Auswahl von Beispielen in interaktiven Code Präsentationen durch strukturelle Programmanalyse

Bachelorarbeit
Universität Bremen
Fachbereich 3: Mathematik und Informatik

Autor: Nils-Janis Mahlstädt

Studiengang: Informatik

Erstgutachter: Prof. Dr. Christoph Lüth
Zweitgutachter: Dr. Berthold Hoffmann

Betreuung: Dipl.-Inf. Martin Ring

Bremen, 11. Februar 2020

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 11. Februar 2020

Nils-Janis Mahlstädt

Zusammenfassung

Die Benutzbarkeit von Cobra, dem interaktiven Präsentationstool für Quellcode und formale Beweise, soll verbessert werden, indem eine neue Sprache zum Schreiben von Folien sowie eine weitere Möglichkeit zur Auswahl von Quellcodebeispielen erarbeitet wird. Ziel dieser Verbesserungen ist es die Arbeit von Autoren bei der Erstellung von Foliensätzen zu vereinfachen.

Unter den Gesichtspunkten der Lesbarkeit des Quelltextes, seiner Features und Verbreitung wird Markdown als Alternative zum bestehenden HTML ausgewählt. Zur Auswahl von Beispielen werden Abfragen auf vereinfachten Syntaxbäumen syntaktisch und semantisch spezifiziert sowie eine Methode zur Erstellung dieser vereinfachten Bäume aus abstrakten Syntaxbäumen vorgestellt. Zur Demonstration der Umsetzbarkeit erfolgt eine Integration in Cobra.

Eine Nutzerstudie zeigt die Anwendbarkeit der Konzepte zur Auswahl von Beispielen und dem Schreiben von Folien in Anwendungsfällen, die im Kontext der Arbeit mit Cobra auftreten können.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der Arbeit „Interactive Proof Presentations with Cobra“ wurde 2017 das Tool Cobra[1] zur interaktiven Präsentation von Beweisen und Quellcode vorgestellt [1]. Sein besonderes Feature ist, dass in die Folien eingebettete Beispiele während der Vorführung inklusive Compiler und Solver-Output verändert beziehungsweise ausgeführt werden können. Dieses Feature unterscheidet Cobra von anderen in der Lehre eingesetzten Programmen für Präsentationen wie LaTeX Beamer Folien und Microsoft PowerPoint / LibreOffice / OpenOffice / Keynote Präsentation. Es entstehen bei der Nutzung von Cobra Präsentationen Interaktionsmöglichkeiten mit den Inhalten, die in anderen Tools nicht möglich sind.

Trotz dieses Features ist Cobra nicht verbreitet. Es wird vermutet, dass unter anderem Benutzbarkeitsprobleme ein Grund dafür sein können. In Gesprächen haben sich zwei Punkte herauskristallisiert, an denen die Benutzbarkeit Potenzial für Verbesserungen bietet. Inhalte in Cobra müssen als HTML-Dokumente verfasst werden. Dieses ist ohne Hilfsfunktionen eines Editors fehleranfällig, abstrakt und umständlich zu schreiben. Außerdem sind HTML-Folien zwar klar strukturiert, aber dennoch durch die Menge an Tags umständlich zu lesen und können für Personen ohne HTML-Vorkenntnisse abschreckend wirken. Die Idee neben HTML eine einfachere Eingabesprache zu unterstützen, ist bereits bei der Vorstellung von Cobra als vorgeschlagene zukünftige Arbeit präsentiert worden [1, S. 6]. Außerdem sind die Möglichkeiten zur Auswahl der Beispiele in Folien entweder in der Form von Datei- und Zeilenangaben rudimentär oder erfordern Annotationen und Marker in den Quelldateien, also ausschließlich für die Präsentation benötigte Anpassungen im Quellcode. Motiviert durch diese Punkte ist die folgende Problemstellung entwickelt worden.

Problemstellung: Im Rahmen dieser Arbeit sollen Konzepte entwickelt werden, die Präsentationsautoren das Arbeiten mit dem Tool Cobra vereinfachen. Zum einen soll eine Eingabesprache für Folien entwickelt und integriert werden, die im Vergleich zur derzeit existierenden Sprache HTML einfacher zu schreiben ist. Zum anderen soll eine bessere Methodik erarbeitet werden, um Quellcodebeispiele in Präsentationen einzubinden. Diese neue Methode soll intuitiv nutzbar,

[1] https://github.com/flatmap/cobra

verständlich und robust gegenüber Dateiänderungen sein. Im Vergleich zur aktuellen Methode soll sie ohne spezielle Markierungen in den Quellcodedateien arbeiten (vgl. 2.1.2). Neben der theoretischen Erarbeitung soll ebenfalls eine Integration in Cobra erfolgen, um die praktische Umsetzbarkeit zu demonstrieren.

Struktur dieser Arbeit: In den Grundlagen (Kapitel 2) wird ein Überblick über die dieser Arbeit zugrunde liegenden Technologien und Projekte gegeben. Der Entwurf (Kapitel 3) beschreibt die in dieser Arbeit erarbeiteten Konzepte und Entwurfsentscheidungen. In der Integration (Kapitel 4) werden einzelne Teile der Umsetzung vorgestellt und abschließend wird in der Evaluation (Kapitel 5) überprüft, inwiefern sich die im Entwurf beschriebenen Konzepte zur Lösung der Problemstellung eignen.

Kapitel 2

Grundlagen

Im Folgenden wird Cobra vorgestellt, da die Fragestellungen und Motivation dieser Arbeit Bezug zu Cobra haben. Es wird ein Überblick über die verwendeten, bereits bestehenden Projekte Markdown, pandoc und das Language Server Protocol gegeben.

2.1 Cobra

Cobra ist ein Tool zur Präsentation von formalen Beweisen und Quellcode im Kontext der Lehre. Ein Schwerpunkt des Tools liegt auf dem Erstellen von interaktiven Präsentationen, welche, während sie präsentiert werden, durch den Vortragenden sowie Zuhörer veränderbar sind, um eine lebendige Interaktion mit dem Publikum zu ermöglichen. Beweise können nahtlos in den Folien angepasst und verändert werden. „Was würde passieren, wenn“-Szenarien können so interaktiv erforscht werden.

Die Präsentationsfolien werden in HTML verfasst und mit HTML-Klassen für das Präsentationss-Framework markiert. Die Funktionalität klickbare Präsentationen im Browser darzustellen, wird durch die JavaScript Bibliothek revealJS bereitgestellt (siehe Abschnitt 2.1.1). Cobra tauscht zur Ausführungszeit Teile des HTML-Baumes durch angepasste Objekte aus. Codebeispiele werden kontinuierlich mit anderen Clients und dem Server synchronisiert, analysiert und an spezialisiertere Tools, wie Präsentationscompiler, weitergeleitet.

Architektonisch basiert Cobra auf einer Client-Server-Architektur. Der Client ist in Scala geschrieben und wird zu JavaScript kompiliert. Der Server ist ebenfalls in Scala beschrieben und wird auf der JVM ausgeführt. Die Kommunikation zwischen beiden Komponenten findet über einen Websocket statt [2]. Der Server ist auf Basis des Akka Frameworks aufgebaut und verwendet für die HTTP- und Websocket-Schnittstellen Akka HTTP und für die Implementierung vieler Komponenten Akka Actors. Es werden nicht typisierte Akka Classic Actoren verwendet. Weitere Informationen zu Akka Actors sind im Abschnitt 2.1.3 zu finden.

Fall nicht anders angegeben, stammen die Informationen aus den vorherigen Absätzen aus der Vorstellung von Cobra [1] und dem Cobra Quellcode sowie Handbuch [3].

2.1.1 Folien schreiben mit reveal.js

Reveal.js ist ein JavaScript Framework, das seit 2011 open source entwickelt wird [4]. Die HTML-Struktur der Folien ist durch vom Framework vorgegebene Regeln geprägt. Alle Slides müssen unter `.reveal > .slides` angeordnet sein. Einzelne Slides müssen durch `section` Tags umschlossen sein und können bis zu einer Tiefe von zwei geschachtelt werden. Aus dieser Schachtelung ergibt sich beim Betrachten der Folien im Browser ein 2D Gitter aus Slides [5, Kapitel Markup]. Alle Slides auf Ebene eins werden, wie in anderen Präsentationsprogrammen, während ihrer Vorstellung in einer horizontalen Bahn angeordnet. Alle Slides, die auf Ebene zwei in einer anderen Section geschachtelt sind, werden vertikal unter dieser Section angeordnet. Das gesamte Slidedeck kann dann während der Präsentation vertikal und horizontal durchlaufen werden.

Viele Aspekte der Präsentation können über eine JavaScript Schnittstelle konfiguriert werden. Hintergrundbilder, Effekte, Animationen, Auslöser für automatisierte Abläufe und Transitionen werden durch eine Mischung aus JavaScript Code, reveal.js-spezifischen HTML-Attributen und CSS-Klassen ausgedrückt. Das allgemeine Styling einer Präsentation wird durch CSS vorgenommen. Im Kontext von Cobra wird eine Standard-CSS-Datei mit ausgeliefert, so dass dieser Schritt optional ist.

2.1.2 Auswahl von Beispielen

Als Werkzeug zum Präsentieren von Programmcode und Beweisen ist es ein zentraler Bestandteil von Cobra, Beispiele in Folien einzubinden. Dazu gibt es zwei Möglichkeiten: die Angabe von Dateipfaden optional mit Angabe von Start- und Endzeilen des Beispiels sowie magische Kommentare in den Quelldateien.

Beide verwendeten Ausdrücke werden im `src` Attribut eines Quellcodeblocks angegeben. Listing 2.1 zeigt zwei Folien, welche jeweils eine der beiden Möglichkeiten verwenden.

```
1  <section>
2          <h1> Beispiel Pfade </h1>
3          <code
4                  src="srcs/beispiel/myclass.scala"
5                  from="0"
6                  to="10"
7                  class="scala"
8          > </code>
9  </section>
10
11 <section>
12         <h1> Beispiel magische Kommentare </h1>
```

```
13          <code
14                  src="srcs/beispiel/comments.scala"
15                  class="hidden"
16          > </code>
17
18          <code
19                  src="#myFunc"
20                  class="scala"
21          > </code>
22  </section>
```

Listing 2.1: Dateipfade und magische Kommentare in Cobra Folien

In Zeilen 3 bis 8 wird ein Beispiel über einen Dateipfad (`src` Attribut) angegeben. Die Attribute `from` und `to` geben die Anfangs- und Endzeilen an und sind optional. Im `class` Attribut wird die Klasse `scala` definiert. Sie wird für das Syntaxhighlighting und die korrekte Anbindung an Spachfeatures im Cobra Server verwendet.

Die Zeilen 13 bis 21 zeigen ein Beispiel eines über magische Kommentare eingebundenen Stücks Quellcode. In den Zeilen 13 bis 16 wird ein Stück Quellcode durch das Attribute `hidden` nicht sichtbar eingebunden. In den Zeilen 17 bis 21 wird dann ein Stück aus dem vorher eingebundenen Quellcode über Angabe des im Code befindlichen magischen Kommentars ausgewählt. Cobra sieht dabei für jede Sprache eine Syntax vor, mit der Bezeichner (im Beispiel „myFunc“) angegeben werden können. Magische Kommentare werden in allen, in einer Präsentation eingebundenen Texten gesucht. [3, Advanced Inclusion Options]

2.1.3 Akka Actors

Das Akka Framework beschreibt Aktoren als die kleinsten Bausteine einer Anwendung, welche in Aktorsystemen hierarchische Eltern-Kind-Strukturen formen [6]. Aktoren dienen als Container für Zustand, Verhalten, eine Nachrichten-Warteschlange und weitere Strukturen. Sie kapseln dabei alle diese Dinge hinter einer Aktor Referenz [7]. Das Konzept wurde initial von Hewitt, Bischoff, Steiger [8] und später Agha [9] geprägt.

Aktoren kommunizieren nur über den Austausch von asynchron an Aktor-Referenzen verschickten Nachrichten miteinander und arbeiten nebenläufig. Eingehende Nachrichten an einen Aktor werden in einer Warteschlange gespeichert und sequenziell abgearbeitet. Das heißt, obwohl mehrere Aktoren parallel arbeiten, ist die Programmausführung innerhalb eines Aktors immer sequenziell. Programme in Aktoren bestehen immer aus Reaktionen auf eingehende Nachrichten. Als Reaktion auf eine Nachricht kann ein Aktor die folgenden logischen Funktionen ausführen: Versenden von Nachrichten, Erzeugen neuer Aktoren und Festlegen eines neuen Verhaltens zur Verarbeitung der nächsten Nachricht. [10]

Aktoren können im Akka Framework aus so genannten Properties erzeugt werden. Dabei handelt es sich um Klassen, die alle Informationen, die ein Aktor beim Start übergeben bekommt, enthält. Akka Actors im speziellen sind außerdem mit einer Reihe von Lifecycle Hooks versehen. Sie werden zu fest definierten

Zeitpunkten im Leben eines Aktors ausgeführt [11] und können unter anderem für die Initialisierung sowie ein ordentliches Beenden des Aktors und Aufräumen seines Zustands genutzt werden.

2.2 Language Server Protocol

Das Language Server Protocol (LSP) ist ein von Microsoft entwickeltes Protokoll zur Interprozesskommunikation von Language Servern (LS) und Clients zum Austausch von Informationen über Programmcode im Context von Texteditoren [12]. Es wurde inspiriert von Konzepten aus vim und emacs sowie dem Omnisharp[1] Language Server aus dem .NET-Ökosystem [13]. Das Ziel bei der Entwicklung des Language Server Protokolls war es, eine standardisierte Möglichkeit zur Integration von Language Servern in Editoren zu ermöglichen [14]. Als Language Server (LS) wird in diesem Kontext ein Programm verstanden, das Informationen und Analysen über Quellcodedateien bereitstellen kann. Der Umfang reicht dabei von einfacher Syntaxanalyse bis hin zu tieferen Navigationsfunktionen wie „springe zu Definition“, „finde Aufrufer“ und Typsignaturvalidierung.

Die generelle Struktur des LSP besteht aus über StdIn und StdOut mit einem Editor über JSON-Objekte kommunizierenden Servern. Dabei wird für jeden Editor ein Server gestartet. Das Protokoll sieht keine Nutzung eines Language Servers durch mehrere Editoren vor.

Ziel des LSP ist es ein einheitliches Protokoll zu definieren, welches von Sprachanalysetools verwendet werden kann. So kann durch die Vereinfachung des „m-times-n“ Problems zu einem „m-plus-n“ Problem eine bessere Sprachunterstützung in vielen Editoren ermöglicht werden [15]. Das „n-times-m“ Problem beschreibt, dass für eine Matrix von N Editoren und M Sprachen $N * M$ Integrationen geschrieben werden müssten, um alle M Sprachen in N Editoren zu unterstützen. Durch Nutzung des LSP kann ein Analysetool pro Sprache entwickelt werden (M) und eine Integration des LSP pro Editor (N). So ist es dann für jeden Editor möglich alle Sprachen, die LSP-kompatible Analysetools anbieten, zu unterstützen.

Es gibt für viele Sprachen verfügbare Bibliotheken, die LSP Clients anbieten. [15, LSP Clients]

2.3 Markdown und pandoc

Markdown ist eine 2004 von John Gruber veröffentlichte und in Zusammenarbeit mit Aaron Swartz [16] entwickelte Markup-Sprache [17]. Gruber beschreibt Markdown als:

1. Textbasierte Formatierungssyntax
2. in Perl geschriebenes Softwaretool, welches in Syntax nach 1. geschrieben Text zu HTML umwandelt.

[1] http://www.omnisharp.net/

Gesamtziel des Projektes war es eine Syntax zu schaffen, die in erster Linie unkonvertiert so lesbar wie möglich ist. In Markdown geschriebene Dokumente sollten so wie sie sind, veröffentlichbar sein und zum komfortablen Lesen keine Umwandlung in HTML benötigen. Der Inhalt des Dokumentes sollte klar erkennbar sein und nicht durch Tags oder auffällige Formatierungsdirektive unterbrochen werden müssen. Als Inspiration für die Syntax werden von Gruber das klassische Design von textuellen Emails (keine HTML-Mails) sowie zur Entwicklungszeit existierende Text-Zu-HTML-Software, wie reStructuredText, angeführt [17, Philosophy].

Die Spezifikation von Markdown ist informell und in manchen Sonderfällen nicht eindeutig [18, S. 1.2]. Basierend auf der Spezifikation und Referenzimplementierung von John Gruber haben sich eine Reihe von Markdown Konvertierungstools entwickelt (pandoc, PHP Markdown[2], MultiMarkdown[3], PEGDown[4] und weitere). Jede Implementierung setzt ihre eigene Interpretation der Spezifikation um und bringt ihre eigene Syntax für über die Spezifikation hinausgehende Funktionalitäten mit, was sie in Teilen inkompatibel miteinander macht. In Markdown formatierte Inhalte können je nach verwendetem Konvertierungstool optisch und strukturell unterschiedliches HTML erzeugen [19].

Listing 2.2 zeigt ein Markdown Beispiel, eine Liste mit einem leeren Listenelement. In den folgen Listings (Listing 2.3 bis Listing 2.6) sind die Ergebnisse der Konversion des Markdownbeispiels aufgeführt. Es handelt sich bei diesem Beispiel[5] um eines von vielen, die von Babelmark zur Veranschaulichung der Uneindeutigkeiten zwischen Markdown Konvertern angeführt werden [19]. Unter jedem der Outputlistings sind, der Leserlichkeit halber, jeweils nur einige Tools ausgeführt, die HTML in dieser Art aus dem Beispiel erzeugen. Die meisten Konverter erzeugen das in Listing 2.4 aufgeführte Ergebnis.

```
- one
-
- three
```

Listing 2.2: Markdown: Liste mit leerem Element

[2] https://github.com/michelf/php-markdown
[3] https://github.com/fletcher/MultiMarkdown
[4] https://github.com/sirthias/pegdown
[5] https://johnmacfarlane.net/babelmark2/?normalize=1&text=-+one%0A-%0A-+three%0A

```
<ul>
  <li>one —</li>
  <li>three</li>
</ul>
```

Listing 2.3: MultiMarkdown

```
<ul>
  <li>one</li>
  <li></li>
  <li>three</li>
</ul>
```

Listing 2.4: commonmark, pandoc

```
<h2>— one</h2>
<ul>
  <li>three</li>
</ul>
```

Listing 2.5: Python Markdown

```
<ul>
  <li>
    <h2 id="one">one</h2>
  </li>
  <li>three</li>
</ul>
```

Listing 2.6: kramdown

Es gibt unter dem Titel commonMark[6] eine Initiative, die eine Standardisierung von Markdown ohne Uneindeutigkeiten als Ziel hat. Diese Initiative wird mitgetragen von Jeff Atwood (StackOverflow Mitbegründer), John McFarlane (Entwickler pandoc) und Mitglieder von Firmen, wie StackOverflow, Github und Reddit [20]. Dabei soll ein Basissatz an weitverbreiteten Funktionalitäten in formaler Sprache festgelegt werden. Über automatisierte Tests soll die Konformität neuer Konvertierungstools verifiziert werden können.

Es gibt noch andere Ansätze. Eine Initiative hat über RFCs ein Konzept zur besseren Interoperabilität von Konvertierungstools veröffentlicht. Im März 2016 wurde über den RFC-7763 der HTTP Content-Type `text/markdown` vorgeschlagen [21] und ein IANA Register über Markdown-Varianten angelegt [22]. Ziel dieser Initiative ist es, bei der Übertragung von Markdowninhalten über Header-Parameter den Zeichensatz sowie Konverter des Autors an den Empfänger zu übertragen. Laut des RFCs ist es dem Empfänger möglich, mit Hilfe dieser Informationen den Text so zu konvertieren, wie es vom Autor vorgesehen ist.

2.3.1 pandoc

Der Name pandoc bezeichnet sowohl eine Haskell-Bibliothek als auch die darauf aufbauende Kommandozeilenanwendung zur Konvertierung von Dokumenten zwischen verschiedenen Formaten.

Zusammengesetzt ist pandoc aus einer Reihe von Readern, einer eigenen Darstellung von Dokumenten in Form eines abstrakten Syntaxbaumes (englisch **A**bstract **S**yntax **T**ree, im weiteren Verlauf AST genannt) sowie einer Reihe von Writern. Reader parsen Texte in den selbst definierten AST und Writer generieren aus dem AST Dokumente. Reader und Writer sind dabei frei kombinierbar. Es existiert jedoch nicht für jede Sprache ein Reader und Writer [23, General options]. Zusätzliche Funktionen in Readern und Writern können durch Aus- oder Abwahl von Erweiterungen aktiviert und deaktiviert werden. pandoc's AST bildet nur den Inhalt und die logische Struktur von Dokumenten ab und

[6] https://commonmark.org/

kann nur in eingeschränkter Form die Formatierungen abbilden. Durch diese Einschränkung können selbst bei Konvertierungen von $Format_A \rightarrow Format_A$ Formatierungsdetails verloren gehen. Textuelle Inhalte werden jedoch korrekt und inhaltlich unverändert übernommen. [23, Description]

pandoc Filter: Eine Möglichkeit in pandoc Dokumente verschiedenster Formate zu manipulieren, ist es zwischen Reader und Writer den AST durch so genannte Filter zu verändern. Abbildung 2.1 zeigt die Wege, die ein Dokument durch die Module pandoc's nehmen kann. Dabei bekommt ein Filter einen Baum aus Dokumentenbestandteilen. Er kann diesen Baum durch Einsetzen, Verändern und dem Entfernen von Knoten verändern.

Filter selbst können in jeder beliebigen Programmiersprache entwickelt werden. Sie bekommen den AST in serialisierter Form als Eingabe, können ihn bearbeiten und müssen ihn wieder ausgeben. Es ist möglich, mehrere Filter gleichzeitig einzusetzten. Der AST durchläuft in diesem Fall die Filter nacheinander und kann von jedem verändert werden. Ein Filter stellt eine eigene Anwendung dar und muss deshalb zur Verwendung ausgeführt werden. Um die Ausführbarkeit eines Filters auf allen Systemen zu garantieren, gibt es die Möglichkeit Filter in Lua zu schreiben. Sie können durch den in pandoc integrierten Lua-Interpreter ausgeführt werden [24].

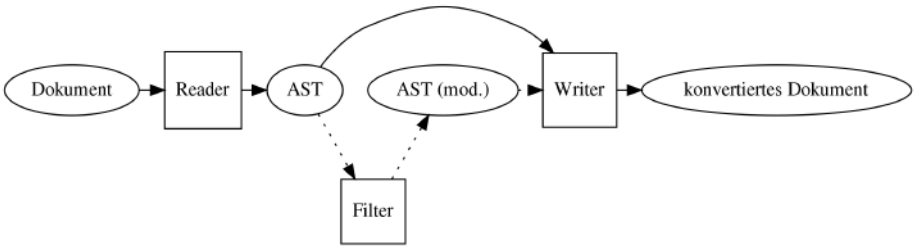


Abbildung 2.1: Informationsfluss einer Dokumentenkonvertierung

2.3.2 Präsentationen in Markdown mit pandoc

Markdown selbst enthält keine Spezifikation zum Schreiben von Präsentationen. pandoc bietet Funktionalitäten an, um dennoch Markdown zu HTML Präsentationen zu konvertieren. Die folgenden Absätze sind, sofern nicht anders angegeben, angelehnt an das Kapitel „Producing Slideshows“ aus dem pandoc Handbuch [23].

Normale Markdown-Dokumente werden bei der Konversion entlang ihrer hierarchischen Struktur von Überschriften und einigen anderen Elementen in Folien unterteilt. Die Regeln dazu sehen wie folgt aus: eine vertikale Linie (horizontal Rule) beginnt immer eine neue Folie, Überschriften auf dem konfigurierbaren Folienlevel beginnen eine neue Folie, Überschriften über dem Folienlevel werden zu Titelfolien konvertiert. Mit Titelfolien kann eine Präsentation in inhaltliche Abschnitte unterteilt und strukturiert werden. Außerdem wird noch eine Ti-

telfolie aus den Metadaten eines Dokuments generiert, sofern diese vorhanden sind.

Bei der Konvertierung zu reveal.js-Folien gibt es noch eine Besonderheit. Die Titelfolien bilden in dem zweidimensionalen Gitter die ersten Folien in der horizontalen Ebene. Nicht-Titel-Folien werden vertikal darunter angeordnet. Beispiele dieser Konvertierung sind in Anhang A.2 zu finden. Dort werden vier Beispiele von Foliensätzen in Markdown und äquivalentem HTML gezeigt. Außerdem wird die aus den Codebeispielen entstehende Struktur grafisch dargestellt. Abbildung 2.2 zeigt einzelne Beispielfolien und den für sie verantwortlichen Markdown Code.

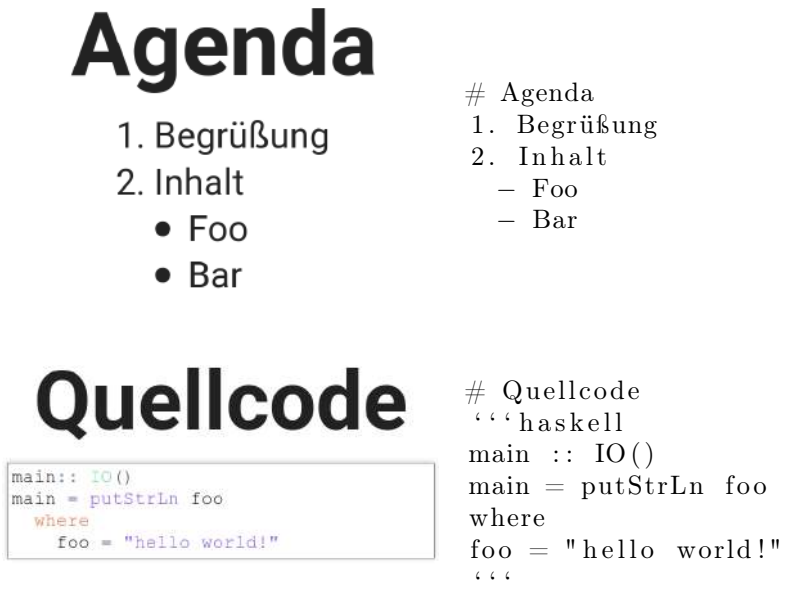


Abbildung 2.2: reveal.js Beispielfolien mit korrespondierem Markdown Quellcode

Folieninhalte werden in Markdown spezifiziert und wie bei normalen Dokumentenkonversionen zu HTML umgewandelt. Texte werden zu HTML-Paragraphen. Listen, Tabellen und Bilder werden zu den standardmäßig in HTML für diese Inhalte vorgesehenen Tags. Quellcode aus Codeblöcken wird übernommen und in `<pre><code>...</code></pre>` eingefasst (vgl. Abbildung 2.2). Präsentationsspezifische Informationen, z.B. ob Listenelemente für die ganze Liste komplett oder nacheinander auftauchen, werden durch `<div>` Elemente mit entsprechenden HTML-Klassen ausgedrückt. Diese entsprechen den HTML Elementen, welche durch reveal.js vorgegeben werden. Zum einfacheren Ausdruck von Divs gibt es das fenced_divs_Plugin in pandoc. Es ermöglicht den Ausdruck von Divs durch drei Doppelpunkte statt der Nutzung von blankem HTML in Markdown. Beides ist möglich und obliegt persönlichen Präferenzen. Beide Notationen werden in Listing 2.7 und Listing 2.8 am Beispiel einer Liste von inkrementell auftauchenden Elementen vorgestellt.

```
<div class="incremental">
  - Punkt 1
  - Punkt 2
  - Punkt 3
</div>
```

Listing 2.7: Markdown Divs HTML

```
::: incremental
  - Punkt 1
  - Punkt 2
  - Punkt 3
:::
```

Listing 2.8: Markdown Divs fenced

Über die fenced_code_blocks und fenced_code_attributes Erweiterungen können durch Akzente abgetrennte Codeblöcke genutzt werden. Sie können durch das Anhängen eines Blocks von geschweiften Klammern mit Klassen und anderen Attributen versehen werden. Diese werden bei der Konversion zu HTML beibehalten.

Meta Informationen: Pandoc kann aus Meta-Informationen, wie Autor und Titel der Präsentation, automatisch eine Titelfolie generieren. Diese Informationen können in einem Metablock (siehe pandoc_title_block Extension [23]) zum Beginn einer Markdown-Datei angegeben werden. In dieser Form können die Meta-Informationen Autor(en), Titel und Datum angegeben werden.

Mit der Verwendung einer Erweiterung ist es möglich, Metadaten im YAML-Format[7] anzugeben, entweder innerhalb eines Blockes frei im Text oder in einer externen Datei. In YAML-Metadatablöcken sind beliebige YAML-Strukturen erlaubt (siehe yaml_metadata_block Extension [23]). Unabhängig von Art, Format und Ort der Angabe haben Filter (siehe 2.3.1) Zugriff auf die Metadaten eines Dokumentes bei der Manipulation von Dokumenten [25].

