache `Java` kennen. Bewerte bitte kurz deine Erfahrung mit `Java`:

Kenntnisse	1	2	3	4	5
Java					

Lade dir das Basisprojekt runter:
`git clone git@github.com:NoahPeeters/ba-demo.git`
oder
`git clone https://github.com/NoahPeeters/ba-demo.git`

Das Projekt enthält zwei Implementierungen von *Conways Spiel des Lebens*. Sieh dir diese noch nicht an! Bitte bewerte zunächst kurz, wie gut du die Simulation kennst.

Kenntnisse	1	2	3	4	5
Conways Spiel des Lebens					

Die beiden Implementierungen liegen in den Paketen `de.noahpeeters.gameoflife.classic` und `de.noahpeeters.gameoflife.adt`. Die folgenden drei Aufgaben sollst du jeweils für beide Implementierungen durchführen. Achte darauf, dass du jeweils mit der selben Implementierung beginnst. Es kann sich lohnen nach dem Bearbeiten einer Aufgabe alle Dateien zu schließen, um nicht mit den beiden Implementierung durcheinander zu kommen. Erkläre während dem Bearbeiten, was du gerade machst, und versuche auch Probleme die du hast zu benennen.

2 Erklärung der Implementierung

Sieh dir den Quelltext an, und erkläre, wie dieser funktioniert und aufgebaut ist. Es geht nur um die Dateien, die Teil des entsprechenden *Java-Paketes* sind. Das Projekt ist nicht sehr umfangreich, versuche also alle Aspekte des Quelltextes durchzusehen. Du solltest dich danach in der Lage fühlen, Änderungen an dem Quelltext durchzuführen.

Führe das Programm abschließend einmal aus.

Classic	1	2	3	4	5
Der Code ist einfach zu lesen					
Der Code ist intuitiv					
Ich habe das Gefühl den Code einfach ändern zu können					

ADT	1	2	3	4	5
Der Code ist einfach zu lesen					
Der Code ist intuitiv					
Ich habe das Gefühl den Code einfach ändern zu können					

Ließ erst weiter, wenn du bis hier alles bearbeitet hast.

3 Kleine Änderung

Nun soll ein neue Funktion mit in das Programm eingebaut werden. Bei der Ausgabe soll eingebaut werden, dass angegeben wird, wie viele Zellen aktuell lebendig sind. Nimm dafür beliebige Änderungen vor, die deiner Meinung nach am Besten geeignet sind. Achte aber darauf, dass du keine Klassen/Interfaces aus der jeweils anderen Implementierung benutzen darfst.

Classic	1	2	3	4	5
Die Änderung war einfach durchzuführen					
Ich wusste, wo ich etwas ändern muss					
Der bestehende Code hat mir geholfen, die Änderung durchzuführen					

ADT	1	2	3	4	5
Die Änderung war einfach durchzuführen					
Ich wusste, wo ich etwas ändern muss					
Der bestehende Code hat mir geholfen, die Änderung durchzuführen					

Ließ erst weiter, wenn du bis hier alles bearbeitet hast.

4 Größere Änderung

Nun soll eine zweite Funktion eingebaut werden. Das Programm soll in der Lage sein auszugeben, wie oft eine Zelle lebendig war. Wenn eine Zelle mehrere Runden hintereinander lebendig war, zählt dies als 1. Zum Beispiel zählt `lebendig -> lebendig -> tot` als 1, während `lebendig -> tot -> lebendig` als 2 zählt.

Das Programm soll am Ende die Anzahl für die Zellen bei (x = 0, y = 0) und (x = 3, y = 2) ausgeben.

Classic	1	2	3	4	5
Die Änderung war einfach durchzuführen					
Ich wusste, wo ich etwas ändern muss					
Der bestehende Code hat mir geholfen, die Änderung durchzuführen					

ADT	1	2	3	4	5
Die Änderung war einfach durchzuführen					
Ich wusste, wo ich etwas ändern muss					
Der bestehende Code hat mir geholfen, die Änderung durchzuführen					

Ließ erst weiter, wenn du bis hier alles bearbeitet hast.

5 Abschluss

Bewerte abschließend noch folgende Aussagen:

Abschluss	1	2	3	4	5
Die Unterschiede zwischen den beiden Implementierung wurde deutlich					
Ich präferiere die `Classic` Implementierung über der `ADT` Implementierung					

A.D Interviewprotokolle

A.D.A Proband 1

Proband 1 hat mit der Classic Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	4	
Conways Spiel des Lebens	5	
1 – Lesen	3	4
1 – Intuitiv	3	4
1 – Änderbar	4	4
2 – Einfach	5	3
2 – Ort	5	2
2 – Hilfe	5	3
3 – Einfach	4	4
3 – Ort	4	4
3 – Hilfe	3	4
Unterschied	5	
Präferenz	1	

Tabelle 4: Bewertung durch Proband 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Die Methode `step` ist schwierig zu lesen – ADT: Quelltext ist leserlicher – ADT: Benennung von Interface und Klasse ungünstig, daher ist die Zuordnung schwierig
Auffälligkeiten	– ADT: Bemerkt Unterschiede im `BoardController` erst spät