[7] „YAML Ain't Markup Language“ https://yaml.org/

Kapitel 3

Entwurf

Im Folgenden werden die zur Lösung der Problemstellung erarbeiteten Konzepte und getroffenen Entwurfsentscheidungen vorgestellt. Ebenso wird für jedes Konzept auf relevante bestehende Technologien und ihren Einfluss auf das jeweilige Konzept eingegangen.

3.1 Eingabesprache

Die neue Foliensyntax soll es dem Autor ermöglichen, Folien in einem einfachen Texteditor zu formulieren. Die Syntax soll es ihm ermöglichen, sich auf den Text und die Inhalte zu konzentrieren. Grundlegende Kenntnissen der Syntax sollen ausreichen, um schlichte Folien ausdrücken zu können. Dennoch darf sie erfahrene Autoren nicht beim Schreiben von komplexeren Folien mit Übergängen, nacheinander auftauchenden Inhalten und anderen aufwendigeren Strukturen und Formatierungen einschränken.

Folien in der neuen Syntax sollen, in einem Texteditor gelesen, ihren Inhalt und grundlegende Struktur offenlegen und leicht verständlich sein.

Anders als in der aktuellen Syntax für Folien - HTML mit reveal.js spezifischen Tags und Klassen, wie im reveal.js Handbuch [5] definiert - sollte ein Autor sich nicht mit einer großen Menge an Markuptext, Einrückungen oder Klammern beschäftigen müssen, wenn er die Folien liest.

Das folgende Kapitel beschäftigt sich mit der Syntax zur Erstellung von Folien und wie sie in dieser Arbeit umgesetzt wurde. Kapitel 5 vergleicht die umgesetzte Sprache mit HTML.

3.1.1 Bestehende Technologien

Es gibt bereits eine Reihe von Softwareprojekten, die sich mit dem oben beschriebenen Problem befassen. Viele mit reveal.js verwandte Projekte stützen sich im gleichen Stil auf Folien in HTML. Dabei ist die exakte Syntax von

Projekt zu Projekt unterschiedlich. Wo in reveal.js `<section>` Tags verwendet werden [5], nutzt S5[1] `<div>` Tags mit Klassen wie `<div class='slide'>` [26] und Sqeenote[2] nutzt eine HTML Liste aus `<ol>` und `<li>` Elementen [27]. Es gibt noch mehr Projekte mit denen Präsentationen in HTML im Browser dargestellt werden können. Vielen ist gemeinsam, dass sie geschachtelte HTML Elemente nutzen, um Folien zu unterteilen.

Eine weitere Familie von Tools nutzt einen Markdown Dialekt, reStructuredText oder AsciiDoc zum Ausdrücken von Folien. Ein kompletter Foliensatz wird dabei in den meisten Fällen in eine Datei verfasst. Oft wird eine toolspezifische Syntax genutzt, um Folien voneinander abzugrenzen. In vielen Markdown-unterstützenden Tools, wie GitPitch[3] oder Landslide[4], können drei Bindestriche als Folientrenner genutzt werden.

Manche Tools sind in der Art, wie ihre Folien aufgebaut werden können, nicht immer klar einer der oben beschriebenen Gruppen, also der HTML oder Markdown nutzenden Tools, zuzuordnen. Remark[5] beispielsweise nutzt als Gerüst des Foliensatzes HTML, doch die Folien und der Folieninhalt selbst sind in Markdown geschrieben. Reveal.js ermöglicht es, den Inhalt, jedoch nicht die Struktur, von Folien in Markdown zu schreiben [5, Markdown] und zum Beispiel Markdown als Syntax selbst erlaubt die Verwendung von HTML in Markdown Dokumenten [28, Inline HTML]. Dies ermöglicht es bei Markdown basierenden Tools, welche Folien vor dem Anzeigen zu HTML konvertieren, Inhalte und Folienstruktur in HTML anzugeben.

3.1.2 Markdown als Basissyntax

Wie bereits erwähnt, muss die zu entwickelnde Syntax Folien ausdrücken können. Folien bestehen, vereinfacht gesprochen, aus Titeln, Textelementen, wie Fließtextabsätzen, Auflistungen und anderen Elementen, wie z.B. Bildern. Im Beispiel von Cobra sind es auch Programmabschnitte, deren Quellcode als so genanntes Listing in die Folien eingebunden wird. Diese Elemente werden meist vertikal übereinander gestapelt auf der verfügbaren Fläche angeordnet. Sie können jedoch von Autoren in einer Vielzahl von Wegen, je nach Inhalt und Aufgabe der Folie, abweichend angeordnet werden. Die verfügbare Fläche kann zur einfacheren Anordnung in Unterbereiche geteilt werden. Komplexere Folien können außerdem unter anderem Informationen zu Animationen von Elementen, Hintergründen, Übergangseffekten beim Wechsel zwischen Folien und Links enthalten.

Wie aus dem oberen Absatz zu erkennen ist, enthalten Folien eine Vielzahl verschiedener Elemente und über den eigentlichen Inhalt hinausgehende, die Darstellung der Inhalte betreffende Informationen. Diese unterschiedlichen Informationen bei der Erstellung von Folien in einer textbasierten Syntax darzustellen, erfordert viele und teils komplexe Konstrukte in der zu entwickelnden Syntax. Diese Anforderung steht der Anforderung entgegen, dass in der Syntax

[1] https://meyerweb.com/eric/tools/s5/
[2] https://github.com/julesfern/Squeenote
[3] https://gitpitch.com/
[4] https://github.com/adamzap/landslide
[5] https://github.com/gnab/remark

verfasste Texte im Quelltext einfach und verständlich zu lesen sein sollen und keine aufwendige Lernphase zum Verfassen von Folien erfordern.

Um einen Mittelweg zwischen diesen im Konflikt stehenden Anforderungen zu finden, soll die Syntax auf einer bereits etablierten und bekannten Syntax aufgesetzt werden. So kann bereits bestehendes Vorwissen des Autors genutzt werden, um trotz erhöhter Komplexität und erhöhtem Umfangs der Syntax einen einfachen Einstieg in das Schreiben von Folien zu ermöglichen. Es gibt bereits eine Reihe von Syntaxen, die Texte und Folien, wenn auch manchmal nicht direkt dafür konzipiert, ausdrücken können. Es wurde eine Auswahl aus verschiedenen bestehenden Syntaxen unter den folgenden Auswahlkriterien durchgeführt:

1. Bekanntheit / Verbreitung
2. Leserlichkeit im Quellcode
3. Bestehen der Möglichkeit zum Ausdruck von Folien
4. *(opt.)* Bestehen eines Konverters von Syntax zu HTML / zu HTML im reveal.js Format

Es wird darüber hinaus darauf geachtet, dass die Syntaxen ohne einen übermäßigen Einsatz von Einrückungen und Klammern spezifiziert sind. Ihr Fokus sollte auf Texten und Inhalten liegen, nicht Formatierungszeichen.

In die nähere Auswahl werden die folgenden Syntaxen aus der Familie der leichtgewichtigen Markup Sprachen gezogen: Markdown[6], AsciiDoc[7], Textile[8], MediaWiki[9] und reStructuredText[10]. Im Rahmen dieser Arbeit werden „leichtgewichtige Markup Sprachen“ als Markup Sprachen (vgl. [29]) für Plaintext verstanden. Sie legen einen Fokus auf gute Lesbarkeit des Quellcodes. Bei ihnen ist die Erscheinung des Quellcodes so konzipiert, dass sie einem konvertierten Dokument ähnelt. Ebenso hält sich markupspezifische Syntax bewusst zurück und der Fokus liegt auf den Inhalten.

Tag- und block-basierte Syntaxen, wie XML, XML-Derivate und LaTeX, wurden wegen ihrer erhöhten Komplexität beim Lesen und Schreiben des Quellcodes nicht mit in die engere Auswahl gezogen. Ebenso keine binären Formate, wie Microsoft PowerPoint (.ppt) [30], da diese sich durch ihre Eigenschaft als binäre Datei nicht in Git versionieren lassen. Es wird angenommen, dass Folien neben den zu präsentierenden Code-Beispielen in einem solchen Versionsverwaltungssystem versioniert werden. Zusätzlich ist es nicht realistisch, binäre Dateien in einem schlichten Texteditor zu editieren.

Maschinell erstellte XML-Formate mit grafischen Editoren, wie das Office Open XML Format [31], welches beispielsweise von Microsoft PowerPoint verwendet wird, werden ausgeschlossen, da durch die hohe Komplexität ihrer XML-Syntax das manuelle Lösen von Merge-Konflikten ohne Unterstützung von Tools in einfachen Editoren nicht möglich ist. Ebenso ist auch hier ein Editieren ohne dediziertes Tool komplex.

[6] https://daringfireball.net/projects/markdown/
[7] https://asciidoc.org
[8] https://textile-lang.com/
[9] https://www.mediawiki.org/wiki/Markup_spec
[10] https://docutils.sourceforge.io/rst.html

Alle Sprachen in der näheren Auswahl sind als Sprachen entworfen worden, die unkonvertiert im Quelltext gute Lesbarkeit aufweisen. Mit der Ausnahme von AsciiDoc können sie unter anderem durch das Dokumentenkonvertierungstool pandoc in reveal.js Folien übersetzt werden. Für AsciiDoc ist dies z.B. mit Asciidoctor Reveal.js[11] möglich. Die Syntax zum Ausdrücken von präsentationsspezifischen Informationen ist in jeder Sprache etwas unterschiedlich, jedoch überall möglich. Somit erfüllen alle Sprachen die Auswahlkriterien 2 und 3, sowie das optionale Kriterium 4. Es bleibt also Kriterium 1 übrig.

Für die Betrachtung der Bekanntheit und Verbreitung von Sprachen ist zu beachten, dass Cobra als Werkzeug zum Präsentieren von Code hauptsächlich durch Personen genutzt wird, die Programmcode oder mathematische Beweise präsentieren. Es handelt sich also um Menschen, die mit Programmcode in verschiedenen Formen vertraut sind.

Unter Kriterium 1 wird Markdown als bekannteste bzw. verbreitetste Sprache gewählt, da es unter Programmierer*innen bekannt und etabliert ist. Namenhafte und hochfrequentierte Webseiten, bzw. Plattformen wie Facebook [32], Reddit [33], Github [34] und Stackoverflow [35] nutzen Markdown in einem seiner Dialekte oder ein Subset der Markdown Syntax zur Formatierung von Nutzerinhalten. Neben Github unterstützen weitere Git-Plattform-Anbieter, wie Bitbucket [36] und Gitlab [37], Markdown zur Formatierung von Nutzerinhalten in Readme's und Wikis. AsciiDoc als ebenfalls verbreitete Sprache wird auch von den hier aufgeführten Git-Plattformen unterstützt, jedoch nicht von Plattformen wie Reddit oder Stackoverflow.

Auch wenn Markdown durch seine ungenaue Spezifikation und vielen Implementierungen in Randfällen unterschiedliche Ergebnisse beim Konvertieren ausweisen kann [18], sind einfache Dokumenten über beinahe alle größeren Implementierungen in ihrer konvertierten Form ähnlich. Es ist deshalb anzunehmen, dass die grundlegende Markdown-Syntax eine sehr hohe Bekanntheit aufweist und Nutzer*innen nur die präsentationsspezifischen Syntax-Bestandteile erlernen müssen. Auf diese Besonderheiten beim Ausdruck von Folien im Vergleich zu Text wird in Abschnitt 2.3.2 eingegangen.

Wegen seiner weitreichenden Verbreitung im Internet und in Tools (welche größtenteils von Programmierer*innen genutzt werden), seiner im Quelltext lesbaren Syntax und Einfachheit ist Markdown als Sprache zum Ausdruck von Folien gewählt worden. Im Kapitel 5 Evaluation wird die Nutzbarkeit von HTML und Markdown in einem Nutzertest verglichen und die Eignung von Markdown zum Schreiben von Folien evaluiert.

Markdown muss, bevor es im Browser angezeigt werden kann, zu HTML konvertiert werden. Es wurde erwogen, einen eigenes Tool zur Konvertierung von Markdown zu HTML für reveal.js Präsentationen auf Basis der commonMark Markdown-Spezifikation zu entwickeln, anstatt einen bereits existierenden Konverter zu benutzen. Durch eine Eigenentwicklung wurde sich größtmögliche Flexibilität bei der Umsetzung von Quellcodeblöcken für Cobra (vgl. Abschnitt 2.1.2 und 2.3.2) und Projektdefinitionen (vgl. Abschnitt 4.1.1) erhofft. Auf Grund des hohen Entwicklungsaufwandes und zu diesem Zeitpunkt noch instabilen commonMark-Spezifikation und damit verbundenem zukünftigen War-

[11] https://asciidoctor.org/docs/asciidoctor-revealjs/

tungsaufwand wurde sich gegen eine Eigenentwicklung entschieden. Stattdessen wird ein bestehendes Konvertierungstool gesucht, das alle Anforderungen erfüllt.

Pandoc ist als Konverter von Markdown zu reveal.js gewählt worden, da es zum einen auf allen größeren Betriebssystemen in Form von Installern und Paketen verfügbar ist. Mit einer langen Entwicklungsgeschichte (initiales Release in 2006) und aktiven Entwicklergemeinschaft wird es auch heute noch mit Updates versorgt und seine umfangreiche Dokumentation gepflegt. Zum anderen bietet die von Haus aus eine breite Menge an Input- und Output-Formatoptionen, welche durch Plugins und eigene Module erweitert und angepasst werden können, eine hohe Flexibilität für zukünftige Projektanforderungen. Mit der Entscheidung pandoc zu nutzen, werden die in Abschnitt 2.3.2 beschriebenen Syntaxregeln für Folien und Foliensätze durch das Tool vorgegeben.

3.1.3 Cobraspezifische Informationen in Markdown

Zur Referenzierung von Codebeispielen, welche nicht Teil des Präsentationstextes sind, muss an Codeblöcken das `src` Attribut gesetzt werden. Zusätzlich muss der Name der Sprache des Beispiels für korrektes Code Highlighting angegeben werden.

In HTML wurde dies über ein HTML-Attribut und eine entsprechende Klasse getan. In Markdown geschieht dies in geschweiften Klammern hinter den Zeichen zur Definition des Codeblockes. Paare nach dem Format `key=value` werden während der Konvertierung zu HTML von pandoc zu HTML-Attributen und alleinstehende Wörter zu Klassen übersetzt. Die Syntax für diese Definition wurde von pandoc vorgegeben und übernommen (siehe fenced_code_attributes Extension [23]). Es sind beliebig viele Attribute und Klassen möglich, für Cobra sind die Attribute `src`, `from` und `to` relevant. Listing 3.1 zeigt ein Beispiel der Codeblocksyntax in HTML und Markdown mit einem Stück Scalacode. Der im Beispiel stehende Code im Codeblock ist zu Anschauungszwecken dort eingetragen worden und entspricht dem Inhalt der referenzierten Datei `sample.scala` in den Zeilen 8 bis 10 einschließlich.

```
<code class="scala" src="sample.scala" from="8" to="10">
        func main(){
                print("Hello World!")
        }
</code>

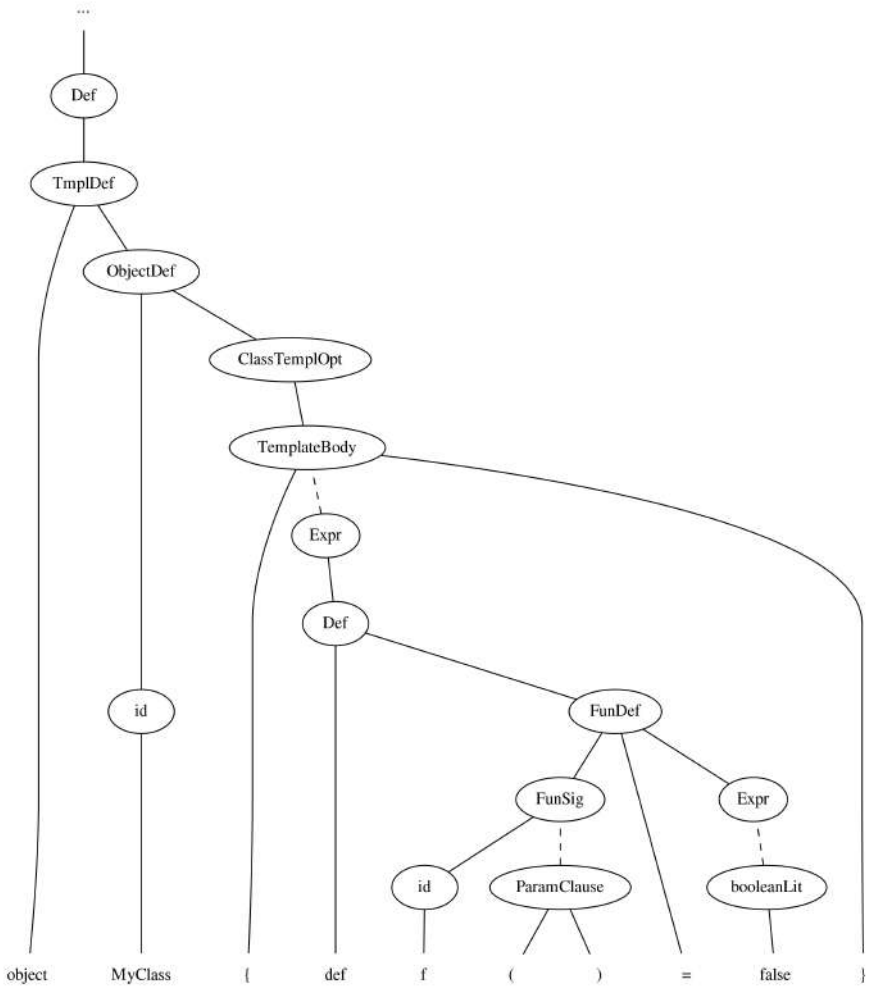
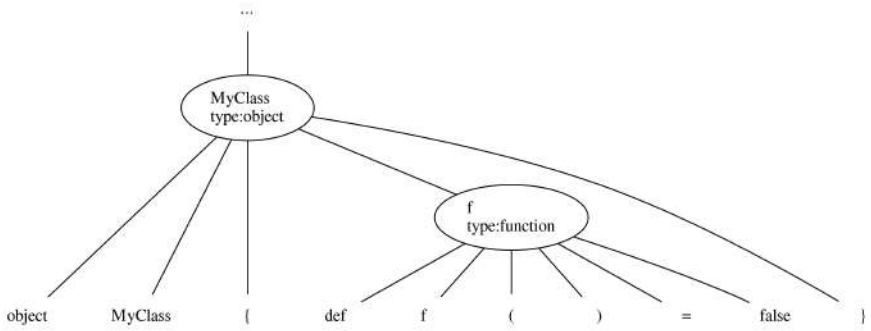
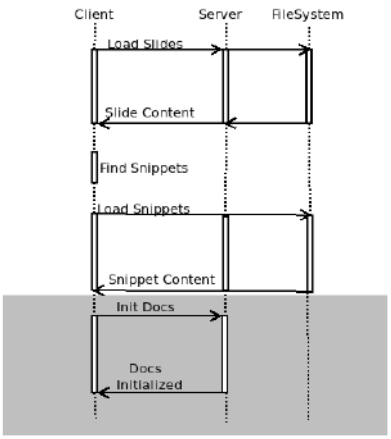
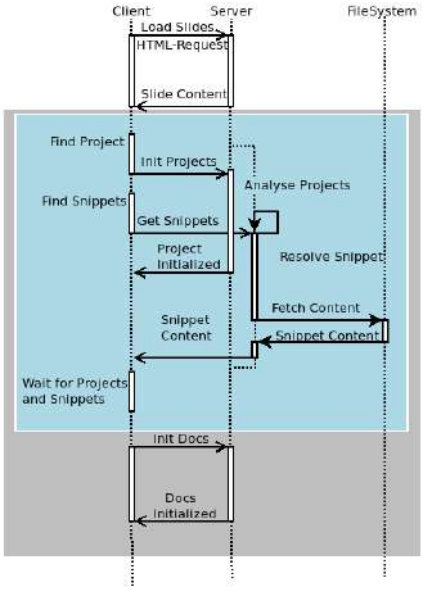
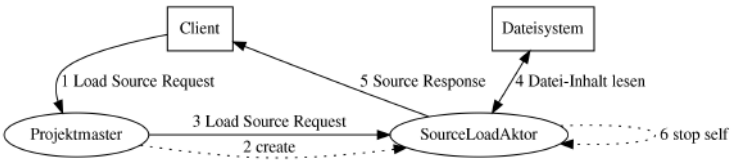
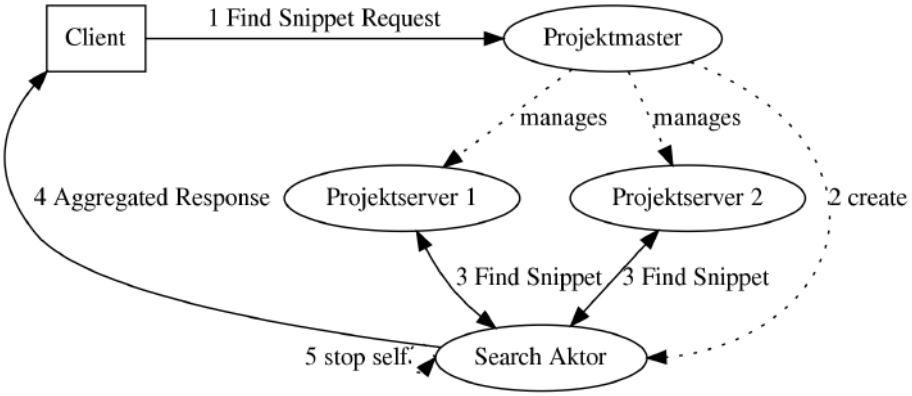
``` {scala src="sample.scala" from=8 to=10}
        func main(){
                print("Hello World!")
        }
```
```

Listing 3.1: Angabe von Cobra Informationen in Markdown

Die Syntax für Folienstruktur und an dieser Stelle nicht aufgeführte Folieninhalte entspricht der in Abschnitt 2.3.2 beschriebenen Syntax.

3.2 Abfragesprache

Codebeispiele in Cobra werden als speziell kodierter String im `src` Attribut eines Codeblocks definiert. Im Rahmen dieser Arbeit soll eine Syntax entwickelt werden, im Folgenden Abfragesprache genannt, die die bestehenden Möglichkeiten zur Definition, wie Dateipfade mit Zeilenangaben und im Quellcode enthaltene magische Kommentare [3, Advanced Inclusion Options], erweitern soll. Dabei werden die bestehenden Möglichkeiten nicht ersetzt, stattdessen wird eine weitere geschaffen, die neben den existierenden zur Verfügung stehen wird.

Es wird sich gegen die Adaption eines bestehenden Konzeptes aus einem der in Abschnitt 3.2.1 beschriebenen Tools entschieden. Mit dem Grund, dass sie entweder nicht alle Anforderungen der Problemstellung erfüllen oder nicht erwartet wird, dass sie vor allem in großen Quellcodebeständen intuitiv nutzbar sind. Stattdessen wird sich für den Entwurf eines eigenen Konzeptes entschieden, das in den folgenden Kapiteln beschrieben wird.

3.2.1 Bestehende Technologien

Die Auswahl von Ausschnitten aus größeren Quellcodedateien an sich ist ein in verschiedenen Kontexten auftretendes und bekanntes Problem. Es gibt einige Projekte, welche bereits Lösungen für dieses Problem in ihren jeweiligen Kontexten erarbeitet haben. In Confluence, einem von Atlassian entwickelten Tool für Kollaboration und Wissensmanagement[12], gibt es ein Plugin zum Einbinden von Quellcode aus externen Quellen[13]. Es ermöglicht die Auswahl von Teilen einer Quelle. Zwei Auswahlmöglichkeiten sind dabei besonders interessant. Entweder kann ein Literal angegeben werden, das exakt in der ersten Zeile des Beispiels vorkommen muss und dann diese Zeile bis zum Ende der Quelle als Beispiel definiert oder es kann ein regulärer Ausdruck genutzt werden. Sollte der Ausdruck zwei Gruppen enthalten, beschreibt die erste Gruppe die Anfangszeile und die zweite Gruppe die Endzeile des Beispiels.[38] Ähnliches ist auch in LaTeX mit dem Listings Paket möglich. Dort kann eine Syntax für Textmarker definiert werden. Über diese Marker kann dann ein Abschnitt in einer Datei ausgedrückt werden. [39, Abschnitt 5.7]

Die direkte Angabe von Textstellen in einer Datei, wie in Confluence, ist im Rahmen dieser Arbeit unzureichend, da kein Ende eines Beispiels definiert werden kann. Würde über dieselbe Methodik auch ein Ende einer Textstelle definiert werden, nehmen wir an, der Namen einer Klasse und eine schließende geschweifte Klammer werden als Start und Ende ausgewählt, dann endet die Textstelle bereits nach der ersten Funktionsdefinition in der Klasse, da diese auch mit geschlossenen geschweiften Klammern den Block ihres Funktionskörpers beendet. Ebenso ist die Auswahl von extra in den Quellcode eingefügten Start- und Endmarkern sowohl im Confluence Plugin als auch im Listings Paket suboptimal, da es eine Veränderung des Quellcodes für die Präsentation darstellt und in seiner Funktionsweise genauso funktioniert, wie die bereits in Cobra vorhandenen Abschnittsmarker.

[12] https://www.atlassian.com/software/confluence
[13] https://bobswift.atlassian.net/wiki/spaces/CODE/overview

Reguläre Ausdrücke können sehr flexibel zur Auswahl passender Zeilen genutzt werden, sind jedoch zeitaufwendig zu entwickeln und zu verstehen. Ebenso geben sie beim Lesen nur in sehr technischer Ausdrucksweise Aufschluss über die tatsächlichen Intentionen des Autors.

3.2.2 Abstraktion von Sprachen

Programmiersprachen können aus verschiedenen Sichten betrachtet werden. Für die Definition von Quellcodebeispielen ist nicht die logische Funktion eines Abschnittes wichtig oder seine exakte Syntax. Stattdessen ist die allgemeine Struktur des Programmcodes relevant. Bei der Definition von Programmbeispielen ist es häufig zweckmäßig, aus geschachtelten Programmstrukturen einen Teil herauszuheben und auf Folien anzuzeigen. Dies können zum Beispiel die Definition einer Variable, einer Funktion, einer Klasse, eines Typs oder Structs sein.

Quellcode einer Sprache wird von einem Compiler während der Kompilation oder Interpretation zu einem abstrakten Syntaxbaum umgewandelt [40, S. 198]. Dieser Baum enthält alle Regeln der Syntax, die der Parser des Compilers anwenden musste, um von terminalen Symbolen, wie Buchstaben, Zahlen und Whitespace zu Programmkonstrukten zu kommen.

Je nach Syntax der Sprache haben alle Elemente andere Namen und sind unterschiedlich aufgebaut. Für die Definition von Abschnitten im Programmcode geben sie einen ersten Blick auf die Struktur eines Programms. Dieser ist jedoch zu detailreich. Deshalb sollen Syntaxbäume vereinfacht werden, um auf diesen Vereinfachungen wiederum Quellcodeabschnitte definieren zu können.

Zur Erarbeitung einer Vereinfachung dieser Syntaxbäume ist im Listing 3.2 ein Ausschnitt aus der Scala Syntax Definition abgedruckt.

```
Def       ::=  PatVarDef
             |  'def' FunDef
             |  'type' {nl} TypeDef
             |  TmplDef

FunDef    ::=  FunSig [':' Type] '=' Expr
             |  FunSig [nl] '{' Block '}'
             |  'this' ParamClause ParamClauses
                ('=' ConstrExpr | [nl] ConstrBlock)

TypeDef   ::=  id [TypeParamClause] '=' Type

TmplDef   ::=  ['case'] 'class' ClassDef
             |  ['case'] 'object' ObjectDef
             |  'trait' TraitDef
TraitDef  ::=  id [TypeParamClause] TraitTemplateOpt


Dcl       ::=  'val' ValDcl
             |  'var' VarDcl
             |  'def' FunDcl
```

```
          |   'type' {nl} TypeDcl

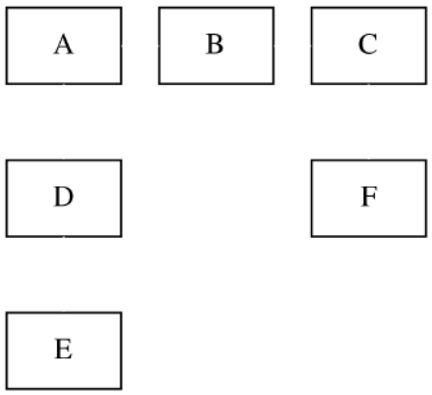
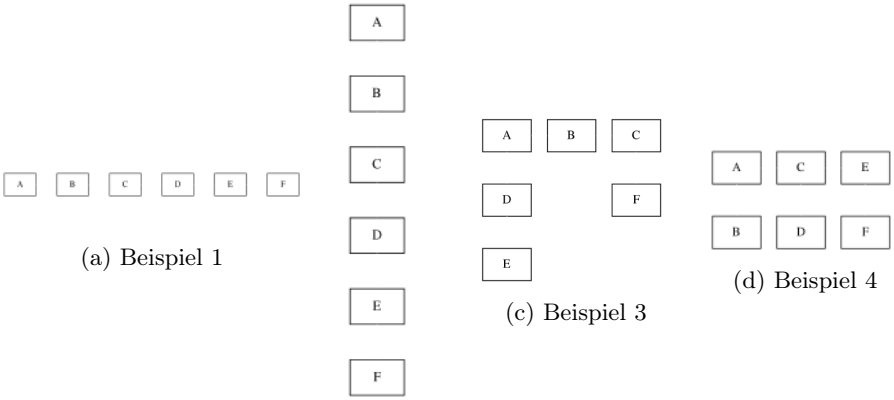
ValDcl    ::=  ids ':' Type
FunDcl    ::=  FunSig [':' Type]
FunSig    ::=  id [FunTypeParamClause] ParamClauses
TypeDcl   ::=  id [TypeParamClause] ['>:' Type] ['<:' Type]
```

Listing 3.2: Auszug aus [41]

Definitionen (`Def`) und Deklarationen (`Dcl`) enthalten in ihrer Syntaxdefinition meistens Identifier (vgl. Listing 3.2). Für Definition (`Def`), Funktionendefinitionen (`FunDef`), Traitdefinition (`TraitDef`) sowie Deklarationen (`Dcl`) und Values Deklarationen (`ValDcl`) ist dies ebenfalls im Listing zu erkennen. Die Ausnahme zu der Regel, dass Definitionen Identifier enthalten, bilden Konstrukte, wie Lambdas. Diese müssen nicht zwingend einem Identifier zugewiesen werden. Ein vergleichbarer Aufbau lässt sich für viele Konstrukte in Haskell aus der Syntaxdefinition ableiten [42, 10 ff] .

Wird ein Programm in einen Syntaxbaum übersetzt, entsteht eine hierarchische Struktur von Konstrukten welche Identifier enthalten und in sich wiederum Konstrukte kapseln, die auch ihrerseits Identifier enthalten können. Ein intuitives Beispiel ist eine Klasse oder Modul, in der Funktionen definiert werden, welche ihrerseits wieder Variablen oder ähnliches deklarieren. In Listing 3.3 wird ein Objekt definiert, das in sich eine Funktion definiert. Dieser Quellcode wird im Folgenden als Beispiel dienen.

Die Eigenschaft, dass Programme als hierarchischer Baum aus Definitionen und Deklarationen von Konstrukten mit Identifiern dargestellt werden können, ist die Grundlage der Abfragesprache.

Zur Abfrage wird aus dem Syntaxbaum ein vereinfachter Baum nach dem folgenden Schema aufgebaut:

1. Beginne mit dem Root-Knoten im Syntaxbaum
2. Lege für jede Definition und Deklaration mit Identifier einen Knoten im vereinfachten Baum an, der den Identifier der Definition beziehungsweise Deklaration sowie die Position und Typen der Definition im Quelltext enthält
3. Hänge den in 2. erzeugten Knoten so in den vereinfachten Baum, dass die Eltern-Kind-Beziehungen aus dem Syntaxbaum erhalten bleiben
4. Wiederhole diesen Algorithmus ab 2. für jeden Kindknoten

Grafisch dargestellt vollführt der Algorithmus die Transformation des Syntaxbaums (Abbildung 3.1a) des in Listing 3.3 gezeigten Programmabschnittes zum vereinfachten Baum (Abbildung 3.1b) nach dem oben beschriebenen Prinzip. Abbildung 3.1a ist an einigen Stellen der Leserlichkeit halber vereinfacht worden. Diese sind durch gestrichelte Kanten zwischen den Knoten gekennzeichnet.



(a) Syntaxbaum



(b) vereinfachter Baum

Abbildung 3.1: Vereinfachung eines Syntaxbaumes am Beispiel von Listing 3.3

```
1 object MyClass {
2     def f() = false
3 }
```

Listing 3.3: Beispielcode für Baumvereinfachung

Abfragen von Codeabschnitten werden auf diesen vereinfachten Syntaxbäumen von Programmen definiert und nicht mehr auf dem kompletten Syntaxbaum oder Quellcode direkt.

3.2.3 Aufbau der Abfragesprache

Abfragen in der Abfragesprache werden als Pfad in vereinfachten Syntaxbäumen definiert. Ein Pfad von der Wurzel zu einem Knoten referenziert dabei den im Knoten enthaltenen Quellcode. Es müssen keine Blätter des Baumes getroffen werden, da Elternknoten durch die Funktionsweise des im vorherigen Abschnitt definierten Algorithmus zur Umformung, den kompletten Quellcode aller Kinder ebenfalls referenzieren. Abfragen nach einem Nicht-Blatt-Knoten dieses Baumes referenzieren also den kompletten Teilbaum, der an dem ausgewählten Knoten hängt. Somit den kompletten Code, dessen Vereinfachung zu der Erzeugung des Teilbaumes geführt hat.

Ähnlich wie XPath-Abfragen haben Pfade das Ziel, Teile eines abstrakten Baumes anzusprechen, der ein Dokument auf Basis seiner logischen Struktur darstellt [43]. Im Kontext von Cobra ist es nicht vorgesehen, mehr als einen Knoten gleichzeitig auszuwählen oder Muster von Knoten zu finden. Deshalb kann der Abfragesprachenaufbau sogar gegenüber den syntaktischen und semantischen Optionen der initialen XPath Version 1.0 [44] vereinfacht werden.

Ebenso wird keine bestehende Abfragesprache für Graphdatenbanken, wie Cypher aus dem neo4j Umfeld[14] oder das vom W3C spezifizierte SPARQL[15], verwendet. Grundsätzlich ist dies möglich. Der vereinfachte Syntaxbaum lässt sich als Graph mit einer Art von Knoten (`Definition`) und einer Art von Kanten (`istKindVon`) modellieren. Darauf könnten Abfragen in Cypher oder vergleichbaren Abfragesprachen gestellt werden. In ihrem Funktionsumfang sind beide Sprachen jedoch viel zu umfangreich und es würde nur ein kleiner Teil ihrer Spezifikation umgesetzt werden. Mit ihren umfangreichen Funktionsumfängen geht bei beiden Sprachen eine erhöhte Komplexität von Abfragen einher. Mit dem Ziel die Leserlichkeit und intuitive Verständlichkeit von Pfaden zu priorisieren, wird sich deshalb gegen die Adoption bestehender Abfragesprachen entschieden.

[14] https://neo4j.com/developer/cypher-query-language/
[15] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

Bisher wurden Pfade als einfaches Mittel beschrieben, Knoten aus einem Baum auszuwählen. In vereinfachten Syntaxbäumen sind Pfade alleine jedoch nicht ausreichend, um eine Abfrage in allen Fällen genau ein Codebeispiel definieren zu lassen. Es gibt in Scala beispielsweise die Möglichkeit, dass unter dem gleichen Pfad in der gleichen Datei sowohl eine Klasse als auch ein Objekt mit dem gleichen Identifier definiert werden kann. Dieses Objekt wird „Companion Object“ der Klasse genannt [41, S. 5.5]. Im vereinfachten Baum bekommen Klasse und Objekt den gleichen Pfad.

Um diese Uneindeutigkeit zu beheben, sind Schlüssel eingeführt worden. Sie haben die Funktion von Filtern und schränken den Suchraum zum Auflösen der Pfade ein. Es sind die folgenden Schlüssel eingeführt worden:

project: Identifier eines Projektes in dem gesucht werden soll

type: Typ des gesuchten Knoten

Um in der oben beschriebenen Pfadkollision die Uneindeutigkeit aufzulösen, bedarf es der Angabe eines Knotentypen.

3.2.4 Syntax

Pfade in den vereinfachten Syntaxbäumen nach Abschnitt 3.2.2 bestehen aus einer Reihe von Identifiern. Diese müssen zur Angabe in Präsentationen in eine Textform gebracht werden. Dabei werden Pfade aus Identifiern und Trennzeichen zwischen ihnen zusammengesetzt. Soll zum Beispiel aus einer Klasse mit dem Namen `MyClass` im Paket `util` die Funktion `function1` ausgewählt werden, müssen diese Identifier in der Form `util <Trennzeichen> MyClass <Trennzeichen> function1` zusammengefügt werden.

Pfade und ihre Darstellung in Textform sind aus vielen Bereichen der Informatik bekannt - unter anderem aus Dateipfaden in Betriebssystemen. Dort werden Pfade als Liste von Ordnernamen, je nach Betriebssystem getrennt durch ein „/“ oder „\“, formuliert. Sie sind außerdem bekannt aus dem Aufbau von Abfragen in XPath, wo jeder Schritt einer Abfrage ebenfalls durch „/“ getrennt ist [43]. Pfade werden aber auch in Programmiersprachen selbst an verschiedenen Stellen genutzt, wie zum Beispiel bei Definition, Import von Paketen / Modulen, oder dem Aufruf von Variablen und Funktionen aus Objekten und Modulen. In Haskell [42, S. 11] und Scala [41, Kapitel 13.2] werden die Identifier der Pfade dabei durch einen Punkt getrennt. Dies setzt sich auch in Dokumentationstools fort. In JavaDoc bzw. ScalaDoc für Scala und Haddock für Haskell wird diese Schreibweise ebenso verwendet, um in der Dokumentation Links zu anderen Typen, Klassen, Modulen, usw. zu erzeugen [45, S. 3.8.7][46]. Auch Doxygen[16] als Dokumentationstool, für unter anderem C++, nutzt Pfade aus Klassen- und Funktionsnamen getrennt durch Doppelpunkte [47], um Links zu diesen Elementen in der Dokumentation zu generieren. Wie auch in JavaDoc und Haddock wird das in der Sprache definierte Trennzeichen für Identifier verwendet.

[16] http://www.doxygen.nl/index.html

Je nach Auswahl des Trennzeichens ist es möglich, das Beispiel von oben wie folgt darzustellen:

- `util/MyClass/function1`
- `util\MyClass\function1`
- `util::MyClass::function1`
- `util.MyClass.function1`

In der Syntax werden Punkte verwendet, um Pfadelemente zu trennen, da erwartet wird, dass diese sich für Programmierer*innen durch ihre Verwendung im Aufruf von geschachtelten Strukturen im Quellcode, natürlich anfühlen. Es werden Punkte statt Doppelpunkte verwendet, da im Rahmen dieser Arbeit zuerst Haskell sowie Scala unterstützt werden und Punkte die dort verwendeten Trennzeichen darstellen. Die These, dass Pfade durch ihre Ähnlichkeit in Quellcode verwendeten Identifiern intuitiv verständlich sind, gilt es in der Evaluation (Abschnitt 5.3.3) zu testen.

Schlüssel zur Suche werden dem eigentlichen Pfad vorangestellt und sind durch eckige Klammern begrenzt. Sie bestehen aus dem Anfangsbuchstaben des Schlüssels und dem Wert, auf den die Suche beschränkt werden soll, also dem Knotentypen oder Projektnamen. Schlüsselbuchstabe und Wert sind durch einen Doppelpunkt voneinander getrennt. Soll die bis jetzt genutzte Beispielabfrage auf den Ergebnistypen „function“ beschränkt werden, muss die Abfrage wie folgt abgewandelt werden: `[t:Function] util.MyClass.function1`

Die EBNF von Pfaden wird in Listing 3.4 dargestellt. Zeichenranges von `'1'` bis `'3'` jeweils einschließlich werden in der Form `'1' - '3'` angegeben.

```
Char    ::= 'a' - 'z' | 'A' - 'Z'
Digit   ::= '0' - '9'
WSpace  ::= ' ' | '\t'
Sep     ::= '.'
KeyChar ::= 'a' - 'z'
Elem    ::= (Char | Digit)+
Key     ::= '[', KeyChar, ':', (Char | Digit)+, ']'
Keyset  ::= Key, ((WSpace*), Key)*
Path    ::= Elem, (Sep, Elem)+
Query   ::= Keyset, (WSpace)*, Path
```

Listing 3.4: EBNF Abfragesprache

In der EBNF (3.4) ist zu erkennen, dass Schlüssel für Typen nur vor den Pfad gesetzt werden können. Es ist somit nur möglich, den Typen des Knotens am Pfadende zu bestimmen und nicht auf jeder Ebene. Dies senkt die Ausdrucksfähigkeit der Pfade, da nun Namenskollisionen in der Mitte eines Pfades nicht mehr aufgelöst werden können, sollte der Rest des Pfades unter allen, an der Kollision beteiligten Knoten existieren. Diese Vereinfachung der Syntax wird vorgenommen, um die Lesbarkeit der Abfragen zu erhöhen. Im Abschnitt 3.2.5 wird genauer auf das Verhalten bei uneindeutigen Pfaden eingegangen.

3.2.5 Semantik

Im Folgenden wird auf die Bedeutung und das Verständnis einiger Teile einer Abfrage aus Abschnitt 3.2.4 eingegangen. Die Beschreibungen der Semantik sind als Handreiche für Implementierende gedacht und spezifiziert, wie einige Aspekte der Abfragesprache in ihrer Funktion umgesetzt werden sollten.

Auflösung von Pfaden: Pfade werden beim Auflösen ohne Beachtung ihrer Groß- und Kleinschreibung abgelaufen. Diese Entscheidung wird getroffen, um die Schreibbarkeit von Abfragen auf Kosten ihrer Genauigkeit zu vereinfachen.

Pfade werden immer versucht vorwärts und rückwärts aufzulösen. Im Fall der Vorwärtsauflösung wird versucht, vom Root Knoten ausgehend, den Pfad bis zu einem Knoten zu verfolgen. Es ist dabei nicht relevant, ob es sich um einen Blattknoten handelt oder nicht. Beim Rückwärtsauflösen wird jeder Knoten mit dem Identifier am Ende des Pfades gesucht und versucht den vorhandenen Pfad rückwärts von diesen Knoten ausgehend zu durchlaufen. Ist dies bis zum Ende des Pfades in der Abfrage möglich, wird der Knoten als Ergebnis des Pfades angesehen. Die Vereinigung beider Ergebnismengen ist das Ergebnis eines Pfades.

Dieses Verhalten wird gewählt, um in kleinen Beispielprojekten einfache Abfragen nach bestimmten Identifiern zu ermöglichen. Wenn es nur eine Funktion mit dem Namen `myFunc` gibt, ist es möglich nur diesen Namen anzugeben, um die Definition als Beispiel zu definieren. In größen Beispielprojekten ist es nach wie vor möglich, komplette Pfade von der Wurzel bis zum gesuchten Element anzugeben, um eindeutig Beispiele zu definieren.

Auflösung von Positionen: Wie in Abschnitt 3.2.2 erwähnt, enthält jeder Knoten die Position der Definition, aus der er erzeugt wurde. Anhand dieser Definition wird das letztendliche Quellcodebeispiel gefunden. Die exakte Art der Auflösung ist dabei von Sprache zu Sprache unterschiedlich. Sie kann einen variablen Grad an semantischem Wissen über die Quellcodedatei bzw. die Projektstrukturen, aus der die Datei stammt, verlangen. Im einfachsten Fall kann diese Position zu einem Dateipfad sowie Start und Endzeilen im Kontext der Datei aufgelöst werden.

Schlüssel: Sie schränken die Ergebnismenge ein. Alle Schlüssel sind optional. Wird ein Schlüssel mehrfach definiert, wird nur die letzte Definition beachtet. Die Reihenfolge ist dabei analog zu ihrer Position in einer Zeile.

Für den Projektschlüssel wird in der Präsenz von mehreren Projektdefinitionen nur in dem Definitionsbaum des Projektes gesucht, welcher mit dem im Schlüssel angegebenen Projektnamen assoziiert ist. Für den Typenschlüssel wird ein Knoten nur als Teil der Ergebnismenge angesehen, wenn sein Pfad, wie in 3.2.5 beschrieben, aufgelöst werden kann und sein Typ dem geforderten Typ entspricht.

Verhalten bei uneindeutigen Abfragen: Abfragen auf einem Baum sind nur gültig, wenn sie zu null oder einem Knoten aufgelöst werden können. Das Ergebnis einer Anfrage ohne passende Knoten ist eine leere Ergebnismenge. Abfragen, die zu mehr als einem Endknoten aufgelöst werden können, sind nicht zulässig. Es wird von der Implementierung erwartet, dass dem Nutzenden eine Fehlermeldung ausgegeben wird. Diese kann zur leichteren Behebung alle getroffenen Knoten und fehlenden Schlüssel aufzeigen, um die Abfrage eindeutig zu gestalten.

3.2.6 Stabilität der Abfragen

Eine geforderte Eigenschaft der Abfragesprache sollte sein, dass sie robuster gegenüber Dateiänderungen ist als die aktuelle Methode mit Dateipfaden und Zeilenangaben. Robuster wird in diesem Kontext wie folgt definiert: Es ist weniger wahrscheinlich, dass eine Dateiänderung in einem Bereich außerhalb des Beispiels dazu führt, dass sich der Inhalt eines Beispiels verändert.

Bei Dateipfaden und Zeilengaben können Änderungen des Dateinamens und Ordnerpfades einer Datei ebenso zu Änderungen des Beispielinhaltes führen, wie das Einfügen sowie Entfernen von Zeilen vor dem Beispiel. Änderungen des Dateiinhaltes vor dem Beispiel können dazu führen, dass sich der durch Zeilennummern ausgewählte Bereich verschiebt und nur noch einen Teil des beabsichtigten Beispiels umfasst. Inhaltliche Änderungen der Datei, wie das Umbenennen der in einer Datei beschriebenen Klasse oder eines Modules ohne eine Änderung der Anzahl von Zeilen, führen zu keinem Verschieben des Beispieles.

Die Abfragesprache ist nicht auf Zeilennummern oder Dateipfade angewiesen. Änderungen an ihnen führen nicht zu einer Änderung eines in der Abfragesprache ausgedrückten Beispiels, da sich Abfragen immer auf alle Dateien eines Projektes beziehen. Anders als Dateipfade und Zeilen ist die Abfragesprache anfällig für Änderungen von Klassen oder Modulnamen, da sie Beispiele über die Schachtelung dieser definiert.

Es wird davon ausgegangen, dass die Abfragesprache gegenüber normalen Dateiänderungen robuster ist. Änderungen, wie das Umbenennen einer Klasse führen in Programmen zu weitreichenden Änderungen in vielen Klassen. Sie können in weiten Teilen eines Programms das Umbenennen von Importen und Aufrufen nötig machen. Es wird angenommen, dass Quellcode-Autoren sich diesen Implikationen bewusst sind und bei der Umbenennung einer Klasse und nachfolgenden Änderung des Programmcodes auch an die Änderung der Beispielpfade denken. Häufig vorgenommene Änderungen, wie das Entfernen einer überflüssigen Leerzeile oder Importes, werden oft nur zur Verbesserung des optischen Erscheinungsbildes des Quellcodes vorgenommen - meist ohne Beachtung, dass sie die Bedeutung von Beispielen verändern können. Gerade gegen diese kleinen unscheinbaren Veränderungen ohne vermeintliche Seiteneffekte sind Zeilenangaben besonders anfällig und Pfade in der Abfragesprache nicht.

Kapitel 4

Integration

Im Rahmen dieser Arbeit ist es ein Anliegen neben der Funktionsweise der Konzepte aus Kapitel 3 auch ihre praktische Umsetzbarkeit zu demonstrieren. Die Konzepte werden deshalb in Cobra implementiert. Bei der Umsetzung wird auf eine in ihrer Funktion alle Teile der Konzepte abdeckende erweiterbare Implementierung Wert gelegt. Nutzerfeedback für Endnutzer*innen im Cobra Client wird nur in grundlegender Form implementiert.

Dieses Kapitel wird die bei der Umsetzung getroffenen Entscheidungen, Abwägungen und Details der Implementierung vorstellen. Es folgt dabei der Startsequenz von Cobra. Die Beschreibung beginnt mit der Konvertierung von Folien aus Markdown (Abschnitt 4.1), gefolgt von der Startsequenz des Clients und den Tätigkeiten des Servers. Auf Serverseite werden die Analyse von Projekten (Abschnitt 4.4) und die Suche von Beispielen (Abschnitt 4.5.2) vorgestellt.

Verweise auf den erstellten Quellcode: Im folgenden Text wird auf den Quellcode aus der Implementierung verwiesen. Der Quellcode ist online zu finden [48]. Wenn sich die in einem Kapitel beschriebenen Änderungen auf einen bestimmten Teil des Quellcodes beziehen, wird dieser angegeben. Zur besseren Leserlichkeit wird darauf verzichtet Änderungen, die mehrere Stellen im Projekt betreffen, mit allen Stellen anzugeben. Außerdem wird die folgende Form für Referenzen auf Quellcode eingeführt.

Das Beispiel [48, cobra-server Cobra myFunc()] hat die folgende Bedeutung: `cobra-server` gibt das betreffende Modul an. Module sind online im Git vom Basisverzeichnis ausgehend, unter `modules/<modul name>` zu finden. `Cobra` ist die Angabe eines Klassennamens. Aus Gründen der Leserlichkeit wird nur, wenn nötig, ein Package mit angegeben. Bei fehlender Angabe ist das Standardpackage in allen Modulen `net.flatmap.cobra` gemeint. Das hier verwendete Beispiel beschreibt die Datei `Cobra.scala` im Ordner `modules/cobra-server/src/main/scala/net/flatmap/cobra`. Nach dem Klassennamen kann noch ein Verweis auf eine Codestelle in dieser Datei erfolgen. Hier im Beispiel auf die Funktion `myFunc()`. Es werden in manchen Fällen auch Zeilenbereiche anstelle eines Funktionsnames angegeben.

4.1 Konvertierung von Folien

Folien in Markdown müssen, wie bereits in Abschnitt 2.3.2 und 3.1 beschrieben, zu HTML-Quellcode konvertiert werden. Erst dann können sie in einem Browser als Präsentation angezeigt werden. Ebenso müssen sie zur korrekten Funktion von Cobra neben den Folieninhalten auch die Definitionen aller verwendeten Projekte, wie in Abschnitt 4.1.1 beschrieben, enthalten.

Als Konverter wird unter anderem aufgrund seiner Verbreitung und Unterstützung vieler Eingabe- und Ausgabeformate pandoc verwendet (vgl. 3.1.2). Es wird dabei von Markdown zu reveal.js (HTML) konvertiert. Es ist die Markdown-Syntaxerweiterung `fenced_code_attributes` aktiviert. Diese ist notwendig, um die in Abschnitt 2.3.2 beschriebene Syntax zur Definition von Klassen und Attributen an Codeblöcken zu ermöglichen. Die Erweiterung ist in der Standardinstallation von pandoc inkludiert.

Die Definition von Projekten kann entweder direkt im Markdown durch HTML-Objekte erfolgen oder in den Meta-Informationen eines Textes. Diese werden von pandoc im Header der Markdowndatei oder in einer separaten Datei gesucht (vgl. 2.3.2).

Das Format für die Angabe von Projekten in den Meta-Informationen erfolgt dabei in der in Listing 4.1 gezeigten Form. Aktuell durch den Cobra Server unterstützte Sprachen sind Scala und Haskell. Die nötigen Schritte zur Unterstützung weiterer Sprachen im Server sind in Abschnitt 4.4 beschrieben.

```
1 projects:
2   project-id-1:
3     root: <Dateipfad zu Projektordner>
4     language: <haskell | scala | java | ...>
5     src-roots: <orderpfad1, ordernerpfad2, ...>
6   project-id-2:
7     ...
```

Listing 4.1: Beipieldefinition von Projekten in YAML

Bei der Konvertierung von Markdown zu HTML wird durch Modifikation des Dokumenten-Syntaxbaums mittels eines pandoc Filters (vgl. 2.3.1) für jede ProjektId (in Listing 4.1 `project-id-1` und `project-id-2`) ein Objekt im HTML angelegt, das die angegebenen Werte der Attribute enthält. Im Listing sind für jedes Projekt die Attribute root und language Pflichtfelder und src-root ein optionales Feld. Der hier erwähnte pandoc Filter ist im Rahmen dieser Arbeit auf Basis der Bibliothek pandoc-types[1] in Haskell entwickelt worden [48, pandocProjectFilter]. In der Umsetzung wird er als kompilierte ausführbare Datei in die Cobra Releasedatei eingebunden. Bei Bedarf wird eine Kopie dieser Datei im Präsentationsverzeichnis abgelegt und mit Ausführungsrechten versehen. So muss der Filter nicht von Hand installiert werden. Aktuell ist der Filter nur unter Linux auf 64 Bit x86-Prozessoren ausführbar, da er noch nicht für andere Betriebssysteme und Prozessorarchitekturen kompiliert wurde. Um in Zukunft

[1] https://hackage.haskell.org/package/pandoc-types

eine Unterstützung vieler Betriebssystem-Hardware-Kombinationen zu ermöglichen, sollte für den Filter das Kompilieren in alle relevanten Betriebssystem-Plattform-Kombinationen unterstützt und automatisiert werden oder er sollte noch einmal in Lua implementiert werden. Ein Filter in Lua kann direkt durch den in pandoc enthaltenen Lua Interpreter ausgeführt werden. So müsste er nicht mehr in kompilierter Form Teil eines Cobra Releases sein. Bei einer Neuimplementierung sollte auch auf Nutzerfeedback geachtet werden. In der aktuellen Umsetzung führen fehlerhaft definierte Projekte zu keinerlei für den Nutzenden direkt ersichtlichen Problemen. Fehlerhafte Projektdefinitionen werden durch den Filter geschluckt und nicht im zu konvertierenden Dokument kenntlich gemacht. Im Rahmen der Implementierung sollte lediglich die Umsetzbarkeit der Dokumentenkonversation von Markdown zu HTML durch pandoc in Cobra demonstriert werden. In einer Umsetzung als Feature für Endnutzer*innen sollte Feedback für Eingabefehler jedoch nicht fehlen, da es erheblich zur Nutzbarkeit des implementierten Features beiträgt.

Eine neue HTML-Datei wird nur erzeugt, wenn entweder noch keine HTML-Datei aber eine Markdown-Datei existiert oder die Markdown-Datei einen neueren Änderungszeitstempel als die HTML Datei besitzt. Dieses Verhalten ist so gewählt, dass ein Autor in einem Präsentationsordner in beiden Sprachen schreiben kann, ohne dass er seine Sprachauswahl Cobra explizit mitteilen muss. Es wird nicht empfohlen während der Erstellung von Folien beide Dateien zu editieren, da Änderungen einer neueren HTML Datei nicht automatisch in eine Markdowndatei übernommen werden. Die Konvertierung wird durch die Cobra Server Main Funktion angestoßen [48, cobra-server Cobra].

Quellcodebeispiele, im folgenden auch Snippets genannt, wurden bewusst während der Konvertierung nicht in die generierten Folien inkludiert. Cobra lädt beim Präsentieren der Folien referenzierten Quellcode direkt aus dem Dateisystem und aktualisiert die angezeigten Beispiele automatisch bei Dateiänderungen. Würde referenzierter Quellcode schon bei der Folienkonvertierung in die Folien inkludiert werden, würde Cobra diesen Code während der Präsentation überschreiben. Wäre Quellcode während der Konvertierung inkludiert und gleichzeitig das `src` Attribut entfernt worden, würde dieses Verhalten unterbunden werden. Das automatische Aktualisieren von Beispielen bei Dateiänderungen wäre dann nicht mehr möglich.

4.1.1 Projektdefinitionen

Snippets, die über die Abfragesprache aus Abschnitt 3.2 definiert sind, sind nur im Kontext eines Projektes möglich. Ein Projekt besteht dabei mindestens aus einem Projekt-Root-Verzeichnis (im folgenden auch Projekt-Root genannt) und der Sprache des Projektes.

Diese Anforderung ist nötig, da für korrektes Highlighting und das Anzeigen von Compilerwarnungen unter Umständen der komplette Inhalt einer Datei, die über den im Beispiel inkludierten Code hinausgehen kann, Cobra bekannt sein muss. Korrektes Highlighting und Anbindungen an Compiler für Compilerinsights in Cobra zu integrieren ist nicht Teil dieser Arbeit, es ist jedoch gefordert

worden, diese Anforderung bei der Implementierung der Abfragesprache zu berücksichtigen.

Die Definition von Projekten in HTML wurde in nicht sichtbaren HTML `<div>` Objekten umgesetzt. Ein Definitions-Objekt enthält dabei die Attribute: data-key, data-language, data-root und data-srcRoots. Tabelle 4.1 gibt einen Überblick über Attribute und ihre Bedeutungen.

Attribute	**Pflichtfeld**	**Bedeutung**
`data-key`	*ja*	Identifier des Projektes. Dieser ist immer anzugeben, wenn die Abfrage von Snippets auf dieses Projekt beschränkt werden soll.
`data-language`	*ja*	Sprache des Projektes. Aktuell sind `scala` und `haskell` unterstützt.
`data-root`	*ja*	Oberverzeichnis des Projektes. Von dort aus wird das Projekt analysiert und sprachspezifische Pfadkonventionen[a] aufgelöst.
`data-srcRoots`	*nein*	Verzeichnisse, in denen der zu analysierende Quellcode liegt. Wird dieses Attribut leer gelassen, wird in allen Unterordnern von `data-root` nach Quellcodedateien gesucht. Dieses Attribut ist sehr hilfreich, wenn z.B. automatisch generierte Dateien von der Analyse ausgeschlossen werden sollen.