Tabelle 5: Protokoll von Proband 1 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Der Quelltext in der Methode `print` enthält bereits ähnlichen Code – ADT: Aufgrund des Interface muss man etwas länger alle Stellen für die Änderung suchen
Auffälligkeiten	– Classic: Änderung wurde sehr schnell durchgeführt – ADT: Versucht bestehende Interfacemethoden zu verwenden

Tabelle 6: Protokoll von Proband 1 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Die notwendigen Änderungen sind offensichtlich – Classic: Kopieren war nicht in der Semantik, daher keine Kopie von der HashMap – ADT: Die Änderungen sind intuitiv – ADT: Die Methode `createCopy` erstellt Kopie von der HashMap aufgrund der Semeantik des Namens
Auffälligkeiten	– Classic: Änderung wurde sehr schnell durchgeführt – ADT: Versucht bestehende Interfacemethoden zu verwenden – ADT: Verwendet nicht die Methode `markAsAlive`, sondern baut `cellBecameAlive`

Tabelle 7: Protokoll von Proband 1 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	– ADT: Die Interface sind unnötig und sollte man weglassen
Auffälligkeiten	/

Tabelle 8: Protokoll von Proband 1 zu Abschluss *(Quelle: Eigene Darstellung)*

A.D.B Proband 2

Proband 2 hat mit der Classic Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	4	
Conways Spiel des Lebens	5	
1 – Lesen	4	5
1 – Intuitiv	3	5
1 – Änderbar	4	5
2 – Einfach	5	4
2 – Ort	5	5
2 – Hilfe	5	2
3 – Einfach	3	5
3 – Ort	4	5
3 – Hilfe	3	5
Unterschied	5	
Präferenz	2	

Tabelle 9: Bewertung durch Proband 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Nur aufgrund des geringen Umfangs einfach zu lesen – ADT: Der Quelltext ist sehr sprechend
Auffälligkeiten	– ADT: Erkennt die Semantik von vielen Methoden am Namen

Tabelle 10: Protokoll von Proband 2 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Bestehender Getter macht die Änderung sehr einfach – ADT: Interface macht das Durchführen der Änderung deutlich aufwändiger
Auffälligkeiten	– ADT: Versucht bestehende Interfacemethoden zu verwenden

Tabelle 11: Protokoll von Proband 2 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Struktur des Quelltextes verkompliziert die Änderung
Auffälligkeiten	– Classic: Baut HashMap erst im BoardController ein und verschiebt diese anschließend ins Board – Classic: HashMap wird im Konstruktor vom Board neu erzeugt. Dies führt zum Datenverlust und das Ergebnis ist falsch.

Tabelle 12: Protokoll von Proband 2 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	– ADT: Datenhoheit wird umgesetzt – ADT: Sprechende Namen vereinfachen den Umgang mit dem Code – ADT: Interfaces erschweren Änderungen – ADT: Auf Dauer machen die Interfaces den Quelltext vermutlich leichter erweiterbar
Auffälligkeiten	/

Tabelle 13: Protokoll von Proband 2 zu Abschluss *(Quelle: Eigene Darstellung)*

A.D.C Proband 3

Proband 3 hat mit der Classic Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	5	
Conways Spiel des Lebens	1	
1 – Lesen	1	3
1 – Intuitiv	2	3
1 – Änderbar	1	4
2 – Einfach	5	5
2 – Ort	5	5
2 – Hilfe	5	5
3 – Einfach	2	3
3 – Ort	3	3
3 – Hilfe	1	4
Unterschied	5	
Präferenz	1	

Tabelle 14: Bewertung durch Proband 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Die Lesbarkeit ist sehr schlecht
Auffälligkeiten	– ADT: Erkennt die Semantik von vielen Methoden am Namen

Tabelle 15: Protokoll von Proband 3 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– ADT: Verwendet bestehende Getter, weil es einfacher ist
Auffälligkeiten	– Löst die Aufgabe in beiden Implementierungen sehr schnell ohne viel nachdenken

Tabelle 16: Protokoll von Proband 3 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Die HashMap sollte ins Board, weil es dort inhaltlich hingehört – Classic: Verschiebt HashMap in den Controller, weil es einfacher ist
Auffälligkeiten	/

Tabelle 17: Protokoll von Proband 3 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	– ADT: Geheimnisprinzip wird erfüllt
Auffälligkeiten	/

Tabelle 18: Protokoll von Proband 3 zu Abschluss *(Quelle: Eigene Darstellung)*

A.D.D Proband 4

Proband 4 hat mit der ADT Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	4	
Conways Spiel des Lebens	4	
1 – Lesen	3	3
1 – Intuitiv	3	4
1 – Änderbar	2	4
2 – Einfach	4	4
2 – Ort	4	4
2 – Hilfe	3	4
3 – Einfach	3	4
3 – Ort	3	4
3 – Hilfe	1	4
Unterschied	4	
Präferenz	2	