[a] z.B. der Pfad zur zur build.sbt Datei in Scala

Tabelle 4.1: Verfügbare Attribute zur Definition von Projekten in HTML

Der Typ und die Position des Definitionsobjektes im HTML sind frei wählbar. Alle Attribute müssen vorhanden, jedoch nicht mit Werten gefüllt sein.

4.2 Startup Sequenz des Clients

Die Aufgaben des (Web-)Clients, die beim initialen Aufrufen einer Präsentation ausgeführt werden, werden erweitert. In der Version von Cobra, die durch diese Arbeit erweitert wird, lädt der Client direkt aus dem Dateisystem des Hosts referenzierte Dateien. Danach löst er mittels magischer Kommentare (vgl. Abschnitt 2.1.2) definierte Beispiele auf, indem er den geladenen Code auf den durch den Kommentar definierten Bereich beschränkt. Im Anschluss wird der verbleibende Code der Beispiele als interaktive Dokumente beim Server angemeldet.

Es wurden die folgenden Erweiterungen/Änderungen vorgenommen:

1. Client sucht Projektdefinitionen und sendet diese an den Server [48, cobra-client Projects initProjects()]
2. Client fragt durch Dateisystempfade definierte Quellcodedateien vom Server an. Dieser lädt den Dateiinhalt aus dem Dateisystem [48, cobra-client Code loadDelayed()].

3. Client fragt über Abfragesprache definierte Quellcodeabschnitte vom Server an.

Wenn der Client die in Abschnitt 4.1.1 beschriebenen Projektdefinitionen an den Server sendet, baut der dort laufende Projektmaster [48, cobra-server project.ProjectMaster] aus den Definitionen Projektserver Aktoren als seine Kinder auf. Dabei analysiert jeder Projektserver [48, cobra-server project.ProjectServer] beim Starten nach dem in Abschnitt 4.4 beschriebenen Verfahren die Quellcodedateien eines Projektes. Er erzeugt aus den Analyseergebnissen eine Struktur zum Suchen von Snippets nach dem in Abschnitt 4.5.2 beschriebenen Verfahren. Das Analysieren von Projekten sowie Verwalten von Suchstrukturen im Client ist nicht praktikabel, da der Client keinen direkten Zugriff auf Quellcodedateien hat und Dateiinhalte immer über den Server anfragen muss.

Das Auflösen von Definitionen von Quellcodeabschnitten nach der in Abschnitt 3.2 beschriebenen Abfragesprache, kann durch das Parsen der Abfragen rechenintensiv sein. Daher wird es als Aufgabe dem Server übergeben. Aus diesem Grund und da, wie bereits beschrieben, der Server alle Snippets aus Projekten verwaltet, ist es also nötig, dass der Client Snippets vom Server abfragt. Um die Kommunikation für alle Einbindungsarten von Beispielen gleich zu halten, werden in der Implementierung alle Einbindungsarten über websocket-basierte Serveranfragen umgesetzt. Der Server antwortet für alle Quellcode-Anfragen mit dem Inhalt des Abschnittes. Dieser wird vom Client an der entsprechenden Stelle in das HTML eingefügt.



(a) Originalzustand



(b) nach Änderungen

Abbildung 4.1: Interaktionen zwischen Client und Server vor/nach Änderungen

In Abbildung 4.1 sind die Interaktionen zwischen Client, Server und dem Dateisystem im Vorher- und Nachher-Zustand grafisch dargestellt. Dabei beschränkt sich die Darstellung auf Interaktionen im Kontext des initialen Aufrufs der Fo-

lie. Die Initialisierung beginnt mit HTTP Requests und wechselt im späteren Verlauf auf einen Websocket. Die im Websocket ablaufende Kommunikation ist in den Diagrammen grau hinterlegt.

Interaktionen, wie „Load Snippets“, sind als eine Anfrage und eine Antwort vereinfacht dargestellt. Für jedes Snippet wird eine Anfrage gestellt und eine Antwort gesendet. Dies ist der Leserlichkeit wegen vereinfacht worden. Gleiches gilt für „Init Projects“, „Get Snippets“ sowie „Init Docs“ und ihre jeweiligen Antworten. In blau markiert sind in Abbildung 4.1 (b) die Bereiche, in denen sich die Interaktion zwischen dem Client und dem Server verändert hat. Abschnitte 4.4 und 4.5.2 gehen genauer auf die Umsetzung der Punkte „Analyse Projects“ und „Resolve Snippets“ ein.

Wie in Abbildung 4.1 auch zu sehen ist, werden Snippets, nachdem sie beim Client ankommen und in das HTML eingefügt werden, wie im Originalzustand von Cobra behandelt. Sie werden mit ihrem kompletten Inhalt als interaktives Dokument beim Server angemeldet („InitDocs“). Der Server verlässt sich dabei auf den vom Client gesendeten Content und lädt den Inhalt nicht noch einmal aus dem Dateisystem. Dies führt dazu, dass der Inhalt eines Snippets mehrfach übertragen wird. Dieses Verhalten wurde gewählt, um die Änderungen an der Interaktion zwischen Server und Client möglichst nur auf das Laden der Snippets zu beschränken. Eine Änderung der Verwaltung von interaktiven Dokumenten im Server hätte noch weitere Änderungen nach sich gezogen, die außerhalb des Fokus dieser Arbeit liegen. Durch die doppelte Übertragung der Snippets und erhöhten Kommunikationsaufwand zwischen Server und Client ist lediglich das initiale Laden der Folien betroffen. Die Performance während der Präsentation wird nicht eingeschränkt.

4.2.1 Asynchrone Kommunikation zwischen Client und Server

Das Absenden von Anfragen an den Server im Client ist eine asynchrone Funktion. Um eine Korrelation zwischen Anfragen und ihren Antworten für den Client zu ermöglichen, werden alle Projekt- und Snippetanfragen mit einer eindeutigen ID versehen. An dieser ID wird im Client zum Zeitpunkt des Versendens der Serveranfrage eine anonyme Funktion hinterlegt [48, cobra-client Projects Zeile 44ff].

Wenn der Client eine Serverantwort mit ID empfängt, wird in einer synchronisierten Map nach einem möglichen Callback für die Antwort-ID gesucht. Sollte ein Callback vorhanden sein, wird es ausgeführt. [48, cobra-client CobraJS Zeilen 65 - 82]. Reaktionen den Clients auf Serverantworten werden in diesen Callback implementiert.

4.3 Projektverwaltung im Server

Projekte im Cobra Server werden in einer Reihe von Projektservern (ein Server pro Projekt) von einem serverweiten Projektmaster verwaltet. Die gesamte Verwaltung der Projekte findet in Aktoren statt, welche in Abschnitt 2.1.3 beschrieben wurden. Der Projektmaster nimmt dabei alle Requests bzgl. der Beschaffung von Snippets und Anmeldung von Projekten entgegen. Er erzeugt Projektserver, wenn diese über eine Projektinitialisierung bei ihm angemeldet werden und verwaltet den Satz aller Projektserver über ihren Lebenszyklus hinweg. Der Projektmaster leitet Nachrichten, welche direkt für Projektserver bestimmt sind, an die korrekten Server weiter.

Ebenso erzeugt der Projektmaster neue One-Of Aktoren für das Laden von Quellcode aus dem Dateisystem und die Suche nach Quellcode basierend auf Abfragen. Diese beiden Aktoren werden genauer in Abschnitt 4.5.1 und 4.5.2 erläutert. One-Of Aktoren beschreiben in dieser Arbeit kurzlebige Aktoren, die für die Lösung einer Aufgabe erzeugt werden. Meistens enthalten sie alle Informationen zur Durchführung ihrer Aufgabe durch den Objektkonstruktor. Nach dem Erfüllen ihrer Aufgabe beenden sie sich selbst.

Da der Projektmaster die Nachrichtenverteilung an alle Projektserver übernimmt, sind seine Aufgaben auf Verwaltungsaufgaben ohne großen Rechenaufwand und Bedarf an Lese- und Schreibvorgängen auf das Dateisystem begrenzt. So soll vermieden werden, dass er zur Engstelle bei der Beantwortung von Anfragen wird oder durch IO-Fehler zum Absturz gebracht wird.

Der Projektserver verwaltet alle Informationen zu einem Projekt, welche benötigt werden, um Anfragen nach Snippets aus diesem Projekt zu beantworten. In seinem Lebenszyklus sind die folgenden Aufgaben untergebracht: Während der Erzeugung der Aktor-Properties, von Akka als Props abgekürzt, findet eine grundlegende Überprüfung der Parameter statt. Dies passiert noch vor dem Start des Aktors, damit bei der Erzeugung des Projektservers mit invaliden Eingaben direkt eine Fehlermeldung für den Client erzeugt werden kann. Im Pre-Start Zustand findet die Analyse des Projektes statt. Aus den Ergebnissen wird die Suchstruktur für dieses Projekt erstellt. Die Details der Projektanalyse und der Aufbau der Suchstruktur sind in Abschnitt 4.4 beschrieben. Nachdem die Analyse des Projektes abgeschlossen ist, werden die Ordner des Projektes im Hintergrund auf Dateiänderungen überwacht. Näheres zum Verhalten des Projektservers bei Dateiänderungen findet sich in Abschnitt 4.3.1. Nach der Einrichtung der Änderungsüberwachung geht der Aktor in den operativen Zustand über und beginnt Anfragen nach Snippets zu beantworten.

4.3.1 Verhalten bei Dateiänderungen

Während der Vorbereitung einer Präsentation oder ihrer Vorstellung kann es zu Änderungen der Quellcodedateien in einem definierten Projekt kommen. Eine Änderung in Quellcode kann zu Änderungen in den Pfaden führen, welche Snippets definieren. In Cobra führt eine Dateiänderung in einer eingebundenen Datei zu einem Invalidieren aller Snippets im Client, welcher daraufhin den Inhalt jedes Beispiels erneut vom Server abfragt. Eine beobachtete Dateiänderung

löst außerdem eine Nachricht an den Projektserver aus, der die entsprechende Datei verwaltet [48, cobra-server project.ProjectServer Zeilen 130 - 140].

Je nach Art der Dateiänderung wird einer der folgenden Mechanismen ausgelöst:

- Das **Löschen einer Datei** führt dazu, dass alle Snippets, die in ihrer Quelle auf diese Datei verweisen, aus der Suchstruktur entfernt werden. Codeblöcke, die sich in ihren Pfaden auf Snippet beziehen, die nun nicht mehr vorhanden sind, laufen ins Leere und bekommen null Treffer zurück.
- Das **Ändern einer Datei** führt zu der Entfernung aller Snippets aus dem Projektserver, welche die geänderte Datei als Quelle angeben. Danach wird die geänderte Datei analysiert und alle Ergebnisse wieder in die Suchstruktur der Projektserver eingefügt.
- Das **Hinzufügen einer Datei** führt zur Analyse dieser. Alle Ergebnisse der Analyse werden in die Suchstruktur eingefügt.

Während auf die Erstellung, Entfernung oder Änderung einer Datei reagiert wird, werden vom Projektserver keine Anfragen beantwortet. An ihn versendete Anfragen sammeln sich in der Mailbox des Aktors und werden erst nach Abarbeitung der entsprechenden oben beschriebenen Mechanismen beantwortet.

4.4 Projektanalyse

Bei der Analyse eines Projektes müssen aus allen Quelltextdateien, die in ihrer Gesamtheit das Projekt bilden, Informationen gewonnen werden. Mit Hilfe dieser Informationen können dann Anfragen nach Snippets beantwortet werden. Die zu gewinnenden Informationen beinhalten Definitionen von Projektstrukturen, wie Pakete und Module, sowie die Positionen und Identifier von Definitionen und ihre strukturelle Schachtelung innerhalb einer Datei. Je nach Sprache müssen noch weitere Elemente untersucht werden, um alle Definitionen zu finden. Ein Beispiel dafür sind Präprozessoranweisungen.

Um die oben erwähnten Informationen aus einem Projekt zu extrahieren, braucht es mindestens einen Parser für Quellcodedateien. Dieser kann einen abstrakten Syntaxbaum einer Datei aufbauen. Aus diesem Syntaxbaum kann dann, wie in Abschnitt 3.2.2 beschrieben, der vereinfachte Syntaxbaum (vgl. Abbildung 3.1a und Abbildung 3.1b) gebildet werden. Ebenso muss die Struktur des Projektes modelliert werden, um Paket- und Modulzugehörigkeiten korrekt aufzulösen, sollte dies nicht aus der Syntax hervorgehen. Je nach Komplexität der Sprache bedarf es noch weiterer Funktionalitäten und Module.

Es ist möglich alle, zur Informationesextraktion benötigten, Komponenten selbst zu entwickeln oder als Abhängigkeiten in das Projekt zu integrieren. Der Aufwand ist hoch, da für jede Sprache mindestens ein Parser integriert oder entwickelt werden muss. Die Unterstützung von zwei Sprachen erfordert, neben der Integration der Parser, auch das Aktualisieren eben jener Parser auf neue Versionen und Erweiterungen, um mit Sprachentwicklungen aktuell zu bleiben.

Statt der Integration von verfügbaren Parsern wurde ein anderer Weg gewählt. In Cobra wurde im Modul `modules/cobra-lspbinding` eine Bibliothek ent-

wickelt, welche die Analyse von Projekten mit Hilfe des LSP und einem Language Server pro Sprache ermöglicht. Das Modul tritt dabei im Kontext des LSP als Client auf. Der Umsetzung liegt die Java Bibliothek lsp4j[2] zugrunde. Durch die Nutzung des LSP muss kein Support mehr pro Sprache implementiert und gepflegt werden, sondern nur die einmalige Anbindung an das LSP.

Projektanalyse durch Language Server: Der Language Server wird zu Beginn der Analyse im Root Verzeichnis des Projektes gestartet. Von dort aus hat er Zugriff auf Dateien und Pfade mit festen Konventionen über ihre Positionierung in einem Projektordner Baum. Nach seinem Start werden alle Quellcodedateien von der entwickelten Bibliothek gesucht und einzeln mit Hilfe des LSP's analysiert. Die genaue Implementierung ist in Abschnitt 4.4.1 beschrieben. Aus den Ergebnissen der Dateianalyse wird eine Liste von Snippets erzeugt. Diese enthalten jeweils den Identifier, den Pfad des Eltern-Snippets, den Dateisystempfad zur Datei, die analysiert wurde und die Start- sowie Endzeile des Definitionsabschnitts in der Quellcodedatei. Abschnitt 4.4.2 beschreibt, wie diese Snippets aus den Ergebnissen des LSP aufgebaut werden. Das Ergebnis der Analyse ist eine Liste von Snippets, welche in einem Projekt extrahiert werden konnten. Auf Basis dieser Liste wird die Suche von Snippets, wie in Abschnitt 4.5.2 beschrieben, implementiert. Der Language Server wird während der kompletten Laufzeit des Projektservers im Hintergrund am Laufen gehalten, um schnelle Analysen bei Dateiänderungen zu ermöglichen, ohne auf eine Neuinitialisierung des Servers warten zu müssen.

Unterstützung weiterer Sprachen: Die Nutzung des LSP vereinfacht auch die Unterstützung weiterer Sprachen. Es gibt laut Microsoft bereits Language Server für 91 Sprachen und Formate.[49]

Um eine weitere Sprache durch das LSP in Cobra zu unterstützen, muss ein Markerobjekt angelegt werden. Es wird unter anderem bei der Kommunikation zwischen Client und Server versendet, um die Sprache eines Projektes oder Snippets anzugeben. Für die Sprache muss ein Standard Language Servers in der Standardkonfiguration angelegt werden und ein Branch zum Starten des korrekten Language Servers eingetragen werden. Je nach Server muss außerdem noch die Logik zur Normalisierung von Pfaden in den Ausgabe des neuen Language Servers implementiert werden. Die Normalisierung von Pfaden wird in Abschnitt 4.4.2 erläutert.

4.4.1 Genutze Funktionalität des LSP

Bei der Analyse von Dateien wird die Funktionalität des „Document Symbol Requests“ (DSR) verwendet. Der Request wird für eine Datei von einem Client an den Language Server versandt und von ihm mit allen Symbolen in diesem Dokument beantwortet [50].

Zuerst wird die Datei für den Language Server mit einer „DidOpenTextDocument Notification“ als geöffnet markiert. In der Notifikation ist neben dem Pfad

[2] https://eclipse.org/lsp4j

zur Datei auch der Inhalt der Datei zum Zeitpunkt des Öffnens enthalten. Die Beschreibung der Notifikation aus der LSP-Spezifikation erläutert, dass diese Notifikation dem Server symbolisiert, dass der Inhalt der Datei vom Client verwaltet wird und nicht mehr mit garantierter Richtigkeit über ein Aufrufen der Datei URI zu erhalten ist. Obwohl anhand dieser Beschreibung angenommen werden kann, dass dieser Schritt nicht notwendig ist, hat es sich beim Testen mancher Language Server als Notwendigkeit herausgestellt, zu analysierende Dateien zuerst über diese Notifikation beim Server anzumelden. Document Symbol Requests werden von den betreffenden LS Implementierungen in manchen Fällen nicht beantwortet, wenn dieser Schritt übersprungen wird[3].

Danach wird der Document Symbol Request ausgeführt und die Ergebnisse, wie in Abschnitt 4.4.2 beschrieben, in Snippet-Objekte umgewandelt. Zuletzt wird die nun analysierte Datei mit der „DidCloseTextDocument Notification“ geschlossen. Dies ist zum einen nötig, damit eine weitere Analyse der Datei, z.B. nach beobachteten Dateiänderungen, durchgeführt werden kann. Zum anderen wird so dem Language Server die Möglichkeit gegeben, Speicher, den er für den Inhalt der Datei belegt hat, freizugeben. [48, cobra-lsbinding languageserver.LSInteraction analyzeFile()]

4.4.2 Datenmapping

Das Ziel des Datenmappings ist die Erzeugung von Snippet-Objekten aus den Ergebnissen von Analysen. Ein Snippet-Objekt kapselt dabei immer den Quellcode einer Definition und enthält die folgenden Informationen:

- Name der Definition
- Typ der Definition
- Pfad bis zur Definition
- Dateisystempfad zur Quellcodedatei
- Start- und Endzeile

Das Ergebnis der Analyse einer Datei ist entweder eine flache Liste oder ein hierachischer Baum aller in der Datei definierten Symbole. Es ist laut der Spezifikation des LSP dem Implementierenden freigestellt, entweder den Baum bestehend aus `DocumentSymbol` Objekten oder eine flache Liste von `SymbolInformation` Objekten zurückzugeben. Beide Objekte haben gemeinsam, dass sie ein `name` Feld besitzen, in dem der Name des Symbols steht, und ein `kind` Feld, welcher den Typen des Symbols enthält. Das LSP versteht 26 verschiedene Typen von Symbolen von Dateien (File), über Methoden und Funktion bis zu Structs und EnumMember. Ebenso enthalten beide `Location` Informationen, die Start- und Endpositionen mit Zeilennummern enthalten.

Die Felder `name`, `kind` und `location` können direkt aus dem Ergebnisobjekt in das Snippet übernommen werden. Die Datei, welche analysiert wurde, um das Ergebnisobjekt zu erhalten, ist ebenfalls bekannt und kann direkt in das Snippet

[3]Metals als Language Server für Scala funktioniert mit und ohne DidOpen Notification, Haskel-IDE-Engine (HIE) erwartet sie und gibt sonst keine Analyse-Ergebnisse aus.

übernommen werden. Lediglich der Pfad bis zur Definition muss für jede Art von Ergebnisobjekt einzeln erzeugt werden.

Für Symbolbäume, bestehend aus `DocumentSymbol` Objekten, werden alle Knoten nach dem Visitor Pattern [51, S. 331] durchlaufen und für jeden Knoten im Baum der Pfad von der Wurzel aus gespeichert.

Für flache Listen aus `SymbolInformation` wird das Feld `container` genutzt, welches den Namen inklusive Pfad des Symbols enthält. Die genaue Formatierung des Container Pfades ist dabei abhängig vom Language Server und wird deshalb vereinfacht, indem Trennzeichen wie '/' und '#' durch '.' ersetzt werden. Die angeführten Trennzeichen sind bei Tests der Language Server als Trennzeichen identifiziert worden. Für jeden weiteren integrierten Language Server müssen die Trennzeichen unter Umständen aktualisiert werden. Für die Erzeugung des Pfades im Snippet wird der normalisierte Inhalt des `container` Feldes mit dem `name` Feld verbunden und genutzt [48, cobra-lsbinding Snippet].

Container als valider Pfad: Die LSP Spezifikation gibt an, dass aus `Symbol Information` Objekten weder die Location des Symbols noch die Information über den Container genutzt werden kann, um die Struktur des Dokumentes zu inferieren. [50, SymbolInformation]

In dieser Arbeit wird jedoch genau dies getan. In Tests mit einfachen und komplexeren Dateien in den Language Servern Metals[4] für Scala und HIE[5] für Haskell haben sich die aus Container und Name entstehenden Pfade als korrekt erwiesen. Sie bilden alle Definitionen von Variablen, Funktionen, Klassen, Objekten und Typen auch in geschachtelten Konstrukten korrekt ab. Es ist anzunehmen, dass Container in Randfällen nicht zu 100 Prozent korrekt sind und z.B. nicht für die Verifikation von Programmen genutzt werden können. Für die Definition von Beispielen in Präsentationen werden sie jedoch als ausreichend erachtet.

Das hier verwendete Verhalten wird nicht durch das LSP garantiert. Vor der Unterstützung jedes weiteren Language Servers, sowohl für bereits unterstützte als auch neue Sprachen, muss geprüft werden, ob dieser für `SymbolInformation` Objekte nutzbare Pfade erzeugt. Gegebenenfalls müssen auch, wie im vorherigen Abschnitt beschrieben, Funktionalitäten entwickelt werden, um abweichende Pfade in eine normalisierte Form zu überführen.

Sollte eine Prüfung und Normalisierung der Pfade für neue Language Server nicht vorgenommen werden, kann es zu dem folgenden Szenario kommen: Zwei Nutzer*innen (A & B) arbeiten gemeinsam an den jeweils eigenen Rechnern an einer Präsentation. Nutzer*in A hat Language Server 1 (LS_1) für Sprache S installiert und Nutzer*in 2 Language Server 2 (LS_2) für eben jene Sprache S. Sollte LS_1 oder LS_2 nicht normalisierte Pfade erzeugen, können A und B unterschiedliche Pfade angeben, um dasselbe Snippet in einer Folie zu referenzieren. Es wird zur Laufzeit auf dem Rechner des jeweils anderen nicht angezeigt werden.

[4] https://scalameta.org/metals/
[5] https://github.com/haskell/haskell-ide-engine

4.5 Suchen und Finden von Snippets

Clientanfragen nach Snippets werden im Server auf unterschiedliche Weise bearbeitet. Es wird nach Anfragen über einen Dateisystempfad und einem Pfad über die Struktur des Quellcodes unterschieden. Im Folgenden werden die Methoden zur Beantwortung dieser beiden Anfragetypen und der projektübergreifenden Suche vorgestellt.

4.5.1 Snippets durch Dateipfade

Im Zuge der Implementierung von Pfaden zur Auswahl von Snippets in Cobra muss, wie bereits in Abschnitt 4.4 erwähnt, die Verantwortung des Ladens von Snippets aus dem Client in den Server verlegt werden. Anfragen nach Snippets, unabhängig ihrer Art, werden an den Projektmaster gesendet. Im Master wird dann entschieden, ob es sich um eine Suchanfrage oder die Anforderung eines Datei-Inhaltes handelt. In beiden Fällen wird zur Beantwortung einer Anfrage ein „Per Session Child Actor“ erzeugt [52]. Mit Hilfe dieses Patterns wird die Bearbeitungszeit des Projektmasters so kurz wie möglich gehalten. Dieser ist singlethreaded und routet sämtliche Nachrichten an den Projektserver.

Für Anfragen von Dateipfaden wird ein Aktor gestartet, welcher diese Datei lädt und den Datei-Inhalt an den originalen Absender der Anfrage sendet. Die zwischen den Aktoren ablaufende Kommunikation ist in Abbildung 4.2 abgebildet. Module sind als Quadrate abgebildet und Aktoren als Ellipsen. Kommunikation ist durch beschriftete Pfeile dargestellt und Aktoraktionen, wie das Erzeugen oder Stoppen eines Aktors, sind durch gepunktete Pfeile kenntlich gemacht. Potenzielle Beziehungen zwischen Aktoren werden auf gleiche Weise dargestellt.



Abbildung 4.2: Aktorkommunikation beim Laden einer Quellcodedatei

Die Tätigkeit des Quellcodeladens in einem eigenen Aktor auszuführen ist rein funktional nicht notwendig. Sie ist trotzdem implementiert worden, um den potenziell lang dauernden Dateisystemzugriff in einen eigenen Thread zu verschieben und nicht blockierend im Thread des Projektmasters auszuführen. Dieser könnte sonst, bis die IO-Operation abgeschlossen ist, keine weiteren Anfragen bearbeiten. Außerdem schützt die Kapselung des Dateizugriffs in einem eigenen Aktor den Projektmaster vor unerwarteten IO-Fehlern und es kann durch die Trennung der Funktionalitäten garantiert werden, dass der Projektmaster nicht durch einen solchen Fehler gestoppt wird [48, cobra-server project.SourceLoadActor].

4.5.2 Snippets durch strukturelle Pfade

Anfragen nach Snippets über die in Abschnitt 3.2 vorgestellte Abfragesprache werden durch den Projektmaster entgegengenommen. Dieser erzeugt für jede Anfrage einen eigenen Per-Session-Aktor namens Search Aktor [48, cobra-server project.SearchActor].

Der Search Aktor parst den Pfad, um ihn in die Pfadkomponenten und potenziell definierten Keys (vgl. 3.2.3) zu zerlegen. Beim Parsen werden unbekannte Keys überlesen und am Ende die Liste der Keys auf eindeutige Keys reduziert. Mehrfach vorkommende Keys werden in Konformität mit der in Abschnitt 3.2.5 beschriebenen Semantik behandelt. Die Umsetzung erfolgt in der Klasse [48, cobra-server paths.PathParser] auf Basis von Parser Combinators aus der fastparse[6] Bibliothek.

Die Aufgabe des Search Aktors ist es, alle Projektserver nach Snippets unter dem gegebenen Pfad zu befragen. Ist das zu durchsuchende Projekt durch die ProjektId fest definiert, wird nur das definierte Projekt befragt. Ist kein Projekt definiert, werden alle Projektserver befragt. Vom Search Aktor werden alle Antworten der Projektserver gesammelt und zusammengefasst an den Autor der Snippet Anfrage gesendet. In Abbildung 4.3 ist die Kommunikation für eine Suchanfrage in einem Beispiel mit zwei zu befragenden Projektservern dargestellt worden. Die Darstellungskonventionen folgen denen in Abschnitt 4.5.1 beschriebenen.



Abbildung 4.3: Aktorkommunikation beim Suchen von Snippets

Durch die Verwaltung von Projekten in einzelnen Projekt Aktoren ist es nötig, eine Suchanfrage nach einem Snippet an potenziell mehrere relevante Projektserver weiterzuleiten. Das Zusammenfassen der Antworten der Projekte zu einer gesammelten Antwort ist logisch nicht zwingend nötig, erleichtert jedoch die Logik im Cobra Client, da pro Anfrage nur auf eine Antwort gewartet werden muss (vgl. 4.2.1). Hat der Search Aktor von jedem befragten Projektserver eine Antwort erhalten, fasst er die Antworten zusammen, sendet sie an den Client und beendet sich selbst.

[6] http://www.lihaoyi.com/fastparse/

Der Search Aktor arbeitet mit einem Timeout, der garantiert, dass er die Suche nach einer festgelegten Zeitspanne beendet, unabhängig davon, ob er zu diesem Zeitpunkt schon Antworten von jedem befragten Projekt erhalten hat. Dieser Timeout garantiert, dass der Search Aktor beendet wird und somit seinen Speicher freigibt. Außerdem garantiert er, dass eine Antwort auf die Suchanfrage an den Client versandt wird, auch wenn nicht alle Projektserver eine Antwort an den Search Aktor gesendet haben. Die Antwort an den Client beinhaltet in jedem Fall die Anzahl der Projektserver, deren Antworten zum Zeitpunkt der Zusammenfassung nicht vorlagen. Eine oder mehrere ausstehende Projektantworten signalisieren an den Client, dass die erhaltenen Suchergebnisse unter Umständen nicht vollständig sind und er diese nur mit einer entsprechenden Warnung verarbeiten sollte.

Eine Implementierung des Search Aktors ohne einen Timeout würde in der aktuellen Implementierung eine endlose Warteschleife auslösen, da die Antwort auf eine Suchanfrage nicht garantiert ist und würde Timeouts oder andere Fehlerlogik im Client erforderlich machen.

4.5.3 Suchen von Snippets

Die eigentliche Suche, das Auflösen von Pfaden zu Snippets, ist im Trait SnippetSearch [48, cobra-server paths.SnippetSearch] implementiert. Die Suche wird durch die Projektserver ausgeführt, da diese beim Starten als Ergebnis der Projekt Analyse eine Liste von Snippets erzeugen und speichern.

Aus der Liste der Snippets wird zum Suchen kein Baum erzeugt. Stattdessen ist die Suche auf einer unsortierten Liste von Snippets implementiert.

Zur Suche wird der Pfad eines Knotens von der Wurzel bis zum Knoten als Liste von Pfadelementen abgebildet. Auf dieser Liste können die beiden Suchrichtungen (von der Wurzel zum Knoten und vom Knoten zur Wurzel, vgl. Abschnitt 3.2.5) wie folgt dargestellt werden:

- Ein Knoten wird von einem Pfad in Richtung Knoten beschrieben, wenn die Liste seiner Pfadelemente in Inhalt und Reihenfolge gleich der Liste des gesuchten Pfades ist.
- Ein Knoten wird von einem Pfad in Richtung Wurzel beschrieben, wenn die Liste der Elemente des gesuchten Pfades ein Suffix seiner Pfadliste darstellt.

Die Implementierung der Suche auf einer Liste von Snippets entspricht einer Filteroperation auf dieser Liste, bei der für jedes Snippet geprüft wird, ob mindestens eine der oben genannten Bedingungen für den im Snippet angegebenen Pfad erfüllt ist.

Während der Ausführung von Cobra werden Snippets nur einmalig beim Laden und Initialisieren der Folien angefragt (vgl. 4.1). Die Suche nach Snippets im Kontext von Clientanfragen stellt keine Aufgabe dar, die während der Laufzeit des Programms häufig auftritt und besonders schnell oder effizient ablaufen muss, um die Nutzbarkeit des Programms nicht zu beeinflussen. Projektserver suchen außerdem nur in der Liste von Snippets, welche aus ihrem Projekt

stammen. Die Anzahl der zu durchsuchenden Elemente pro Ausführung des Suchalgorithmus ist deshalb nicht überwältigend groß. Durch die parallele Suche in mehreren Projekten, wie in Abschnitt 4.5.2 beschrieben, ist dieser simple Suchalgorithmus in Tests auch auf älterer Hardware schnell genug, um eine flüssige Nutzung von Cobra zu ermöglichen und nimmt im Vergleich zur Projektanalyse sehr wenig Zeit in Anspruch. Eine performantere Implementierung auf Basis von Präfixbäumen ist denkbar, sollte die Suche zu einem Performanceproblem werden.

Kapitel 5

Evaluation

Es wird ein Nutzertest durchgeführt, um Markdown und HTML als Sprachen zum Schreiben von Folien zu vergleichen und die entwickelte Abfragesprache zu testen. Im Folgenden werden der Aufbau des Nutzertests vorgestellt und die Ergebnisse ausgewertet.

5.1 Ziel der Evaluation

Im Kontext von Markdown als Foliensprache und der Definition von Quellcodebeispielen durch Pfade soll die Nutzbarkeit dieser Konzepte evaluiert werden. In der Evaluation sollen die folgenden Fragen beantwortet werden:

1. **Ist durch die Nutzung von Markdown eine Verbesserung der Nutzbarkeit gegenüber HTML erreicht worden?** Diese Frage wird betrachtet unter den Gesichtspunkten Verständnis der Folieninhalte im Quelltext, Verständnis der Struktur von Folien und Foliensätzen sowie Effizienz, Effektivität und Zufriedenheit beim Verständnis und der Erstellung von Folien.
2. **Sind Pfade als Metapher zur Auswahl von Beispielen geeignet?** Diese Frage wird unter den Gesichtspunkten Verständnis der Bedeutung und Erstellung von Pfaden betrachtet und unter den Punkten Effektivität und Nutzerzufriedenheit ausgewertet.

Die Evaluation wird am Ende der im Rahmen dieser Arbeit erfolgten Umsetzung der Konzepte durchgeführt. Die oben aufgestellten Fragen sind für eine Weiterentwicklung der Konzepte zu vollendeten Features in Cobra von Relevanz. Im Kontext dieser Arbeit erfüllt die Evaluation das Ziel einen Einblick in die Nutzbarkeit der entwickelten Konzepte zu geben. Im Rahmen der Entwicklung von Cobra ist sie ein Wegweiser, ob und in welcher Form die Konzepte nutzbare Features darstellen und in nachfolgenden Entwicklungsaufwänden zu vollständigen Features weiterentwickelt werden sollten.

Wegen der Durchführung der Evaluation im Entwicklungsprozess wurde entschieden, einen formativen Usability Test mit einer begrenzten Anzahl (n=12)

von Teilnehmer*innen durchzuführen. Bedingt durch die begrenzte Zahl an Teilnehmenden wird auf eine statistische Auswertung verzichtet, dadurch bedingt können aus dieser Evaluation keine allgemeingültige Aussagen über die Nutzbarkeit von Markdown oder HTML als Sprachen zum Ausdruck von Folien aufgestellt werden. Als Ergebnis der Evaluation sollen aktuelle Probleme in der Nutzbarkeit offengelegt werden, welche in weiteren Entwicklungsschritten bearbeitet werden sollten.

5.2 Aufbau des Nutzertests

Im Folgenden werden die Aufgaben des Nutzertests vorgestellt. Neben den Materialien, Aufgabenstellungen und Fragen an Nutzer*innen wird auf die Testumgebung und Kriterien zur Nutzerauswahl eingegangen.

5.2.1 Testumgebung

Der Usability Test wird in einer kontrollierten Umgebung mit einem Teilnehmer zur Zeit durchgeführt. Es werden vor Ort leere Büros und Konferenzräume genutzt, um jedem Teilnehmer eine ruhige Arbeitsumgebung ohne weitere Personen zu bieten. Dies soll es den Teilnehmern zum einen erleichtern, in einem geschützten Raum und zum anderen ohne äußere Ablenkungen, die im Test gestellten Aufgaben zu bewältigen. Außerdem stellen ruhige Büros oder büroähnliche Räume Orte dar, in denen das Erstellen von Folien für Cobra Präsentationen im realen Einsatz vorstellbar ist.

Alle Teilnehmenden müssen zum Erstellen von Folien Papier und Stifte nutzen. Dies entspricht keinem realen Einsatzszenario. Es bietet jedoch für alle Nutzer*innen die gleichen Bedingungen. Keinem Teilnehmenden wird so durch Vorkenntnisse Vorteile verschafft, sollten die Person ausgewählte Tastaturlayouts, Betriebssystem- oder Editorshortcuts kennen. In Abschnitt 3.1 wird als Bedingung für die Eingabesprache erwähnt, dass diese es einem Autor ermöglichen soll, einfache Folien in einem einfachen Editor zu schreiben. Als einfacher Editor wird hier ein Programm wie `vi` oder `notepad` angenommen, welches im schlimmsten Fall kein Syntaxhighlighting oder Syntaxchecks anbietet. Die Nutzung von Papier ist in diesem Kontext adäquater Ersatz für ein solches Programm. Ein weiterer Vorteil von Papier ist, dass alle Änderungen und Korrekturen Teilnehmender erkennbar sind, da ausschließlich Kugelschreiber als Stifte verwendet werden.

Während der Tests ist neben dem Teilnehmenden immer ein Moderator anwesend. Er stellt die Aufgaben vor und stellt Fragen. Gleichzeitig protokolliert er das Verhalten des Teilnehmenden stichpunktartig und stellt bei Bedarf weiterführende Fragen. Die Fragen, die er stellt, sind in Abschnitt 5.2.2 beschrieben. Teilnehmende werden ermutigt, während der Durchführung dem Think-Aloud Prozess [53, S. 221 - 224] folgend, ihre Gedanken, Probleme und Erkenntnisse mitzuteilen, damit diese ebenfalls erfasst werden können. Sollten Teilnehmer an einer Stelle offensichtlich hängen, jedoch nichts mitteilen, werden sie auch aktiv auf ihren aktuellen Gedankenprozess angesprochen. Es ist durch das Testsetup

nicht möglich vor dem Teilnehmenden zu verbergen, dass Notizen vom Moderator aufgenommen werden. Deshalb wird die teilnehmende Person vor Beginn des Tests über diese Tatsache informiert. Zusätzlich zu den Beobachtungen des Moderators werden außerdem für manche Aufgaben die Zeiten zum Lösen, Versuche und Fehler sowie die Art der Fehler erfasst.

5.2.2 Praktische Aufgaben und Fragen

Um die in Abschnitt 5.1 aufgestellten Fragen zu beantworten, werden alle Teilnehmenden gebeten eine Reihe von praktischen Aufgaben zu bearbeiten. Die gestellten Aufgaben sind an möglichen Tätigkeiten orientiert, die bei der Nutzung von Cobra und der Auswahl von Beispielen anfallen können. Während der Aufgaben werden einige Messwerte, wie Zeit zur Bearbeitung, Anzahl an Versuchen, sowie Menge und Art der Fehler erhoben. Zwischen den Aufgaben werden Teilnehmenden einige Fragen gestellt, die sie in freier Form beantworten können. Diese Fragen nehmen die Rolle eines klassischen Fragebogens ein und unterstützen die Beobachtung der Teilnehmenden während ihrer Arbeit, um ein besseres Bild über die Zufriedenheit der Teilnehmenden zu erlangen (vgl. [54, S. 10]).

Im Folgenden wird immer eine praktische Aufgabe mit ihren dazugehörigen Fragen und ihrem Zusammenhang zu den aufgestellten Fragen erläutert. Alle in der Evaluation verwendeten Materialien sind in Anhang A zu finden. Alle Materialien, wie Bilder und Quelltexte, werden den Teilnehmenden immer passend zur Aufgabe in ausgedruckter Form vorgelegt.

Wenn nicht anders angegeben, werden bei allen praktischen Aufgaben die Messwerte Anzahl der Fehler und Zeit zum Lösen der Aufgabe in Anlehnung an die in ISO25022[54, S. 11] definierten Messwerte Errrors in Task und Task Time erhoben. Zusätzlich werden Teilnehmende nach ihrer Zufriedenheit mit dem in der Aufgabe genutzten Konzept befragt in Anlehnung an den ISO25022 Messwert Overall Satisfaction [54, S. 14]. Die Zufriedenheit wird anhand einer vereinfachten Single Ease Question [55] mündlich erhoben. In den Beobachtungen wird ebenfalls ein Fokus auf Beschwerden von Nutzern gelegt. Bedingt durch die geringe Anzahl von Teilnehmenden wird keine statistikbasierende Messgröße, wie Proportion of Users Complaining [54, S. 15], berechnet.

Vor allem bei einer Antwort in der Single Ease Question zur Nutzerzufriedenheit, dass die Aufgabe als schwierig empfunden wird, werden weitere Nachfragen gestellt, um die Hintergründe dieser Bewertung zu ergründen. Fragen zur Nutzerzufriedenheit werden ohne Skalen gestellt und stattdessen in Freitextform durchgeführt. Der Fokus bei diesen Fragen liegt darauf Nutzbarkeitsprobleme, welche Nutzer*innen während der Durchführung hatten, zu dokumentieren und nicht die exakten Werte in Zahlen darzustellen.

Lesbarkeit von Folien: Zuerst werden Teilnehmer gebeten Foliensätzen in Quellcode-Form Informationen zu entnehmen. Die zu findenden Information sind immer auch in der Struktur der Folien kodiert. Teilnehmer müssen so nicht nur Text überfliegen, sondern auch seine Präsentation verstehen. Es gibt zwei Beispiele A und B, welche jeweils in HTML und Markdown vorliegen.

Beispiel A (A.1.1) ist ein Foliensatz, der eine Reihe geometrischer Formen beschreibt. Zusätzlich zu dem Quellcode der Folien, werden den Teilnehmenden die im Anhang abgebildeten Figuren, ebenfalls auf Papier ausgedruckt, vorgelegt. Auf den Foliensätzen werden Formen in einer durchnummerierten Liste Zahlen zugeordnet. Auf einer späteren Folie werden diese Zahlen mit den Innenzahlen (weiß im Inneren der Abbildungen der Formen) in den Kontext gesetzt. Ziel für Teilnehmer ist es, die falsch zugeordneten Innenzahlen zu finden. Die auf der zweiten Folie abgebildete Form des Sechsecks ist ein Hinweis zur Lösung. In der Innenzahlenzuordnung sind diese und der Kreis vertauscht.

Beispiel B beinhaltet eine Wahrheitstabelle mit den Eingängen `A` bis `C`. Ziel der Aufgabe ist es, das Resultat für die Eingangsbelegung `A=1,B=1,C=0` zu finden. Als Überschrift der Resultatsspalte wird bewusst „result“ gewählt, um Teilnehmenden kein Lösen des in der Tabelle repräsentierten logischen Terms zu ermöglichen.

Teilnehmende müssen Beispiel A und B in jeweils unterschiedlichen Sprachen bearbeiten. In dieser Aufgabe wird die Leserlichkeit und inhaltliche Verständlichkeit von Markdown und HTML gegeneinander verglichen. Zwei inhaltlich unterschiedliche Beispiele sind nötig, da eine Person jeweils jede Sprache einmal vorgelegt bekommt.

Da ein Teilnehmer beide Sprachen testet, spricht man bei diesem Aufbau von „Within-Subject Design“ [53, S. 133]. Es kann durch die Reihenfolge der zu bearbeitenden Beispielen zu „Carry-Over Effekten“ kommen [56, S. 18]. Für diese Studie sind die Art der auftretenden Carry-Over Effekte nur von zweitrangigem Interesse. Es soll jedoch im Aufbau ihre Auswirkung minimiert werden. Deshalb bearbeitet eine Hälfte der Teilnehmenden zuerst ein Beispiel in HTML und danach eines in Markdown. Die anderen Hälfte bearbeitet sie in umgekehrter Reihenfolge. Da nicht garantiert werden kann, dass Beispiel A & B gleich komplex sind, wird jede Gruppe (HTML vor Markdown und umgekehrt) nochmal halbiert, wobei die eine Hälfte zuerst Beispiel A und danach Beispiel B bearbeitet. Daraus ergeben sich die in Tabelle 5.1 beschriebenen Teilnehmergruppen mit der Menge an jeweils zugeordneten Teilnehmenden.

Erste Aufgabe	**zweite Aufgabe**	**Teilnehmende**
Beispiel A in HTML	Beispiel B in Markdown	3 Teilnehmende
Beispiel B in HTML	Beispiel A in Markdown	3 Teilnehmende
Beispiel A in Markdown	Beispiel B in HTML	3 Teilnehmende
Beispiel B in Markdown	Beispiel A in HTML	3 Teilnehmende

Tabelle 5.1: Teilnehmergruppen für Aufgabe 1

Im Anschluss an diese praktische Aufgabe werden den Teilnehmenden die folgenden Fragen gestellt:

1. Wie schwer war es für dich den Inhalt und die Struktur der Folien zu verstehen?
2. Welches Format ist für dich einfacher zu lesen?

Zuordnung von Folienstrukturen: Teilnehmenden werden vier Foliensätze und vier Strukturen von Foliensätzen vorgelegt. Die Aufgabe ist es, Quellcode und bildlich dargestellte Strukturen zuzuordnen. Jeder Foliensatz besteht aus sechs Folien, welche jeweils nur eine Überschrift mit einem Buchstaben von A bis F enthalten. Die Struktur eines Foliensatzes ist durch eine Reihe von Boxen mit passenden Inhalten von A bis F dargestellt. Listing 5.1 und Abbildung 5.1 zeigen ein in der Aufgabe verwendetes Beispielpaar aus Quellcode und Struktur. Die kompletten Paare aus der Aufgabe sind in Anhang A.2 zu finden.

```
# A

## D

## E

# B

# C

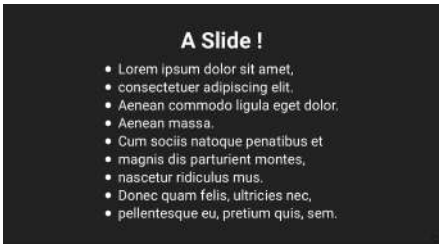
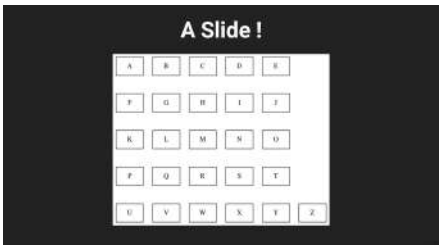
## F
```

Listing 5.1: Folienstruktur Beispielfolie



Abbildung 5.1: Folienstruktur Beispielstruktur

Jeder einzelne Foliensatz ist in HTML und Markdown verfügbar. Über die vier Foliensätze hinweg bekommt jeder Teilnehmer zwei Sätze in HTML und zwei in Markdown. Tabelle 5.2 zeigt dabei alle möglichen Kombinationen. Insgesamt wird jede Möglichkeit für zwei Teilnehmer genutzt. So wird eine gleichmäßige Verteilung der Teilnehmenden auf alle möglichen Foliensätze und Syntaxen garantiert.

	B1	B2	B3	B4
1	H	H	M	M
2	M	H	H	M
3	M	M	H	H
4	H	M	M	H
5	H	M	H	M
6	M	H	M	H

Tabelle 5.2: Syntaxmöglichkeiten für vier Beispiele B1 bis B4

Zusätzlich zu den in Abschnitt 5.2.2 beschriebenen Messwerten, wird das Vertrauen der Nutzer*innen in die Korrektheit ihrer Lösung in Anlehnung an den ISO22025 Messwert User Trust [54, S. 15] erfragt. Dies geschieht ebenfalls in Form einer vereinfachten Single Ease Question in Freitextform, um Gründe für Unsicherheiten der Nutzer*innen bei der Zuordnung von Quelltexten zu erfahren.

Im Anschluss an diese praktische Aufgabe werden den Teilnehmenden die folgenden Fragen gestellt:

1. Wie schwer fandest du diese Aufgabe? Was waren Probleme? Was waren Erkenntnisse?
2. Wie sicher hast du dich bei der Zuordnung gefühlt?

Zuordnung von Einzelfolien: Teilnehmenden werden vier einzelne Folien in Quellcode vorgelegt und vier Screenshots von Folien, wie sie im Browser angezeigt werden. Die Aufgabe ist es, die Folien und Screenshots zuzuordnen. Drei der vier Folien enthalten dabei den exakt gleichen Abschnitt Blindtext und haben diesen nur in verschiedenen präsentationstypischen Arten dargestellt. Die vierte Folie enthält ein Bild. Alle Folien liegen, wie in der vorherigen Aufgabe, auch in HTML und Markdown vor. Listing 5.2 und Abbildung 5.2 zeigen ein Beispielpaar. Alle Paare sind in Anhang A.3 zu finden. In dieser Aufgabe werden die Beispiele ebenfalls, wie in Tabelle 5.2 beschrieben, gleichmäßig unter allen Teilnehmenden verteilt.

```
<section>
        <h2> A Slide !</h2>
        <p>
                Lorem ipsum dolor sit amet, consectetuer
                adipiscing elit. Aenean commodo ligula
                eget dolor. Aenean massa. Cum sociis
                natoque penatibus et magnis dis
                parturient montes, nascetur ridiculus mus.
                Donec quam felis, ultricies nec,
                pellentesque eu, pretium quis, sem.
        </p>
</section>
```

Listing 5.2: Einzelfolien Beispielfolie

Abbildung 5.2: Einzelfolien Beispielfolie gerendert

Es wird zusätzlich nach dem Vertrauen der Teilnehmenden in die Korrektheit ihrer Lösung, wie bereits in Abschnitt 5.2.2 Zuordnung von Folienstrukturen: beschrieben, gefragt.

Im Anschluss an diese praktische Aufgabe werden den Teilnehmenden die folgenden Fragen gestellt:

1. Wie schwer fandest du diese Aufgabe? Was waren Probleme? Was waren Erkenntnisse?
2. Wie sicher hast du dich bei der Zuordnung gefühlt?

Ebenso werden sie im Anschluss an diese Aufgabe befragt, wie sie ihre Kenntnisse von der Struktur von Folien und Foliensätzen einstufen würden und ob noch Verständnisprobleme gibt. Ziel dieser letzten Frage ist es, Einblicke zu bekommen, ob die ersten drei Aufgaben und der in ihnen gezeigte Quellcode ausreicht, um Teilnehmenden die Grundlagen der Folienerstellung zu vermitteln.

Erstellen von Folien: Teilnehmende werden aufgefordert, einen einfachen Foliensatz aus vier Folien zu entwerfen. Auf den vier Folien sind Überschriften, Listen und Codeblöcke enthalten, um einen Teil der regelmäßig in Folien vorkommenden Elemente in der Aufgabe zu inkludieren. Einfache Textabsätze werden wegen ihres Schreibaufwandes bei der Erstellung nicht inkludiert, ebenso keine Bilder. Diese benötigen zur korrekten Anzeigen in vielen Fällen noch HTML Zusatzinformationen für Größe und Positionierung. Um die benötigten Sprachkenntnisse gering zu halten, wurde auf Bilder in den Folien verzichtet. Der Foliensatz besteht aus einer Titelfolie (A) mit einer Unterfolie (B) die eine Agenda als nummerierte Liste enthält. Außerdem aus einer weiteren Titelfolie (C) und einer Unterfolie (D) mit einem Code Block. Alle Folien enthalten Überschriften. Der Aufbau des Foliensatzes ist in Abbildung A.4 dargestellt. Diese Abbildung wird auch den Teilnehmenden als Referenz zusätzlich zu der mündlichen Erläuterung der Aufgabe vorgelegt. Die eine Hälfte der Teilnehmenden muss die Folien in HTML entwerfen, die andere Hälfte in Markdown.

Im Anschluss an diese praktische Aufgabe werden den Teilnehmenden die folgenden Fragen gestellt:

1. Wie war es für dich diese Folien zu schreiben? Gab es Hürden? Was ging einfach?
2. Hast du dich effizient bei der Umsetzung der Aufgabe gefüllt?

Auflösen von Pfaden: Teilnehmenden wird erklärt, dass es nun um das Einbinden von Quellcode in Folien geht. Ihnen wird ausschließlich der Name des Konzepts zur Auswahl von Quellcode „Pfade“ genannt. Im Anschluss werden sie gebeten, vier Pfade in einer auf Papier vorliegenden Quellcodedatei zu finden und die durch den Pfad beschriebene Zeile oder beschriebenen Zeilen zu markieren. Die vorgelegten Pfade und die Quellcodedatei sind im Anhang unter A.5 in dieser Arbeit inkludiert.

Als Sprache für den Quellcode wird ein einfaches Scala Subset genutzt, da Scala eine der implementierten Sprachen dieser Arbeit ist. Es wird versucht, den Quellcode möglichst java-ähnlich aussehen zu lassen, indem beispielsweise nur `var` Keywords zur Definition von Variablen genutzt werden. Die Ähnlichkeit mit Java wurde angestrebt, da es sich um eine verbreitete Sprache handelt, deren optischer Eindruck für viele Programmierer bekannt ist. Es wird erhofft, dass viele

Teilnehmende verschiedener Sprachhintergründe die generelle Schachtelung von Definitionen in Java wiedererkennen können. Ein semantisches Verständnis des Quellcodes ist für diese Aufgabe nicht erforderlich.

Im Anschluss werden den Teilnehmenden die Frage gestellt, welche Hürden und Verständnisprobleme beim Lösen der Aufgabe überwunden werden mussten.

Erstellen von Pfaden: Teilnehmende werden aufgefordert, für vier in einer Quelltextdatei markierte Stellen Pfade aufzuschreiben. Die Quelltextdatei und zu markierenden Zeilen sind in Anhang A.6 zu finden.

Im Anschluss werden die selben Fragen gestellt wie im vorherigen Abschnitt.

5.3 Auswertung

Im Folgenden werden die Beobachtungen und Ergebnisse der Teilnehmenden im Kontext der am Anfang der Evaluation aufgestellten Fragen dargestellt.

Die Form der Auswertung folgt nicht dem ISO/IEC Standard 25062. Dieser ist eigentlich als Komplement zu ISO/IEC 25022 gedacht und stell einen Standard für Usability Reports dar. ISO/IEC 25063 wird nicht angewendet, da dieser Standard für summative Tests konzipiert ist und viele Informationen sowie Details verlangt, die für die meisten Reports formativer Tests nicht notwendig sind [57, S. 5]

Wo es möglich ist, wird auf die Aussagen der Teilnehmenden verwiesen. Diese sind in Anhang B zu finden. Dort sind für jeden einzelnen Teilnehmenden die Aussagen und Beobachtungen, wie sie während des Tests aufgenommen wurden, festgehalten. Eine Markierung der Form „(P_3, P_4)“ im folgenden Text verweist dabei auf die Aussagen der Teilnehmenden 3 und 4.

In den Aufzeichnungen in Anhang B sind direkte Zitate der Teilnehmenden in Anführungszeichen gesetzt und Kommentare des Autors in eckigen Klammern markiert. Bei der Testdurchführung wurden alle Fragen und Aufgaben durchnummeriert. Die Aussagen der Teilnehmenden sind, um Platz zu sparen, nur den jeweiligen Nummern der Fragen und Aufgaben zugeordnet. Tabelle B.2 in Anhang B.2 zeigt die Zuordnung der Aufgaben und ihrer Fragen zu den verwendeten Zahlen.

5.3.1 Teilnehmende

Als Teilnehmende wurden zwölf Personen ausgewählt, die potenzielle Nutzer von Cobra darstellen. Das Feld der Teilnehmenden reicht von Master-Studierenden der Informatik und Digitalen Medien bis hin zu angestellten Softwareentwickler*innen. Auch Zwischenformen waren dabei, im Speziellen Auszubildende Informatiker*innen für Anwendungsentwicklung und duale (Master-) Studenten*innen.

Bei Softwareentwickler*innen handelt es sich in allen Fällen um Entwickler*innen aus dem Webumfeld mit guten HTML-Kenntnissen. Es handelt sich bei der Aus-

wahl von Softwareentwicklern aus dem Webumfeld um keine bewusste Entscheidung. Die Häufung dieses Hintergrundes ist der Art des Zugangs zu Entwickler*innen geschuldet. Bei den Studierenden reichte das Spektrum der HTML-Kenntnisse nach eigenen Einschätzungen von sehr gut bis hin zu grundlegenden Kenntnissen aus Schulzeiten, welche in den letzten Jahren nicht mehr genutzt wurden.

Vorerfahrungen in Markdown reichten von regelmäßiger Nutzung zum Schreiben von Notizen und Abgaben, Dokumentationen oder Websiteinhalten bis hin zu keinen Vorkenntnissen. Im Fall von fehlenden Vorkenntnissen wurde den Teilnehmenden eine kurze Einführung in die Struktur von Markdown gegeben. Dabei wurde vor allem die Syntax für Überschriften erklärt und wie diese mit HTML-Überschriften zusammenhängen.

Tabellen 5.3 und 5.4 geben einen Überblick über die Hintergründe aller Teilnehmenden sowie ihre Markdown- und HTML-Vorkenntnisse. Tabelle B.1 in Anhang B.1 gibt einen zusammengefassten Überblick über diese Informationen sowie die reveal.js-Kenntnissen der Teilnehmenden.

P_n	**Hintergrund**
1	PHP Entwickler_in Full Stack
2	Java Entwickler_in Full Stack
3	Frontend Entwickler_in
4	Azubi Informatik (2. LJ) Schwerpunkt PHP
5	Frontend Entwickler_in
6	Frontend Entwickler_in
7	Azubi Informatik (2. LJ) Schwerpunkt Frontend
8	Master Student_in Digitale Medien
9	Master Student_in Informatik
10	Master Student_in Informatik
11	Master Student_in Informatik
12	Master Student_in Informatik

Tabelle 5.3: Hintergründe der Teilnehmenden

5.3.2 Markdown für Folien

Im Folgenden werden die Ergebnisse präsentiert, die sich mit dem Vergleich von Markdown und HTML als Sprache für Folien befassen. Insbesondere wird sich mit der in Abschnitt 5.1 gestellten Frage, ob durch die Nutzung von Markdown eine Verbesserung der Nutzbarkeit gegenüber HTML erreicht worden ist, auseinander gesetzt.

Inhaltsverständnis: Im Kontext von Cobra kommen Aufgaben vor, bei denen Informationen in einem Foliensatz gefunden werden müssen, z.B. um Korrekturen vorzunehmen oder Beispiele auszutauschen. Aus diesem Grund wird die Leserlichkeit von Folien untersucht.

Es war allen Teilnehmenden möglich, den Inhalt der Folien so weit zu verstehen, wie es für die Beantwortung der Aufgabenstellung nötig war.

In HTML wurde die klare vertraute Struktur des Quellcodes positiv erwähnt (P_1, P_2, P_5, P_8). Dem gegenüber steht die Teilnehmerbemerkung, dass HTML durch seine vielen strukturgebenden Elemente vom Inhalt ablenkt (P_3, P_4, P_6, P_{12}). Zu Markdown gab es in ihrer Anzahl insgesamt weniger Äußerungen, sowohl positive als auch negative. Positiv wurde angemerkt, dass die endgültige Struktur, Form sowie Aufbau des Inhaltes im Code ersichtlich sind (P_9, P_{11}).