Tabelle 19: Bewertung durch Proband 4 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: In größeren Projekten vermutlich schlechtere Bewertung – Classic: Zugriff auf Instanzvariablen über getter ist unschön.
Auffälligkeiten	– ADT: Erkennt die Semantik von vielen Methoden am Namen

Tabelle 20: Protokoll von Proband 4 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: aliveCount getter, damit man nicht von internen Zustand abhängt
Auffälligkeiten	– ADT: Versucht bestehende Interfacemethoden zu verwenden

Tabelle 21: Protokoll von Proband 4 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Der entstehende Quelltext ist nicht schön, aber erfüllt seinen Zweck – Classic: HashMap im den Controller, weil es einfacher ist
Auffälligkeiten	/

Tabelle 22: Protokoll von Proband 4 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	/
Auffälligkeiten	/

Tabelle 23: Protokoll von Proband 4 zu Abschluss *(Quelle: Eigene Darstellung)*

A.D.E Proband 5

Proband 5 hat mit der ADT Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	4	
Conways Spiel des Lebens	4	
1 – Lesen	4	5
1 – Intuitiv	3	5
1 – Änderbar	3	4
2 – Einfach	5	5
2 – Ort	5	5
2 – Hilfe	5	5
3 – Einfach	4	4
3 – Ort	3	3
3 – Hilfe	3	4
Unterschied	5	
Präferenz	2	

Tabelle 24: Bewertung durch Proband 5 *(Quelle: Eigene Darstellung)*

Anmerkungen	– ADT: Semantische Bedeutung der statischen Methode `empty` im Board ist einfacher als bei einem Konstruktor zu erkennen
Auffälligkeiten	– Empfindet beide Implementierung als sehr ähnlich

Tabelle 25: Protokoll von Proband 5 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	/
Auffälligkeiten	– Implementiert die Änderung sehr schnell ohne Probleme in beiden Implementierungen

Tabelle 26: Protokoll von Proband 5 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Es würde Sinn ergeben eine Methode zu bauen, die `getOrDefault(zelle, 0)` aufruft. Wird aber nicht gemacht, weil es einfacher ist
Auffälligkeiten	– Anfänglich Schwierigkeiten einen geeigneten Ort zu finden, will zunächst die Zellposition zu erweitern. Das Funktioniert aber nicht mit der aktuellen Implementierung – ADT: Ist sich unsicher, wann die HashMap kopiert werden sollte

Tabelle 27: Protokoll von Proband 5 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	/
Auffälligkeiten	/

Tabelle 28: Protokoll von Proband 5 zu Abschluss *(Quelle: Eigene Darstellung)*

A.D.F Proband 6

Proband 6 hat mit der ADT Implementierung begonnen.

Frage	Antwort	
	Classic	ADT
Java	4	
Conways Spiel des Lebens	2	
1 – Lesen	3	4
1 – Intuitiv	3	3
1 – Änderbar	3	5
2 – Einfach	4	3
2 – Ort	3	3
2 – Hilfe	4	2
3 – Einfach	4	3
3 – Ort	5	4
3 – Hilfe	3	2
Unterschied	5	
Präferenz	4	

Tabelle 29: Bewertung durch Proband 6 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Der Quelltext ist aufgrund der verknüpften Aufrufe wenig sprechend – ADT: `isAlive` auf dem Board ist unintuitiv – ADT: Insbesondere bei größeren Projekten vermutlich einfacher zu ändern
Auffälligkeiten	/

Tabelle 30: Protokoll von Proband 6 zu Aufgabe 1 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Die Methode `print` ist mit dem Ausschnitt verwirrend. – ADT: Getter für alle Zellen anstatt von nur der Anzahl ist flexibler in der Zukunft
Auffälligkeiten	– Es wurde zunächst nicht erkannt, dass nur ein Teil des Boards in der Methode `print` ausgegeben wird. Daher wurde zunächst an einer falschen Stelle versucht, die Zellen mitzuzählen.

Tabelle 31: Protokoll von Proband 6 zu Aufgabe 2 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic: Weniger Quelltext ist notwendig – Classic: Keine Änderung am Board notwendig – Classic: Bläht den Quelltext auf – ADT: Im Vergleich aufwändiger – ADT: Das Interface ist unnötig
Auffälligkeiten	– ADT: Versucht es zunächst in der Methode `step` einzubauen, bemerkt dann aber Redundanz bim Initialisieren – Classic: Versucht bei Classic den selben Ansatz wie bei ADT, bemerkt dann aber Probleme mit der Initalfüllung und baut die HashMap in den Controller

Tabelle 32: Protokoll von Proband 6 zu Aufgabe 3 *(Quelle: Eigene Darstellung)*

Anmerkungen	– Classic ist besser bei kleinen, schnellen Lösungen – ADT ist besser bei großen Projekten – ADT: Schnittstellendefinition ist unnötig bei häufigen Änderungen.
Auffälligkeiten	/

Tabelle 33: Protokoll von Proband 6 zu Abschluss *(Quelle: Eigene Darstellung)*