Ein Vergleich der beiden Sprachen basierend auf der Bearbeitungszeit ist auf Grund der geringen Zahl an Teilnehmenden nicht möglich. Ebenso ist ein Vergleich der Errors in Task nicht sinnvoll, da viele Teilnehmende Probleme hatten die Aufgabe 5.2.2 zu verstehen. Für das Beispiel A (A.1.1) gab es Rückfragen und Verständnisprobleme bezüglich der Aufgabenstellung und Materialien von sechs Teilnehmenden (P_3, P_6, P_7, P_9, P_{11}, P_{12}). Vor allem der Zusammenhang zwischen den Folien und den in den Materialien enthaltenen Bildern mit den weißen Zahlen war für diese Teilnehmer nicht klar. Bei Beispiel B war vier Teilnehmenden (P_3, P_4, P_6, P_{11}) das Konzept einer Wahrheitstabelle nicht bekannt und musste zuerst erläutert werden. Falsche Antworten und lange Bedenkzeiten bei Teilnehmenden in beiden Beispielen waren oft verbunden mit Nachfragen zu Aufgabenstellungen und Materialien.

Es konnte keine klare Präferenz einer Sprache bei allen Teilnehmenden festgestellt werden. Auf die direkte Frage, welche Sprache angenehmer zu lesen war, antworteten sieben Teilnehmende mit Markdown, vier mit HTML, eine Person war unentschlossen.

Es lässt sich vermuten, durch die geringe Teilnehmerzahl jedoch nicht statistisch belegen, dass es einen Zusammenhang zwischen der präferierten Sprache und der Vertrautheit mit einer der beiden Sprachen gibt. In Tabelle 5.4 sind die Vorkenntnisse der Teilnehmenden und ihre Präferenz zwischen HTML und Markdown zusammengefasst. Alle HTML bevorzugenden Teilnehmenden haben durch ihren Hintergrund sehr regelmäßigen Kontakt mit HTML und alle Markdown bevorzugenden Personen laut eigenen Angaben bereits Erfahrungen in Markdown (vgl. vgl. Tabelle B.1 und Tabelle 5.4). Vorkenntnisse einer Sprache alleine scheinen nicht als Indikator für eine Präferenz auszureichen. Es gibt in beiden Richtungen selbst in dieser kleinen Teilnehmergruppe Beispiele für Personen die trotz Markdownkenntnissen HTML bevorzugen und anders herum. Eine Überschneidung der Präferenz mit Vorkenntnissen ist jedoch zu erkennen.

Strukturverständnis: Das Schreiben und Lesen von Folienquellcode erfordert ein Verständnis des Schreibenden, wie Quellcode und Folien in Relation zueinander stehen. Dies ist sowohl auf dem Level einer einzelnen Folie, wie auch einer vollständigen Präsentation der Fall. Aus diesem Grund wurde die Fähigkeit von Teilnehmenden Folien und Foliensätze ihrem Quellcode zuzuordnen getestet.

Beim Zuordnen von Foliensätzen zu ihrer Gitterstruktur erschienen monotone Strukturen (Beispiele 1 und 2) schwieriger zuordenbar als abwechslungsreichere Strukturen (Beispiele 3 und 4). Während der Durchführung äußerten sich

P_n	**HTML**	**MD**	**Präferenz**
1	sehr gut	vorhanden	HTML
2	sehr gut	Website Content	HTML
3	sehr gut	Dokumentation und Tickets	Markdown
4	sehr gut	Abgaben Berufsschule	Markdown
5	sehr gut	Notizen	HTML
6	sehr gut	Dokumentation	Markdown
7	sehr gut	„nicht wirklich“	HTML
8	gut	nein	beides gleich
9	gut	Readmes	Markdown
10	„etwas eingestaubt“	Readmes und Uni Abgaben	Markdown
11	grundlegend	Notizen und Uni Abgaben	Markdown
12	gering	Gitlab Wikis und Uni Abgaben	Markdown

Tabelle 5.4: Vorkenntnisse der Teilnehmenden in HTML und Markdown und ihre Präferenz

drei Personen dazu, dass die explizite Struktur des HTML Quellcodes bei der Zuordnung von Foliensätzen zu Strukturen hilfreich war (P_3, P_4, P_7).

Weder bei der Zuordnung von Foliensätzen zu Quellcode noch bei der Zuordnung von gerenderten Einzelfolien zu Quellcode äußerten Personen Probleme. Allen war es möglich einzelne Folien basierend auf der Struktur ihres Inhaltes im ersten Versuch korrekt und nach eigenen Angaben sicher, ungeachtet der Quellcodesprache, zuzuordnen.

Schreiben von Folien: Da das Schreiben von Folien die primäre Aufgabe von Autor*innen ist, wird untersucht, ob es basierend auf der verwendeten Sprache Unterschiede in der Nutzbarkeit gibt.

Während der Erstellung von Folien bemerkten mehrere Teilnehmer, dass für sie die Regeln zum Konvertieren von Quelltext zu Folien nicht klar verständlich waren (P_3, P_5, P_6) und die Anordnung von Folien in einem Gitter anstelle einer vertikalen Linie ungewohnt (P_1, P_9, P_{10}) war. Diese Kritik kam zu gleichen Teilen von HTML- und Markdown-Schreibenden.

Vier von sechs HTML-schreibende Personen fühlten sich bei der Umsetzung der Folien nach eigenen Angaben nicht effizient (P_4, P_7, P_10, P_11) und zwei bemängelten die Menge der zu schreibenden Syntaxelemente (P_{10}, P_{11}). Fünf von sechs Teilnehmenden äußerten sich positiv (P_2, P_3, P_6, P_9, P_{12}) über ihre Schreiberfahrung mit Markdown.

Die Betrachtung der Bearbeitungszeit stützt diese Aussage, da mit nur wenigen Ausnahmen Markdown-schreibende Teilnehmende weniger Zeit für die Umsetzung der Aufgabe gebraucht haben als die HTML-Schreibenden. Dies sollte jedoch nicht als ein eindeutiges Zeichen dafür gesehen werden, dass Markdown effizienter in der Erstellung von Folien ist als HTML. Die Lösung in HTML ist durch die zu schreibenden Tags circa doppelt so lang wie die Markdown Lösung. Da alle Teilnehmenden auf Papier schreiben mussten, kann die längere Zeit zur Umsetzung schlicht daran liegen, dass in HTML mehr Zeichen geschrie-

ben werden mussten. Beim Erstellen von Folien in einem realistischeren Szenario in einem Texteditor können z.B. schließende Tags automatisch erzeugt werden, was den Schreibaufwand für HTML verringert und die Erstellungszeiten näher zusammenbringt.

Im Gegensatz dazu äußerten sich zwei andere HTML-Schreibende positiv zu ihrer Erfahrung beim Erstellen der Folien (P_1, P_5). Dies ist aber auch unter dem Gesichtspunkt zu betrachten, dass die Teilnehmenden P_1 und P_5 durch ihren Background täglich HTML schreiben.

In beiden Gruppen wurden ähnlich viele Fehler gemacht. Die verwendete Sprache schien keinen Einfluss auf die Anzahl von Fehlern zu haben, doch je nach Sprache gab es unterschiedliche Fehlerschwerpunkte:

- **Schließende Section Tags in HTML** Diese wurden immer an einer von zwei möglichen Stellen vergessen, in denen eine Unter- und Hauptfolie beendet werden musste.
- **Falsche Folienstruktur in Markdown** Es wurden die falschen Überschriftenlevel verwendet, sodass die von Teilnehmern erstellten Folien in der Struktur nicht der Geforderten entsprach.

Allen Teilnehmenden war es unabhängig von der Sprache möglich, ihre Fehler nach Hinweis auf diese direkt zu beheben.

Konklusion: Stellt Markdown nun eine Verbesserung gegenüber HTML beim Erstellen und Verstehen von Folien dar? Wenn es um das Lesen von Folien und das Verständnis ihres Inhaltes oder Struktur geht, scheint Markdown keinen offensichtlichen Vorteil zu bieten, der es für diese Aufgabe objektiv über HTML stellt. Es ist eine Abwägung zwischen auf den Inhalt konzentrierten Dokumenten in Markdown oder klar definierten Strukturen in HTML. Wo in Markdown die visuelle Ähnlichkeit des Quellcodes gelobt wurde, präferierten andere Teilnehmer die klare und explizite Struktur von HTML mit seinen Einrückungen.

Beim Schreiben von Folien zeichnet sich ein anderes Bild. Markdown ist im Fall von einfachen Folien kompakter als HTML. Der weitestgehende Verzicht auf syntaktische Konstrukte ermöglichte es Teilnehmenden, meist schnell und nach eigenen Aussagen effizient und mit Vertrauen in ihre Fähigkeiten, Folien in Markdown zu schreiben. Dem gegenüber steht die kritische Aussage, dass die Abstraktion von Markdown auf Folien schwierig sei. Die genauen Ursachen der Kritik zu finden war nicht Ziel dieser Evaluation. Ein möglicher Grund könnte jedoch die Tatsache sein, dass Markdownfolien in den gezeigten Beispielen nur implizit durch das Beginnen neuer Überschriften voneinander getrennt sind und nicht wie in HTML durch klare explizite umfassende Tags. In einer späteren Evaluation wäre es interessant zu testen, ob durch die Nutzung von expliziten Folientrennern (vgl. 2.3.2) eine solche Kritik verringert werden könnte.

Bemerkenswert in der Evaluationsdurchführung ist das Ergebnis von Person P_8: Es war ihr möglich ohne vorherige Markdownkenntnisse Folien in Markdown fehlerfrei zu schreiben. Es war nicht damit gerechnet worden, dass die bis zu diesem Punkt gezeigten Beispiele für Markdownfolien ausreichend wären, um fehlerfrei selbst Folien zu produzieren.

Zusammenfassend eignet sich jede der beiden Sprachen besser für die eine oder andere Art von Folien und besser für den einen oder die andere Nutzer*in. Soweit möglich, sollten beide Sprache angeboten werden, um die größtmögliche Anzahl an Nutzer*innen mit jeweiligen Präferenzen in möglichst vielen Anwendungsfällen zu unterstützen.

5.3.3 Pfade zur Auswahl von Beispielen

Im Folgenden werden die Ergebnisse der Evaluation in Bezug auf die Frage, ob Pfade als Metapher zur Auswahl von Beispielen geeignet sind, vorgestellt. Dabei wird ausgewertet, ob Teilnehmende durch Beispiele ein Verständnis für die Bedeutung von Pfaden entwickeln konnten und es ihnen möglich war selbst Pfade zu erzeugen.

Verständnis der Bedeutung: Damit das Konzept von Pfaden durch Nutzer*innen angenommen wird, ist es wichtig, dass es für sie verständlich ist. Im Idealfall bedarf es zum Verständnis keiner Erläuterung. Deshalb wird untersucht, ob Pfade intuitiv verstanden werden.

Es wurde beobachtet, dass sieben Teilnehmer nur mit Erläuterung des Ziels von Pfaden - der Definition von Abschnitten in Quellcode - intuitiv beim Auflösen von Pfaden zu Quellcodestellen jedes Element durchgegangen sind (P_4, P_6, P_8, P_9, P_{10}, P_{11}, P_{12}). Mit dieser Technik haben sie sich vom Beginn des Pfades zu der gesuchten Stelle durchgearbeitet. Sie ergriffen also ohne Anleitung die korrekten Schritte, um Pfade aufzulösen. Die Methodik zum Auflösen der Elemente reicht dabei vom Suchen gleichnamiger Elemente im Quelltext (P_7) über das Finden von Elementen entlang der geschachtelten hierarchischen Struktur von Codeelementen (P_8). Vier Teilnehmende empfanden Pfade nach eigenen Aussagen als intuitiv (P_1, P_4, P_5, P_{12}). Im Gegensatz äußerte sich eine Person explizit gegenteilig (P_7). Alle hier nicht aufgeführten Teilnehmenden äußerten sich nicht explizit, ob sie dieses Feature als intuitiv empfanden. Ebenso gab es keine Aussagen über die Verwendung von Punkten als Trennzeichen von Elementen. Alle Teilnehmenden haben Pfade korrekt an Punkten in Elemente getrennt.

Im Gegensatz zu den intuitiv verstandenen Pfaden stehen Schlüssel. Ein Beispiel enthielt einen Schlüssel, welcher den Pfad auf eine Variablendefinition beschränkt und die Kollision mit einer gleichnamigen Typdefinition auflöst. Im Beispielcode ist im Textfluss zuerst die auszuwählende Variablendefinition und später die kollidierende Typdefinition abgedruckt. Es kam beim Auflösen von dieser Pfadkollision zu den meisten Fehlern in der Aufgabe. Fünf Teilnehmende haben statt der Variable die Typdefinition als Antwort ausgewählt (P_4, P_5, P_6, P_9, P_{10}). Viele andere der hier nicht aufgeführten Teilnehmenden haben die Variablendefinition ausgewählt. Es wird vermutet, dass dies so ist, weil diese zuerst im Quellcode vorkommt. Zwei dieser Teilnehmenden gaben auf spätere Nachfrage hin an, die Kollision an sich übersehen zu haben.

Auf ihre Intuition angesprochen, was „[t:Variable]“ bedeuten könnte, antworteten neun Teilnehmende, dass sie vermuten „t“ stünde für Typ (P_2, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}). Nur in zwei Fällen wurde vermutet, dass es sich um

die Beschränkung der Abfrage auf den Typen einer Variable handelt (P_2, P_9). Insgesamt äußerte sich keine Person, dass sie Schlüssel als intuitiv verständlich empfände. Schlüssel wurden im Gegenteil von neun Teilnehmenden als nicht intuitiv bezeichnet (P_1, P_4, P_5, P_6, P_9, P_{10}, P_{11}, P_{12}).

Die hohe Ähnlichkeit von Pfaden zu Konstrukten, wie den Zugriff auf Variablen in geschachtelten Klassen, wird als Grund für die Beobachtung vermutet, dass mehrere Teilnehmende initial annahmen, dass Pfade mit dem gezeigten Quellcode interagieren. Beispielsweise wurde von einer Person vermutet, dass ein Pfad den in einer Variable gespeicherten Wert auswählt (P_3). Eine weitere Person war sich unsicher, ob durch einen Pfad die Definition oder Nutzung einer Variable gemeint war (P_9). Wie auch in vorherigen Aufgaben, könnte es interessant sein, diese Vermutung in einem zukünftigen Nutzertest mit abgewandelten Pfade zu testen. Eine mögliche Abwandlungen ist z.B. „/“ statt „.“ als Trennzeichen zu verwenden.

Erstellung von Pfaden: Neben dem Verstehen müssen Pfade, ebenso wie Folien, erstellt werden. Nachdem Nutzer*innen mit dem Konzept von Pfaden in einer früheren Aufgabe konfrontiert waren, wird untersucht, ob sie selbst Pfade erstellen können.

Pfade wurden auch beim Erstellen als „sich natürlich anfühlend“ (P_2) und einfache Baumtraversierung zu der gewünschten Stelle (P_{10}) beschrieben. Das Erstellen mehrerer Pfade wurde als einfaches Abarbeiten empfunden (P_6, P_{11}).

In fünf Fällen war es Teilnehmenden nicht klar, wie sie mehrere Zeilen, in der Aufgabe die komplette Definition einer Klasse, durch einen Pfad ausdrücken können (P_5, P_6, P_8, P_9, P_{11}). Alle Teilnehmenden haben trotz ihrer Unsicherheit mit einem Pfad bis zur Klassendefinition intuitiv den richtigen Pfad für diese Aufgabe erstellt.

Fehlerschwerpunkt bei der Erstellung von Pfaden war, wie auch schon beim Lesen, die Verwendung von Schlüsseln, um den in der Quellcodedatei vorhandenen Namenskonflikt für einen aufzustellenden Pfad aufzulösen. In fünf Fällen wurde der Schlüssel vergessen (P_5, P_7, P_8, P_9, P_{10}), in zwei Fällen davon wurde die zu lösende Pfadkollision übersehen (P_7, P_{10}). Im Fall von Fehlern wurde Teilnehmenden gezeigt auf welche Stelle ihr fehlerhafter Pfad deutet. In allen Fällen war es Teilnehmenden möglich, ihre Fehler im Anschluss zu berichtigen. Eine beobachtete Strategie war, dass Teilnehmende ihre aufgeschriebenen Pfade wie in der vorherigen Aufgabe versuchten aufzulösen und dabei auf ihre Fehler gestoßen sind (P_4).

Konklusion: Durch Punkte getrennte Pfade scheinen ein intuitiv verständliches und nutzbares Konzept zur Auswahl von Quellcode darzustellen. Es bedarf nur minimaler Einweisung, um ein unbekanntes Konzept für Teilnehmende nutzbar zu machen und nicht erläuterte Konzepte, wie die Auswahl mehrerer Zeilen, konnten sich alle selbst korrekt herleiten. Die große visuelle Ähnlichkeit von Pfaden mit Punkten als Trennzeichen zu normalen Programmierkonstrukten hat in Einzelfällen für Verunsicherung gesorgt. Größte Quelle für Verunsicherung in den Nutzertests jedoch war die Verwendung von Scala als Sprache für Snippets

(P_4, P_5, P_{10}, P_{12}) und die Vorlage von Quellcode auf Papier (P_5, P_7, P_8). Ob andere Trennzeichen außer Punkte für weniger Verunsicherung sorgen und ebenso oder besser verständlich sind, wurde nicht evaluiert.

Schlüssel zum Einschränken von Pfaden scheinen in ihrer aktuellen Form weniger verständlich als Pfade zu sein und waren in der Evaluation ein klarer Fehlerschwerpunkt. Es sollte im Rahmen einer Weiterentwicklung der Pfade evaluiert werden, ob sie mit der Vorlage einer kurzen Dokumentation und Beispielen für Teilnehmende verständlicher werden. Wenn dies nicht der Fall ist, sollte überlegt werden ein anderes Konzept zur Vermeidung oder Auflösung von Pfadkollisionen zu erarbeiten. Denn Pfade ohne Schlüssel haben sich in der Evaluation als vielversprechendes Konzept bewiesen, welches zur Zeit in seiner Nutzbarkeit von Schlüsseln mit Verbesserungspotential eingeschränkt wird.

Kapitel 6

Fazit

Ziel dieser Arbeit war es Konzepte zu entwickeln, die Autoren das Verfassen von Folien für das Tool Cobra vereinfachen. Dazu sollte eine Methode zur Auswahl von Beispielen in Folien entwickelt werden. Sie sollte intuitiv verständlich sein, ohne Annotationen an den Beispielen funktionieren, sowie robuster gegenüber Dateiänderungen sein als aktuell verfügbare Methoden. Im Genaueren sollte sie stabiler sein als die Angabe von Dateipfaden und Zeilennummern. Neben der neuen Beispielauswahl sollte eine Sprache zum Verfassen von Folien eingeführt werden, die im Vergleich mit HTML einfacher zu schreiben ist.

Die Methode zur Auswahl von Beispielen wurde auf Basis von Pfaden in vereinfachten Syntaxbäumen entwickelt. Dabei verläuft die Auswahl durch die Angabe von Pfaden von Identifiern in diesen Bäumen. Die Methode erfordert es Projekte zu definieren, um aus ihnen Quellcode auszuwählen. Aus allen Quellcodedateien in einem Projekt wird ein gemeinsamer vereinfachter Baum erstellt. Diese Art von Abfragen funktioniert ohne Annotationen von Quellcodedateien und ist gegenüber Dateiänderungen resistenter als Pfade und Zeilenangaben. Dies gilt vor allem für Änderungen, welche im Kontext des Programmierens als eigentlich nebeneffektfrei angenommen werden, wie das Entfernen einer überflüssigen Leerzeile oder eines nicht benötigten Import Statements. Das Konzept eignet sich zur Lösung der gegebenen Problemstellung. Die Anforderung der intuitiven Verständlichkeit wurde in einem Nutzertest überprüft. Es kann angenommen werden, dass Pfade an sich die Anforderungen erfüllen, da es allen Teilnehmenden intuitiv möglich war, Pfade aufzulösen und zu erstellen.

Während der Suche nach einer Alternative zu HTML wurde sich entschieden, die Suche auf bereits etablierte Sprachen beschränkt, mit dem Ziel, dass Autoren bereits vorhandene Sprachkenntnisse nutzen können. Genauer wurde die Suche auf leichtgewichtige Markup-Sprachen beschränkt. Diese zeichnen sich durch simple Syntax und gute Lesbarkeit des Quellcodes aus. Wegen seiner weiten Verbreitung wurde Markdown ausgewählt. Es ist aus den Ergebnissen des Nutzertests anzunehmen, dass es eine nutzbare Alternative zu HTML zum Schreiben einfacher Folien darstellt. Teilnehmende äußerten sich, dass sie sich beim Erstellen von Folien in Markdown effizient fühlten. Aus den Testergebnissen kann jedoch nicht geschlussfolgert werden, dass Markdown HTML als

Foliensprache komplett ersetzen kann. Nutzer*innen mit regelmäßigem Kontakt zu HTML bevorzugten in manchen Fällen diese Sprache.

Die Umsetzung erfolgte unter Verwendung des Dokumentenkonvertierungstools pandoc. Die durch pandoc vorgegebenen Regeln für die Konvertierung von Markdown zu Folien erwiesen sich im Nutzertest als verständlich und durch Teilnehmende anwendbar. Sie bestätigen Markdown als in der Praxis nutzbare Alternative zu HTML. Vor allem für regelmäßige Markdown-Nutzer*innen kann angenommen werden, dass Markdown für sie die Nutzung von Cobra vereinfacht.

Ausblick: Im Nutzertest wurde eine Schwachstelle des Pfadkonzeptes entdeckt. Durch die starke Vereinfachung von Syntaxbäumen kann es zu Pfadkollisionen kommen. Diese müssen durch die Angabe von Zusatzinformationen am Pfad, wie Projektzugehörigkeit und Typ des ausgewählten Knoten, behoben werden. Als Syntax dazu wurden Key-Value-Paare in der Form `[k:v]` festgelegt. Als Schlüssel wurden einzelne Buchstaben genutzt, z.B. „t“ für Typen in `[t:Variable]`. Für Teilnehmende war es nur in wenigen Fällen möglich, diese Typeneinschränkung zu verstehen. In einer zukünftigen Arbeit sollte die Option aussagekräftigerer Schlüssel untersucht werden. „type“ statt „t“ in `[type:Variable]` könnte bereits die gewünschte Verbesserung bewirken. Ebenso sollten natürlichsprachliche Definitionen wie `[key mustBe value]` oder `[key of value]` in Betracht gezogen werden.

Bei der Umsetzung der Projektanalyse stand die Performance der Implementierung nicht im Vordergrund. Bei jedem Starten von Cobra werden alle Projekte analysiert. Es sollte das Speichern von Analyseergebnissen zwischen Ausführen von Cobra im Dateisystem untersucht werden. So müssten nur geänderte Projekte oder Projektdateien analysiert werden und die Zeit vom Starten von Cobra bis zum Anzeigen einer Präsentation kann verkürzt werden. Dazu kann z.B. das während der Erstellung dieser Arbeit veröffentlichte Language Server Index Format verwendet werden [58].

Ebenso ist das Nutzerfeedback der integrierten Konzepte eher technischer Natur und aktuell zum größten Teil nur in Konsolen-Ausgaben und Logdateien zu finden. Um eine gute Nutzbarkeit von Cobra für Endnutzer*innen zu gewährleisten, sollte das Nutzerfeedback im Client erweitert werden.

Einordnung der Ergebnisse: Markdown als Alternative zu HTML stellt eine sinnvolle Erweiterung von Cobra dar. Im Vergleich zu anderen Sprachen ist Markdown sehr weit verbreitet und vielen Entwickler*innen bekannt. Durch die Verwendung des flexiblen Konvertierungstools pandoc ist eine einfache Möglichkeit gegeben, in Zukunft weitere Sprachen zu unterstützen. Es ist anzunehmen, dass die parallele Verwendung von HTML und Markdown es einer breiteren Menge von Entwickler*innen ermöglicht, durch Nutzung bereits bestehender Sprachkenntnisse mit geringem Lernaufwand Cobra zu nutzen. Ebenso bietet Markdown bestehenden Nutzer*innen die Chance einfache Folien mit hohem Fokus auf Inhalt in weniger Zeichen als vergleichbares HTML zu schreiben.

Pfade zur Auswahl von Beispielen, wie in dieser Arbeit beschrieben, haben sich als intuitiv verständliches und nutzbares Konzept bewiesen. Sie ermöglichen die Auswahl von Beispielen in einer dynamischen Form, die Quellcode in seiner originalen Form erhält und eine geringeres Risiko hat, durch Dateiänderungen invalide zu werden, als bestehende Konzepte. Es ist vorstellbar das Konzept in vielen anderen Kontexten wie LaTeX-Dokumenten und Markup-Sprachen zur Auswahl von Beispielen zu nutzen. Die Integration des Konzeptes in pandoc-Filter bietet die Chance dieses Konzept mit vergleichsweise geringem Entwicklungsaufwand einer großen Menge von Sprachen zugänglich zu machen.

Die Auswahl von Beispielen über strukturelle Pfade bietet einen vielseitig einsetzbaren Weg für Programmierer*innen Beispiele in ihnen vertrauten Konzepten auszudrücken. Es wird gehofft, dass Pfade sich auch außerhalb von Cobra bewähren und etablieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literatur

[1] Martin Ring und Christoph Lüth.
„Interactive Proof Presentations with Cobra“.
In: *Workshop on User Interfaces for Theorem Provers (UITP2016)*.
Hrsg. von Serge Autexier und Pedro Queresma. Bd. 239.
Electronic Proceedings in Theoretical Computer Science.
Open Publishing Association, 2017, S. 43–52.

[2] Arnout Engelen, Konrad Malawski und Roman Filonenko.
Server Websocket Support. (Abruf: 03.01.2020 20:00). März 2019.
URL: https://doc.akka.io/docs/akka-http/current/server-side/websocket-support.html.

[3] Martin Ring. *Cobra*. (Abruf: 14.01.2020 20:25). Juli 2019.
URL: https://github.com/flatmap/cobra.

[4] Hakim El Hattab u. a. *reveal.js Contributions*. (Abruf: 03.01.2020 19:45).
Dez. 2019. URL:
https://github.com/hakimel/reveal.js/graphs/contributors.

[5] Hakim El Hattab u. a. *reveal.js readme/manual*.
(Abruf: 19.11.2019 14:05). Apr. 2019.
URL: https://github.com/hakimel/reveal.js/.

[6] Patrick Nordwall, Johan Andrén, Jimin Hsieh u. a.
Actor Systems - Hierarchical Structure. (Abruf: 03.01.2020 21:25).
Nov. 2019.
URL: https://doc.akka.io/docs/akka/current/general/actor-systems.html#hierarchical-structure.

[7] Johan Andrén, Robert Stoll, Patrick Nordwall u. a. *What is an Actor?*
(Abruf: 03.01.2020 21:20). Nov. 2019. URL:
https://doc.akka.io/docs/akka/current/general/actors.html.

[8] Carl Hewitt, Peter Bishop und Richard Steiger. „Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence“.
In: (1973). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.77.7898.

[9] Gul Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
Cambridge, MA, USA: MIT Press, 1986. ISBN: 0262010925.

[10] Johan Andrén, Robert Stoll, Patrick Nordwall u. a. *What is an Actor?*
(Abruf: 12.01.2020 20:00). Nov. 2019. URL:
https://doc.akka.io/docs/akka/current/general/actors.html.

[11] Patrick Nordwall, Johan Andrén, Helena Edelson u. a.
Classic Actors - Actor Lifecycle. (Abruf: 03.01.2020 21:15). Dez. 2019.

URL: https://doc.akka.io/docs/akka/current/actors.html#actor-lifecycle.

[12] Microsoft. *Language Server Protocol*. (Abruf: 05.02.2020 22:00). URL: https://microsoft.github.io/language-server-protocol/.

[13] Erich Gamma und Dirk Bäumer. *Protocol History*. (Abruf: 14.01.2020 20:40). Juni 2016. URL: https://github.com/microsoft/language-server-protocol/wiki/Protocol-History.

[14] Microsoft. *Overview - What is the Language Server Protocol?* (Abruf: 07.12.219 20:00). URL: https://microsoft.github.io/language-server-protocol/overview.

[15] Keegan Carruthers-Smith, Dan Adler, Waldir Pimenta u. a. *Langserver.org - A community-driven source of knowledge for Language Server Protocol implementations*. (Abruf: 03.01.2020 15:35). Dez. 2019. URL: https://langserver.org/.

[16] Aaron Swartz. *Markdown*. (Abruf: 03.01.2020 15:20). März 2004. URL: http://www.aaronsw.com/weblog/001189.

[17] John Gruber. *Markdown*. (Abruf: 03.01.2020 15:10). Dez. 2004. URL: https://daringfireball.net/projects/markdown/.

[18] John MacFarlane. *CommonMark Spec Version 0.29*. (Abruf: 14.01.2020 21:05). Apr. 2019. URL: https://spec.commonmark.org/0.29/.

[19] John MacFarlane. *Babelmark 2 - FAQ*. (Abruf: 14.01.2020 21:10). 2012. URL: https://johnmacfarlane.net/babelmark2/faq.html.

[20] John MacFarlane, David Greenspan, Vicent Marti u. a. *CommonMark - A strongly defined, highly compatible specification of Markdown*. (Abruf: 26.01.2020 19:15). URL: https://commonmark.org/.

[21] Sean Leonard. *The text/markdown Media Type*. RFC 7763. März 2016, S. 1–15. URL: https://tools.ietf.org/html/rfc7763.

[22] IANA. *Markdown Variants*. (Abruf: 03.01.2020 14:45). März 2016. URL: https://www.iana.org/assignments/markdown-variants/markdown-variants.xhtml.

[23] John MacFarlane u. a. *pandoc Manual*. (Abruf: 05.01.2020 21:50). 2019. URL: https://pandoc.org/MANUAL.html.

[24] John MacFarlane. *Pandoc Lua Filters*. (Abruf: 12.01.2020 20:10). 2019. URL: https://pandoc.org/lua-filters.html.

[25] John MacFarlane. *Pandoc Filters*. (Abruf: 12.01.2020 22:00). 2019. URL: https://pandoc.org/filters.html.

[26] Eric A. Meyer und Kathryn S. Meyer. *S5 1.1 Reference*. (Abruf: 12.01.2020 22:05). URL: https://meyerweb.com/eric/tools/s5/structure-ref.html.

[27] Chris Castle u. a. *Squeenote - A way for nerds to present to other nerds.* (Abruf: 12.01.2020 22:10). Juni 2010. URL: https://github.com/julesfern/Squeenote.

[28] The Daring Fireball Company LLC. *Markdown: Syntax*. (Abruf: 12.01.2020 22:15). URL: https://daringfireball.net/projects/markdown/syntax.

[29] Ethan V. Munson. „Markup Language“. In: *Encyclopedia of Database Systems*. Hrsg. von LING LIU und M. TAMER ÖZSU.

Boston, MA: Springer US, 2009, S. 1696–1696. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_5044. URL: https://doi.org/10.1007/978-0-387-39940-9_5044.

[30] Microsoft Corporation. *[MS-PPT]: PowerPoint (.ppt) Binary File Format.* (Abruf: 20.11.2019 14:30). Feb. 2019. URL: https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-ppt/6be79dde-33c1-4c1b-8ccc-4b2301c08662.

[31] *Information technology — Document description and processing languages — Office Open XML File Formats — Part 1: Fundamentals and Markup Language Reference.* Standard. Geneva, CH: International Organization for Standardization, Nov. 2016.

[32] Facebook Inc. *How do I format my posts with markdown?* (Abruf: 20.11.2019 15:10). 2019. URL: https://www.facebook.com/help/work/541260132750354.

[33] LanterneRougeOG. *New Reddit-flavored Markdown.* (Abruf: 20.11.2019 15:15). https://www.reddit.com/user/LanterneRougeOG/. Jan. 2019. URL: https://www.reddit.com/wiki/markdown.

[34] Github Inc. *About writing and formatting on GitHub.* (Abruf: 20.11.2019 15:20). 2019. URL: https://help.github.com/en/github/writing-on-github/about-writing-and-formatting-on-github.

[35] Stack Exchange Inc. *Markdown Editing Help.* (Abruf: 20.11.2019 15:15). 2019. URL: https://stackoverflow.com/editing-help.

[36] Atlassian Corporation Plc. *Markdown Syntax Guide.* (Abruf: 20.11.2019 15:45). Mai 2018. URL: https://confluence.atlassian.com/bitbucketserver/markdown-syntax-guide-776639995.html.

[37] Gitlab Inc. *Gitlab Markdown.* (Abruf: 20.11.2019 15:30). Nov. 2019. URL: https://docs.gitlab.com/ee/user/markdown.html#gitlab-markdown.

[38] Bob Swift. *Code Pro for Confluence User's Guide- Sections Parameter.* (Abruf: 12.01.2020 22:25). Apr. 2015. URL: https://bobswift.atlassian.net/wiki/spaces/CODE/pages/80543784/Sections+Parameter.

[39] Jobst Hoffmann, Brooks Moses und Carsten Heinz. *The Listings Package.* (Abruf: 12.01.2020 22:25). Sep. 2019. URL: https://ftp.agdsn.de/pub/mirrors/latex/dante/macros/latex/contrib/listings/listings.pdf.

[40] Robert W. Sebesta und Soumen Mukherjee. *Concepts of programming languages.* 10. ed., international ed. Always learning. 815 S. : graph. Darst. Boston: Pearson, 2013. ISBN: 0273769103 and 9780273769101 and 9780273769101. URL: https://suche.suub.uni-bremen.de/peid=B71560610.

[41] Martin Odersky u. a. *Scala Language Specification Version 2.13.* (Abruf: 17.11.2019 18:47). Juni 2019. URL: https://scala-lang.org/files/archive/spec/2.13/.

[42] Simon Peyton Jones.
Haskell 98 language and libraries: the revised report.
Cambridge University Press, 2003.

[43] Jonathan Robie, Michael Dyck und Josh Spiegel.
XML Path Language (XPath) 3.1. (Abruf: 25.01.2020 21:00). März 2017.
URL: https://www.w3.org/TR/xpath-3/.

[44] James Clark und Steve DeRose.
XML Path Language (XPath) Version 1.0. (Abruf: 25.01.2020 21:00).
Nov. 1999. URL: https://www.w3.org/TR/1999/REC-xpath-19991116/.

[45] Simon Marlow und David Waern. *Haddock User Guide.*
(Abruf: 12.01.2020 20:40). 2010.
URL: https://www.haskell.org/haddock/doc/html/ch03s08.html.

[46] Tyler Nienhouse, Cory Veilleux, Aaron S. Hawley u. a.
Style Guide - Scaladoc. (Abruf: 12.01.2020 20:40). Juli 2019.
URL: https://docs.scala-lang.org/style/scaladoc.html.

[47] Doxygen. *Automatic link generation.* (Abruf: 12.01.2020 20:45).
Dez. 2019.
URL: http://www.doxygen.nl/manual/autolink.html#linkfunc.

[48] Nils Mahlstaedt u. a. *cobra - Proof and Code Presentation Framework.*
(Abruf: 07.02.2020 10:15). Feb. 2020.
URL: https://github.com/nilsmahlstaedt/cobra/tree/5170e2ebed938c0dc45aa8f7a93a14b0fc029598.

[49] Microsoft. *Implementations - Language Servers.*
(Abruf: 07.12.219 23:00). Dez. 2019.
URL: https://microsoft.github.io/language-server-protocol/implementors/servers/.

[50] Microsoft. *Language Server Protocol Specification - 3.14.*
(Abruf: 07.12.2019 20:00). Juli 2019.
URL: https://microsoft.github.io/language-server-protocol/specifications/specification-3-14/.

[51] Richard Helm, Erich Gamma, Ralph Johnson u. a.
Design patterns : elements of reusable object-oriented software.
Addison-Wesley professional computing series.
XV, 395 S ; 25 cm : graph. Darst.
Reading, Mass. [u.a.]: Addison-Wesley, 1995.
ISBN: 0201633612 and 9780201633610 and 97802016336110.
URL: https://suche.suub.uni-bremen.de/peid=B18023122.

[52] Patrick Nordwall, Johan Andrén, Rober Stoll u. a.
Akka Documentation - Interaction Patterns - Per session child Actor.
(Abruf: 16.01.2020 15:30). Dez. 2019.
URL: https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html#per-session-child-actor.

[53] Carol M. Barnum. *Usability testing essentials : ready, set...test!*
Online-Ressource (XXII, 382 S.)
Amsterdam [u.a.]: Elsevier, Morgan Kaufmann, c 20112011.
ISBN: 9780123785534 and 9781282879034 and 012375092X and 9780123750921 and 9780123750921. URL:
https://www.sciencedirect.com/science/book/9780123750921.

[54] *Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Measurement of quality in use.*

Standard.
Geneva, CH: International Organization for Standardization, Juni 2016.

[55] MeasuringU. *Implementations - Language Servers.*
(Abruf: 27.12.2019 19:45). 2019.
URL: https://measuringu.com/seq10/.

[56] W. Albert und T. Tullis. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics.* Interactive Technologies.
Elsevier Science, 2010. ISBN: 9780080558264.
URL: https://books.google.de/books?id=KsjpuMJ6T-YC.

[57] Nigel Bevan, Jim Carter, Jonathan Earthy u. a. „New ISO Standards for Usability, Usability Reports and Usability Measures".
In: *Human-Computer Interaction. Theory, Design, Development and Practice: 18th International Conference, HCI International 2016, Toronto, ON, Canada, July 17-22, 2016. Proceedings, Part I.*
Hrsg. von Masaaki Kurosu. Jan. 2016. Kap. 10, S. 266–290.

[58] Dirk Bäumer. *The Language Server Index Format (LSIF).*
(Abruf: 23.01.2020 14:15). Feb. 2019.
URL: https://code.visualstudio.com/blogs/2019/02/19/lsif.

Anhänge

Anhang A

Testmaterialien

A.1 Aufgabe 1

A.1.1 Beispiel A

```
1  <section>
2          <h1>2D Formen</h1>
3  </section>
4
5  <section>
6          <h2>Eine geometrische Form</h2>
7          <img src="hex.png"/>
8  </section>
9
10 <section>
11         <h2>Form</h2>
12         <ol>
13                 <li>Rechteck</li>
14                 <li>Hexagon</li>
15                 <li>Dreieck</li>
16                 <li>Kreis</li>
17         </ol>
18         <p>
```

(a) square.png (b) triangle.png (c) hex.png (d) circle.png

Abbildung A.1: Bilder aus dem Kontext con Aufgabe 1 Beispiel A

```
19                  Die hier gezeigten Zahlen werden auf den
                        ↪ folgenden
20                  Folien als Bezeichner für diese Form
                        ↪ benutzt!
21          </p>
22  </section>
23
24  <section>
25          <h2>Innenzahl</h2>
26          <ol>
27                  <li>1</li>
28                  <li>4</li>
29                  <li>2</li>
30                  <li>3</li>
31          </ol>
32  </seciton>
33
34  <section>
35          <h2>Kantenzahl</h2>
36          <ol>
37                  <li>4</li>
38                  <li>6</li>
39                  <li>3</li>
40                  <li>1 oder sehr viele?!</li>
41          </ol>
42  </section>
```

Listing A.1: Beispiel A in HTML

```
1  # 2D Formen
2
3  ## eine geometrische Form
4  ![a hexagon](hex.png)
5
6  ## Form
7    1. Rechteck
8    2. Hexagon
9    3. Dreieck
10   4. Kreis
11
12 Die hier gezeigten Zahlen werden auf den folgenden
13 Folien als Bezeichner für diese Form benutzt!
14
15 ## Innenzahl
16   1. 1
17   2. 4
18   3. 2
19   4. 3
20
21 ## Kantenzahl
22   1. 4
```

```
23   2. 6
24   3. 3
25   4. 1 oder sehr viele?
```

Listing A.2: Beispiel A in Markdown

A.1.2 Beispiel B

```
1  <section>
2          <h1>Projektdefinitionen</h1>
3  </section>
4
5  <section>
6          <h2> Worum geht's?</h2>
7          <p>
8                  In Cobra werden eine Reihe von Attributen
                       ↪ zur
9                  Definitionen von Projekten genutzt.
10         </p>
11         <p>
12                 Die folgende Tabelle stellt diese vor.
13         </p>
14 </section>
15
16 <section>
17 <h2>Attribute</h2>
18 <table>
19         <tr>
20                 <td>A</td>
21                 <td>B</td>
22                 <td>C</td>
23                 <td>Result</td>
24         </tr>
25         <hr/>
26         <tr>
27                 <td>0</td>
28                 <td>0</td>
29                 <td>0</td>
30                 <td>0</td>
31         </tr>
32         <tr>
33                 <td>0</td>
34                 <td>0</td>
35                 <td>1</td>
36                 <td>0</td>
37         </tr>
38         <tr>
39                 <td>0</td>
40                 <td>1</td>
41                 <td>0</td>
```

```
42                      <td>1</td>
43              </tr>
44              <tr>
45                      <td>0</td>
46                      <td>1</td>
47                      <td>1</td>
48                      <td>0</td>
49              </tr>
50              <tr>
51                      <td>1</td>
52                      <td>0</td>
53                      <td>0</td>
54                      <td>1</td>
55              </tr>
56              <tr>
57                      <td>1</td>
58                      <td>0</td>
59                      <td>1</td>
60                      <td>0</td>
61              </tr>
62              <tr>
63                      <td>1</td>
64                      <td>1</td>
65                      <td>0</td>
66                      <td>1</td>
67              </tr>
68              <tr>
69                      <td>1</td>
70                      <td>1</td>
71                      <td>1</td>
72                      <td>0</td>
73              </tr>
74  </table>
75  </section>
```

Listing A.3: Beispiel B in HTML

```
1  # Projektdefinitionen
2
3  ## Worum geht's
4  In Cobra werden eine Reihe von Attributen zur
5  Definitionen von Projekten genutzt.
6
7  Die folgende Tabelle stellt diese vor.
8
9  ## Attribute
10 | A | B | C | Result |
11 |---|---|---|--------|
12 | 0 | 0 | 0 | 0      |
13 | 0 | 0 | 1 | 0      |
14 | 0 | 1 | 0 | 1      |
```



(a) Beispiel 1

(b) Beispiel 2

(c) Beispiel 3

(d) Beispiel 4

Abbildung A.2: Zuzuordnende Strukturen aus Aufgabe 4

```
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
```

Listing A.4: Beispiel B in Markdown

A.2 Aufgabe 4

```
<section>
        <h1>A</h1>
</section>

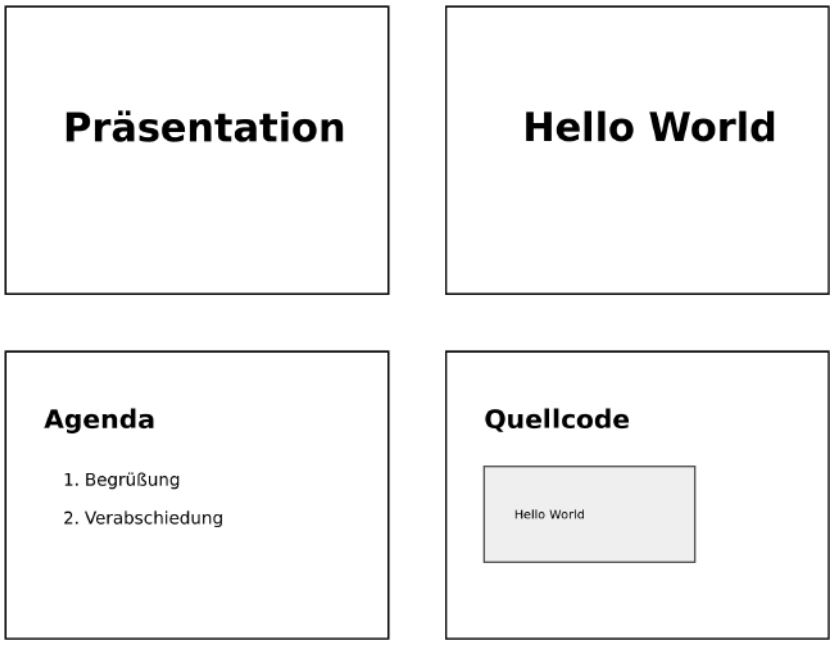
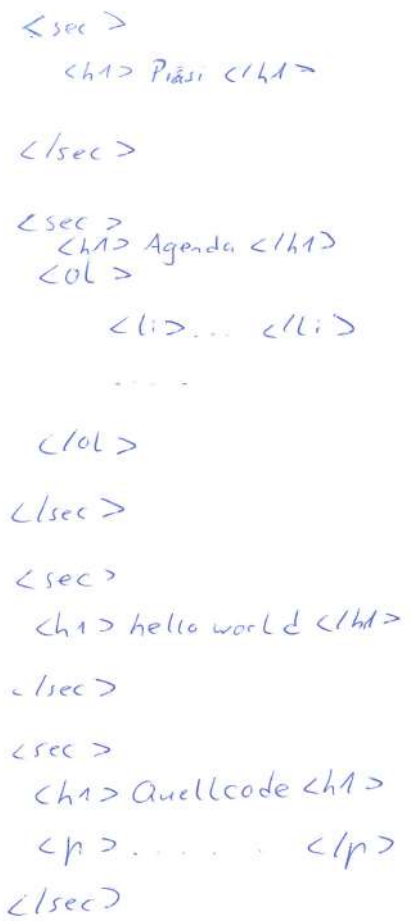
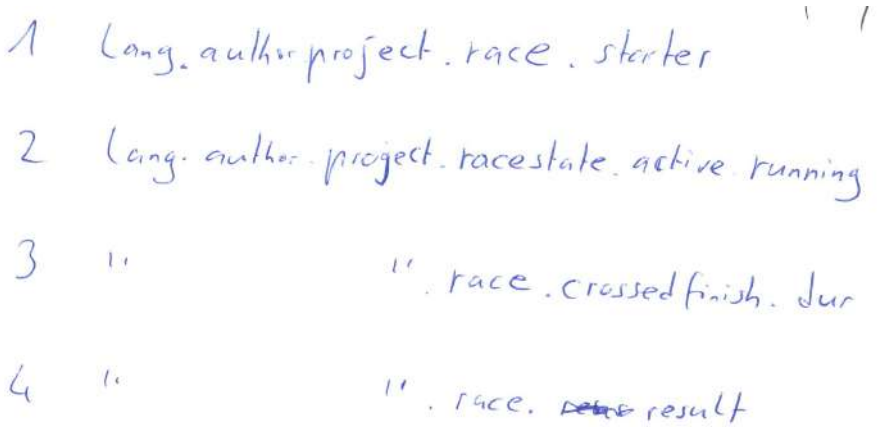
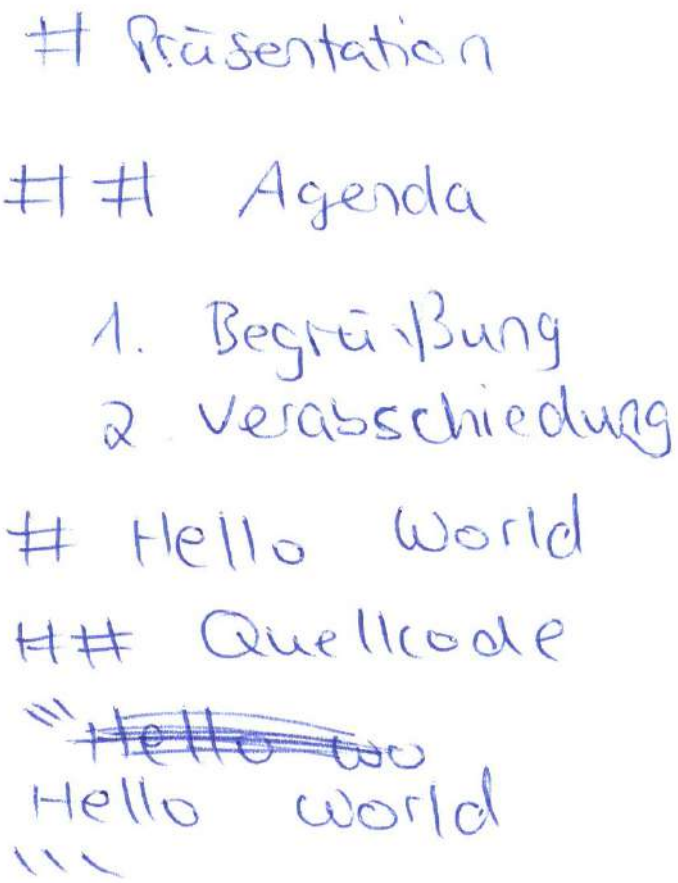
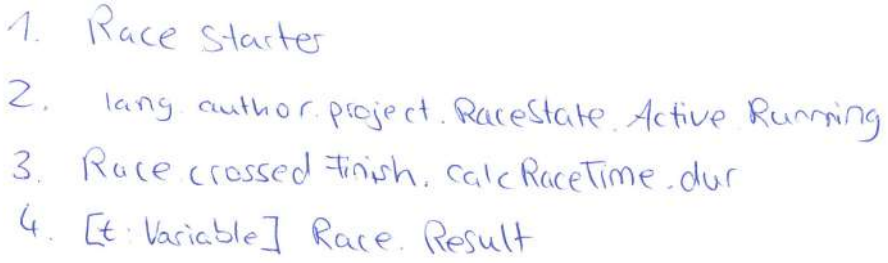
<section>
        <h1>B</h1>
</section>

<section>
        <h1>C</h1>
</section>

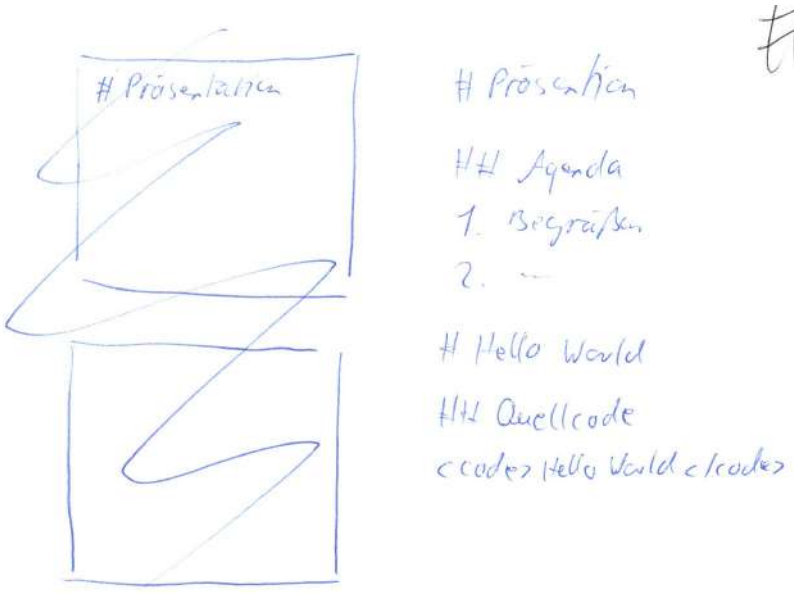
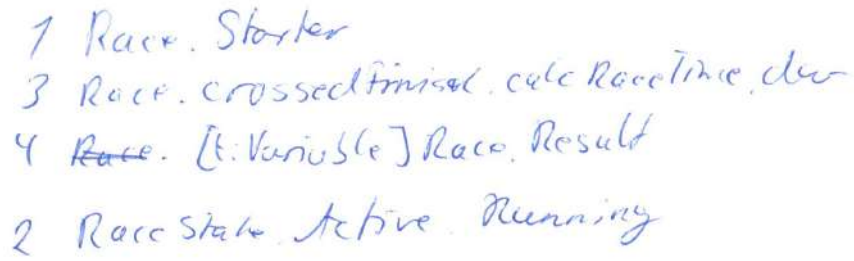
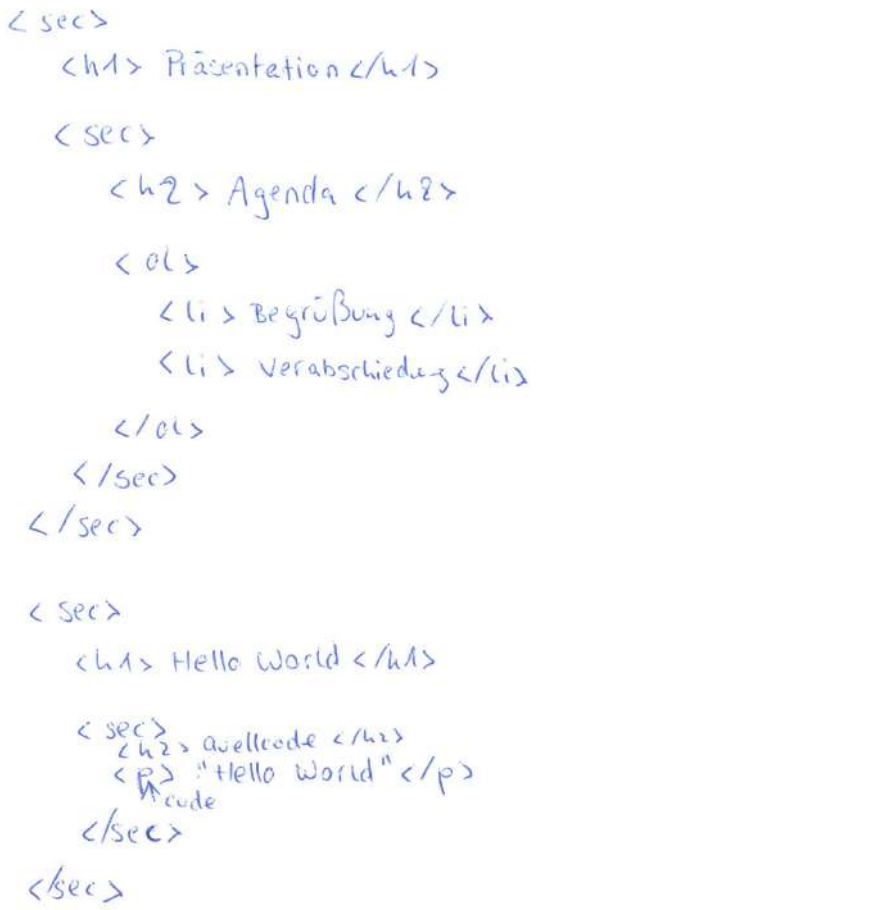
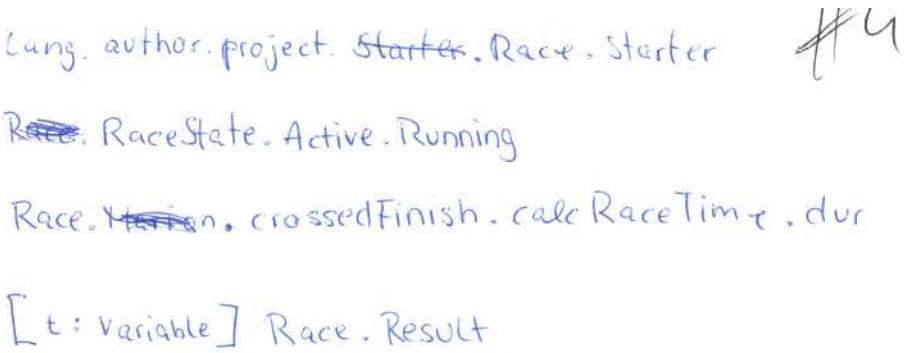
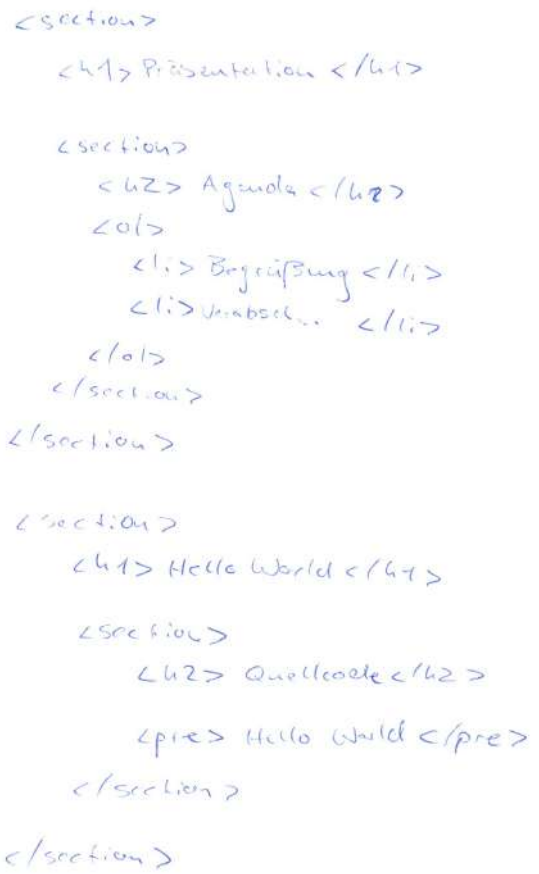
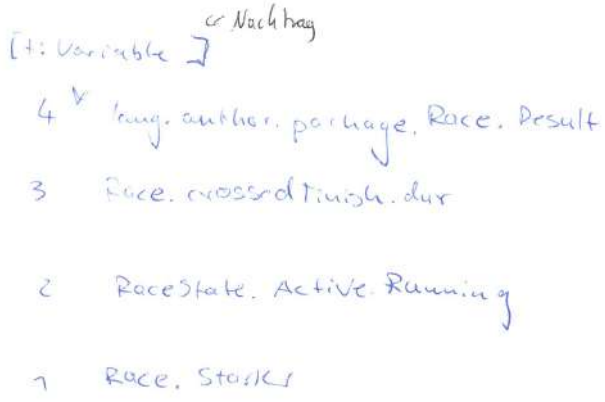
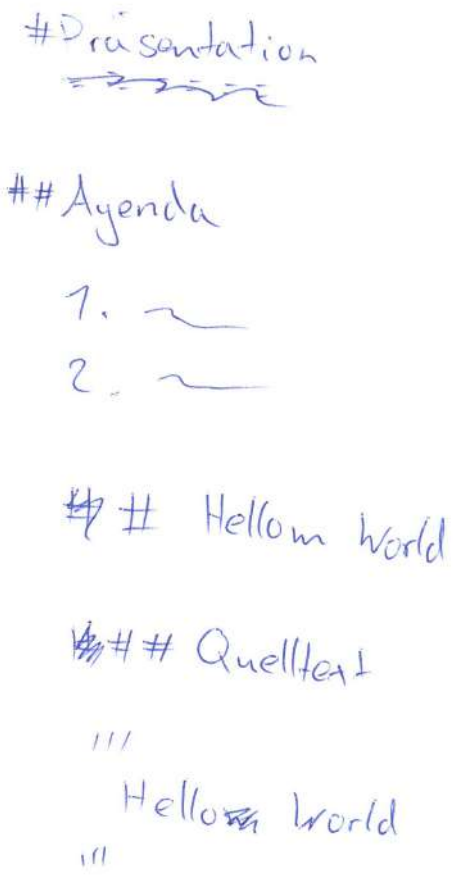
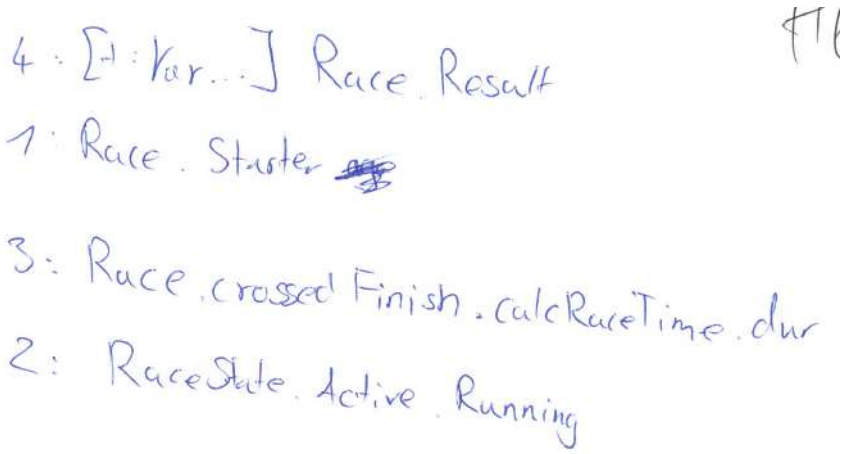
<section>
        <h1>D</h1>
</section>

<section>
        <h1>E</h1>
</section>

<section>
        <h1>F</h1>
```

```
23 </section>
```

Listing A.5: Aufgabe 4 Beispiel 1 in HTML

```
1  # A
2
3  # B
4
5  # C
6
7  # D
8
9  # E
10
11 # F
```

Listing A.6: Aufgabe 4 Beispiel 1 in Markdown

```
1  <section>
2          <h1>A</h1>
3
4          <section>
5                  <h2>B</h2>
6          </section>
7
8          <section>
9                  <h2>C</h2>
10         </section>
11
12         <section>
13                 <h2>D</h2>
14         </section>
15
16         <section>
17                 <h2>E</h2>
18         </section>
19
20         <section>
21                 <h2>F</h2>
22         </section>
23 </section>
```

Listing A.7: Aufgabe 4 Beispiel 2 in HTML

```
1  # A
2
3  ## B
4
5  ## C
6
7  ## D
8
```

```
9  ## E
10
11 ## F
```

Listing A.8: Aufgabe 4 Beispiel 2 in Markdown

```
1  <section>
2          <h1>A</h1>
3
4          <section>
5                  <h2>D</h2>
6          </section>
7
8          <section>
9                  <h2>E</h2>
10         </section>
11 </section>
12
13 <section>
14         <h1>B</h1>
15 </seciton>
16
17 <section>
18         <h1>C</h1>
19
20         <section>
21                 <h2>F</h2>
22         </section>
23 </seciton>
```

Listing A.9: Aufgabe 4 Beispiel 3 in HTML

```
1  # A
2
3  ## D
4
5  ## E
6
7  # B
8
9  # C
10
11 ## F
```

Listing A.10: Aufgabe 4 Beispiel 3 in Markdown

```
1  <section>
2          <h1>A</h1>
3
4          <section>
5                  <h2>B</h2>
6          </section>
```

```
7  </seciton>
8
9  <section>
10         <h1>C</h1>
11
12         <section>
13                 <h2>D</h2>
14         </section>
15 </seciton>
16
17 <section>
18         <h1>E</h1>
19
20         <section>
21                 <h2>F</h2>
22         </section>
23 </seciton>
```

Listing A.11: Aufgabe 4 Beispiel 4 in HTML

```
1  # A
2
3  ## B
4
5  # C
6
7  ## D
8
9  # E
10
11 ## F
```

Listing A.12: Aufgabe 4 Beispiel 4 in Markdown

A.3 Aufgabe 7

```
1  <section>
2          <h2> A Slide !</h2>
3          <p>
4                  Lorem ipsum dolor sit amet, consectetuer
5                  adipiscing elit. Aenean commodo ligula
6                  eget dolor. Aenean massa. Cum sociis
7                  natoque penatibus et magnis dis
8                  parturient montes, nascetur ridiculus mus.
9                  Donec quam felis, ultricies nec,
10                 pellentesque eu, pretium quis, sem.
11         </p>
12 </section>
```

Listing A.13: Aufgabe 7 Beispiel 1 in HTML

(a) Beispiel 1



(b) Beispiel 2

(c) Beispiel 3



(d) Beispiel 4

Abbildung A.3: Zuzuordnende Slides aus Aufgabe 7

```
## A Slide !

Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Aenean commodo ligula eget
dolor. Aenean massa. Cum sociis natoque penatibus
et magnis dis parturient montes, nascetur
ridiculus mus. Donec quam felis, ultricies nec,
pellentesque eu, pretium quis, sem.
```

Listing A.14: Aufgabe 7 Beispiel 1 in Markdown

```
<section>
        <h2> A Slide !</h2>
        <ul>
                <li>
                        Lorem ipsum dolor sit amet, c
                        onsectetuer adipiscing elit.
                </li>
                <li>
                        Aenean commodo ligula eget dolor.
                        Aenean massa.
                </li>
                <li>
                        Cum sociis natoque penatibus et
                        magnis dis parturient montes,
                </li>
                <li>
                        nascetur ridiculus mus.
                </li>
```

```
19                  <li>
20                          Donec quam felis, ultricies nec,
21                          pellentesque eu, pretium quis, sem.
22                  </li>
23          </ul>
24  </section>
```

Listing A.15: Aufgabe 7 Beispiel 2 in HTML

```
1  ## A Slide !
2
3    - Lorem ipsum dolor sit amet, consectetuer
4          adipiscing elit.
5    - Aenean commodo ligula eget dolor. Aenean massa.
6    - Cum sociis natoque penatibus et magnis dis
7          parturient montes,
8    - nascetur ridiculus mus.
9    - Donec quam felis, ultricies nec,
10         pellentesque eu, pretium quis, sem.
```

Listing A.16: Aufgabe 7 Beispiel 2 in Markdown

```
1  <section>
2          <h2> A Slide !</h2>
3          <code class="plaintext">
4                  Lorem ipsum dolor sit amet,
5                  consectetuer adipiscing elit.
6                  Aenean commodo ligula eget dolor.
7                  Aenean massa.
8                  Cum sociis natoque penatibus et magnis
9                  dis parturient montes,
10                 nascetur ridiculus mus.
11                 Donec quam felis, ultricies nec,
12                 pellentesque eu, pretium quis, sem.
13         </code>
14 </section>
```

Listing A.17: Aufgabe 7 Beispiel 3 in HTML

```
1  ## A Slide !
2
3  ```plaintext
4          Lorem ipsum dolor sit amet,
5          consectetuer adipiscing elit.
6          Aenean commodo ligula eget dolor.
7          Aenean massa.
8          Cum sociis natoque penatibus et magnis dis
9          parturient montes,
10         nascetur ridiculus mus.
11         Donec quam felis, ultricies nec, pellentesque
12         eu, pretium quis, sem.
```

```
13 ```
```

Listing A.18: Aufgabe 7 Beispiel 3 in Markdown

```
1 <section>
2         <h2> A Slide !</h2>
3
4         <image src="the_alphabet_grid.png" />
5 </section>
```

Listing A.19: Aufgabe 7 Beispiel 4 in HTML

```
1 ## A Slide !
2
3 ![alphabet_grid](the_alphabet_grid.png)
```

Listing A.20: Aufgabe 7 Beispiel 4 in Markdown

A.4 Aufgabe 11



Abbildung A.4: Skizze von Folien zur Umsetzung in HTML oder Markdown durch Teilnehmer

A.5 Aufgabe 14

Pfade welche in einer ausgedruckten der Quellcodes markiert werden müssen

1. `lang.author.project.TaxCalculator.getStateTax.getZipCode`

2. `MyClass.calculateEarnings.bruttoReward`
3. `MyClass.Job1.security`
4. `[t:Variable] Myclass.name`

```
1  package lang.author.project
2
3  class MyClass {
4    var name: Name = "Max␣Mustermann"
5    var address = "Musterweg␣4"
6    var occupation: Occupation = null
7
8    type name = String
9
10   abstract class Occupation {
11     def hours: Int
12   }
13
14   class Job1 extends Occupation {
15     var name = "some␣job"
16     var hours = 12
17     var security = JobSecurity.High
18   }
19
20   class Hobby extends Occupation {
21     var name = "water␣skiing"
22     var hours = 3
23     var funFactor = 4
24   }
25
26   /**
27    * calculate potential reward for doing your occupation
        ↪  with a rate of reward
28    * @param reward earned per hur of doing occupation
29    * @return total earnings based on reward and hour
        ↪ spend with occupation
30    */
31   def calculateEarnings(reward: Int): Int ={
32     var earningsPerDay = 24 * reward
33     var dayShareOfOccupation = this.occupation.hours / 24
34     var bruttoReward = earningsPerDay *
         ↪ dayShareOfOccupation
35     var taxes = TaxCalculator.getStateTax(this.address,
         ↪ bruttoreward)
36
37     return (nettoReward - taxes)
38   }
39
40   override def toString(): name = name+"␣spends␣his␣time␣
       ↪ with:␣"+occupation
```

```
41 }
42
43 object TaxCalculator {
44   def getStateTax(address: String, bruttoIncome: Int):
       ↪ Int = {
45     def getZipCode(addr: String): Int = {28209}
46
47     def getTaxBracket(income: Int): Float = {0.5}
48
49     def calcDeduction(zip: Int, bracket: Float): Int =
         ↪ {42}
50
51     return calcDeduction(getZipCode(address),
         ↪ getTaxBracket(bruttoIncome))
52   }
53
54   override def toString(): String = "This␣is␣a␣Tax␣
       ↪ Calculator!"
55 }
```

Listing A.21: Aufgabe 14 Quellcode

A.6 Aufgabe 16

Zeilen Ranges zu denen Pfade erstellt werden müssen

1. LL 15 - 18
2. L 75
3. L 57
4. L 13

```
1  package lang.author.project
2
3  class Race {
4    type Time = Long
5    type Result = (Time, Starter)
6
7    var startTime: Time = 0
8    var raceState: RaceState.State = RaceState.Planned
9    var minAge = 20
10   var maxAge = 28
11
12   var starters: Set[Starter] = Set.empty
13   var Result: Set[Result] = Set.empty
14
15   abstract class Starter() {
16     var name: String
17     var age: Int
```

```
18    }
19
20    class Mike() extends Starter() {
21      var name = "mike"
22      var age = 24
23    }
24
25    class Marion() extends Starter() {
26      var name = "marion"
27      var age = 26
28    }
29
30    /**
31     * adds participant to race
32     * if age requirements are met
33     */
34    def addParticipant(s: Starter) = {
35      if(s.age < minAge || s.age > maxAge){
36        println("could not add starter. Reason: Not the
            ↪ right category!")
37      }else{
38        var updatedStarters = this.starters + s
39        this.starters = updatedStarters
40      }
41    }
42
43    /**
44     * starts race
45     */
46    def startRace(startTime: Time) = {
47      this.startTime = startTime
48      this.raceState = RaceState.Active.Running
49    }
50
51    /**
52     * records result for starter that crossed finish line
            ↪ at currentTime
53     * calculates run duration
54     */
55    def crossedFinish(s: Starter, currentTime: Time) = {
56      def calcRaceTime(sTime: Time, cTime: Time): Time = {
57        var dur = cTime - sTime
58        return dur
59      }
60
61      var starterResult = (calcRaceTime(this.startTime,
            ↪ currentTime), s)
62      var updatedResults = this.result + starterResult
63      this.Result = updatedResults
64    }
```

```
65 }
66
67 object RaceState {
68   abstract class State
69   object Planned extends State
70
71   object Active {
72     object AtStartBlocks extends State
73     object RowCall extends State
74     object Running extends State
75   }
76
77   object Finished extends State
78 }
```

Listing A.22: Aufgabe 14 Quellcode

Anhang B

Testergebnisse

B.1 Teilnehmende

Tabelle B.1 gibt einen Überblick über alle Teilnehmenden, die die Evaluation absolviert haben.

P_n	**Hintergrund**	**HTML-Kenntnis**	**Markdown (MD)-Kenntnis**	**reveal.js-Kenntnisse**
1	PHP Entwickler_in Full Stack	sehr gut	vorhanden	nein
2	Java Entwickler_in Full Stack	sehr gut	eigenen Website Content in md verfasst	nein
3	Frontend Entwickler_in	sehr gut	technische Dokumentation und Tickets	ja
4	Azubi Informatik (2. LJ) Schwerpunkt PHP	sehr gut	Dokumente für Berufsschule	nein
5	Frontend Entwickler_in	sehr gut	Notizmittel der Wahl	ja
6	Frontend Entwickler_in	sehr gut	technische Dokumentation	nein
7	Azubi Informatik (2. LJ) Schwerpunkt Frontend	sehr gut	„nicht wirklich“	nein
8	Master Student_in Digitale Medien	gut	nein	nein
9	Master Student_in Informatik	gut	einfache Readmes	nein
10	Master Student_in Informatik	„etwas eingestaubt“	Readmes und Uni-Abgaben	nein
11	Master Student_in Informatik	grundlegend	Notizen und Uni-Abgaben	nein
12	Master Student_in Informatik	gering	Gitlab Wiki's und Uni-Abgaben	nein

Tabelle B.1: Teilnehmende der Evaluation

B.2 Fragen Zuordnung

Im den Ergebnisprotokollen der Teilnehmenden werden, um Platz zu sparen, Fragen nur durch ihre Nummer referenziert. Tabelle B.2 gibt eine Übersicht über die Fragen und im Folgenden an ihrer Stelle verwendeten Nummern.

#	**Aufgabe / Frage**
1	5.2.2 Lesbarkeit von Folien:
2	Wie schwer war es für dich den Inhalt und die Struktur der Folien zu verstehen?
3	Welches Format ist für dich einfacher zu lesen?
4	5.2.2 Zuordnung von Folienstrukturen:
5	Wie schwer fandest du diese Aufgabe. Was waren Probleme? Was waren Erkenntnisse?
6	Wie sicher hast du dich bei der Zuordnung gefühlt?
7	5.2.2 Zuordnung von Einzelfolien:
8	Wie schwer fandest du diese Aufgabe. Was waren Probleme? Was waren Erkenntnisse?
9	Wie sicher hast du dich bei der Zuordnung gefühlt?
10	Wie sicher fühlst du dich im Verständnis von Folien ihrem Aufbau und ihrer Anordnung?
11	5.2.2 Erstellen von Folien:
12	Wie war es für dich diese Folien zu schreiben? Gab es Hürden? Was ging einfach?
13	Hast du dich effizient bei der Umsetzung der Aufgabe gefüllt?
14	5.2.2 Auflösen von Pfaden:
15	Gab es Hürden, Verständnisprobleme oder Ähnliches? Wenn ja, welche?
16	5.2.2 Erstellen von Pfaden:
17	Gab es Hürden, Verständnisprobleme oder Ähnliches? Wenn ja, welche?

Tabelle B.2: Zuordnung von Fragen und Antworten in den Testergebnissen

B.3 Ergebnisprotokolle

Teilnehmer*in 1

Teilnehmerinformationen

Teilnehmer	1
Background	PHP Entwickler
Markdown Kenntnisse	grundlegende Kenntnisse
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	nein

Reihenfolge 1	A (html)	B (md)		
Syntaxbeispiele 4	1 html	2 md	3 html	4md
Syntaxbeispiele 7	1 html	2 md	3 html	4 md
Syntax 11	html			

Testparameter

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.html)	90	1	
1 (B.md)	34	1	
4	50	2	1
7	12	1	
11	130	2	1
14	60	1	0
17	160	2	2

Beobachtungen

1. Verständnis von Aufgabe A schwierig
2. "beide Formate sind einfach zu verstehen"
3. HTML bevorzugt, ist als PHP-Entwickler geläufiger als MD
4. Unterschied zwischen Beispiel 1&2 schwierig

   Fehler:
   - senkrechte und waagerechte Anordnungen (1&2) vertauscht
     Korrektur nach Hinweis sofort möglich
5. Zuordnung nicht schwer
   "3 war am einfachsten, es war am ehesten aus dem HTML
   zu sehen"
6. bei HTML Beispielen sicher,
   bei MD Beispielen etwas unsicherer
7.
8. überhaupt nicht schwer
9. bei allen ziemlich sicher
10. "jetzt nach Aufgaben [4 & 7] ziemlich gutes Verständnis"
    "wenn man erstmal das Raster mit dem Gitter
    [revealJS Folienanordnung] verstanden hat"
11. Anordnung der Folien in einem Gitter
    nicht einfach in der Umsetzung

```
Fehler:
 - Schließende Section der ersten Folie (Präsentation)
   vergessen und dadurch falsches Gitter erzeugt
   Korrektur nach Hinweis auf Fehler sofort möglich
```

12. `Gridanordnung war nicht intuitiv`
13. `ja, habe mich effizient gefühlt`
14. `Fehler:`
    `- 3. Pfad Element 'calcRaceTime' vergessen`
    `- 4. [t: Angabe vergessen`

    `Beide Fehler konnten in einem Versuch nach Hinweis auf Fehler korrigiert werden`
15. `"Pfade waren logisch"`
    `"einmal durch die Datei lesen, dann war die Aufgabe einfach zu lösen"`
    `"[t: Syntax war unverständlich`
16.
17. `[t: Syntax unverständlich`
    `das restliche Konzept aber verständlich`

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	4

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 2

Teilnehmerinformationen

Teilnehmer	2
Background	Java Fullstack Entwickler
Markdown Kenntnisse	gut, eigene Website mit Markdown und Static Ste Generator
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	A (md)	B (html)		
Syntaxbeispiele 4	1 html	2 md	3 html	4md
Syntaxbeispiele 7	1 html	2 md	3 html	4 md
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.md)	39	1	
1 (B.html)	40	1	
4	35	1	0
7	30	1	0
11	75	1	0
14	37	1	0
17	60	1	2

Beobachtungen

1.
2. `einfach für beide Beispiele`
3. `Markdown schwieriger zu lesen`
   `"HTML durch tagtägliche Arbeit bekannter"`
   `"HTML hat mehr Struktur"`
4.
5. `Zuordnung war einfach`
6. `sehr sicher`
7.

8. `Zuordnung einfach`

   `"Es ist einfach, wenn man die Syntax der`
   `Elemente [Bilder, Code Blöcke, Listen] kennt"`

9. `sicher`
10. `Verständnis von Folien ist besser als das Verständnis`
    `von der Anordnung von Folien in einem Slide Deck`

    `Zuordnung ist einfach, selbst Aufbauen erscheint`
    `noch als komplexe Aufgabe`

11. `Code Fence Syntax für Code Blöcke war nicht bekannt`
    `und musste erfragt werden`
12. `Es gab bei der Erstellung von Folien keine Probleme`
    `oder Herausforderungen`
13. `definitiv effizient bei der Erstellung gefühlt`
14.
15. `Auflösen von Pfaden war einfach`

    `"über die [t:Variable] Sache bin ich gestolpert,`
    `habe mir aber etwas in die Richtung [Festlegen von Typen]`
    `gedacht"`

16.
17. `"Nach Erklärung von [t: war Nutzung klar"`

    `"Definition von Beispielen`
    `hat sich natürlich angefühlt"`

    `Es war nicht klar, wann die Package Definition im Pfad`
    `notwendig war und wann nicht`

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	4

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 3

Teilnehmerinformationen

Teilnehmer	3
Background	Frontend Entwickler
Markdown Kenntnisse	gut, technische Dokumentation, Readmes und Tickets
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	ja

Testparameter

Reihenfolge 1	A (md)	B (html)		
Syntaxbeispiele 4	1 html	2 html	3 md	4 md
Syntaxbeispiele 7	1 html	2 html	3 md	4 md
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.md)	80	1	
1 (B.html)	46	1	
4	90	1	0
7	25	1	0
11	135	2	1
14	50	1	0
17	210	1	0

Beobachtungen

1. `Wahrscheinlichkeitstabelle in B.html war nicht bekannt und musste als Konzept erklärt werden`
2. `"Formulierung der Aufgabe und Folien in A.md war komisch"`

   `Beziehung zwischen Slide "Innenzahl" und den in den Formen gezeichneten Zahlen war unklar`

   `Struktur der Tabelle sofort klar und hat keine Verständnisschwierigkeit durch html gegeben`

3. Markdown ist einfacher zu lesen

   Tags in HTML stören den visuellen Eindruck
   Tabelle sehr lang

4. 3.md und 4.md durch Aufbau der Folien direkt klar zu erkennen.

   1.html und 2.html durch ähnlichen Aufbau schwer voneinander
   zu unterschieden und zuzuordnen.

5. Aufgabe nicht schwierig

   1&2 währen ohne weitere Beispiele sehr schwierig zuzuordnen
   Einrückung von HTML war bekannt und hat beim Lösen geholfen

6. bei Zuordnung von 3&4 absolut sicher

7. Liste in md sofort als solche erkannt und zugeordnet
   3.md über Dateiname des Bildes sofort erkannt
   2.html über bekannte <code> Syntax erkannt und zugeordnet

8. Aufgabe war einfach
   Bild nur über Dateiname erkannt und zugeordnet,
   nicht über Syntax
   Bild und Codeblock Folie nicht intuitiv voneinander unterscheidbar

9. 100% sicher

10. "Ich denke ich bin mir ziemlich sicher"
    "Relation von Folien und Unterfolien noch nicht sicher"
    das 2D-Grid für Slides ist ungewohnt
    "schreiben in HTML und Markdown ist kein Problem"

11. Die Konvertierungsregeln von Markdown Text zu Folien
    sind nicht intuitiv

    Fehler:
     - Jede Slide einzeln aufgeschrieben und mit Blöcken umrahmt

    Erklärung der Konvertierungsregeln von Markdown zu Folien und
    Erklärung, dass Folien in einem Text zu schreiben sind Versuch 2
    (Zeit aus währendessen gestoppt)

    Markdown Codeblock Syntax nicht bekannt ->
    stattdessen <code> verwendet

12.

13. Konvertierungsregeln (Markdown -> HTML) waren unklar

14.

15. Initiales Verständnis: Pfad greift auf zugewiesene
    Werte der Variable zu
    -> Problem bei Pfad auf Funktionsdefinition

```
-> Definitionen wurden intuitiv trotzdem wie Variablen behandelt

[Klarifizierung der Funktion als
Definition von Quellcodeabschnitten]
"Schachtelungen gibt es in jeder Sprache"
```

16. ```
    "Schachtelung für 1,3,4 müssen mit 'Race' beginnen"
    "Codestelle für 4 sieht genauso aus [wie 14 Pfad 4],
    deshalb beginne ich auch mit [t:"
    "Ich weiß nicht, ob immer ein [t: davor gebraucht wird"

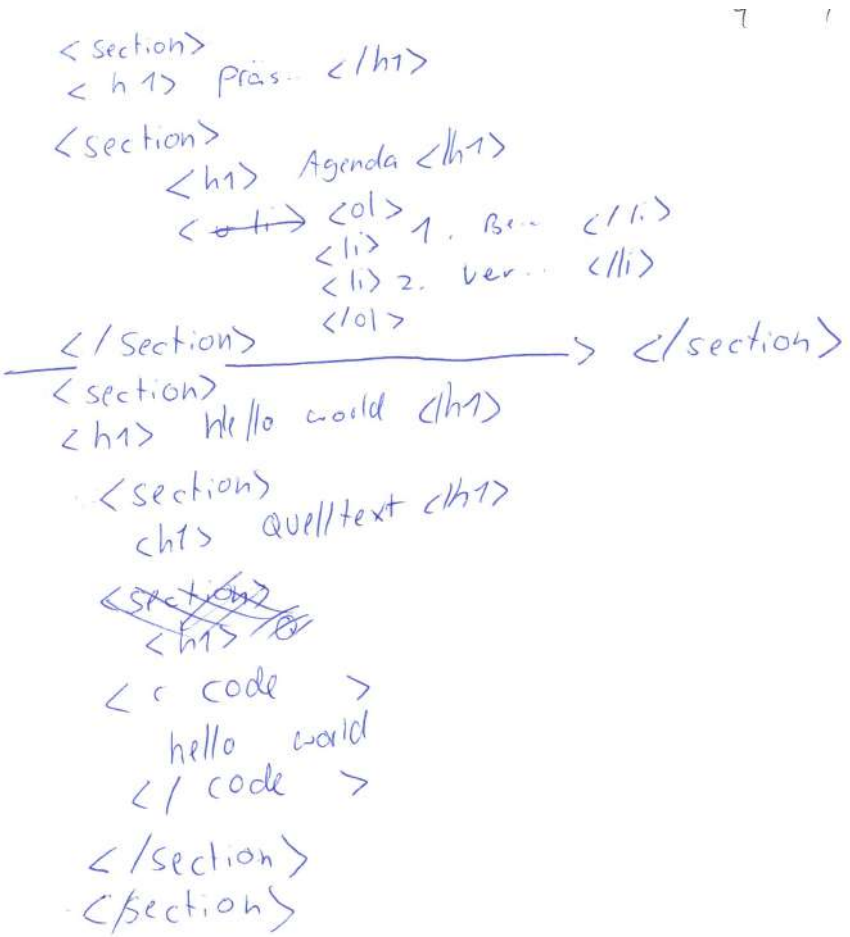
    Erzeugt Pfade entlang der Stufen der Einrückung
    ```

17. ```
    Bedeutung von [t: Syntax ist nicht logisch ersichtlich
    Unsicherheit über Bedeutung von 'object' und 'def' Keywords
    in Scala Beispiel
    ```

Ergebnisse



Lösung Aufgabe 11

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	4

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 16

Teilnehmer*in 4

Teilnehmerinformationen

Teilnehmer	4
Background	Azubi Informatik (2. LJ), Schwerpunkt PHP
Markdown Kenntnisse	grundlegende Kenntnisse, einfache Dokumente für Schule
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	B (html)	A (md)		
Syntaxbeispiele 4	1 html	2 md	3 md	4 html
Syntaxbeispiele 7	1 html	2 md	3 md	4 html
Syntax 11	html			

Zeiten, Versuche und Fehler

Beobachtungen

1. `(B.html) Wahrheitstabelle als Konzept nicht bekannt und hat zu Verständnisproblemen bei der Aufgabe geführt`

   `(A.md) Struktur des Inhalts ist klar, Fehlersuche nur auf Section "Innenzahl" begrenzt. "Wie soll ich anfangen zu suchen?"`

2. `Bei A.md schnell Aufteilung der Sections verstanden Inhaltlicher Zusammenhang der Folien unklar`

Aufgabe	Zeit	Versuche	Fehler
1 (B.html)	240	4	
1 (A.md)	40	1	
4	75	1	0
7	35	1	0
11	175	1	2
14	240	2	1
17	630	2	2

"bei [B.html] hat mir das Verständnis für
Wahrheitstabellen gefehlt
und wie ich sie lesen muss"

3. Markdown ist übersichtlicher
   enthält keine Tags
4. zuerst Beispiel 4.html bearbeitet
   und an diesem Beispiel Schachtelung
   von Sections verstanden

   Danach Konzept von Sections und Untersections
   für restliche Beispiele, inklusive 1&2 klar

   Zuordnung der Markdown Beispiele ebenfalls kein Problem
5. nicht so schwer
6. sehr sicher bei Zuordnung
   Schachtelung in HTML und MD sichtbar
   Genaue Konvertierungsregeln von Markdown zu
   Folien noch nicht ganz klar
7. Über bekannte Syntax von Codeblock 2.md zugeordnet
   Bild an Dateinamen erkannt
8. nicht schwer
   Folien im Quellcode einfacher zu unterscheiden als
   Foliensätze aus Überschriften

   Codeblock durch Verwendung von MD zur
   Formatierung von Nachrichten
   im firmeninternen Chattool bekannt
9. "ich war mir sicher!"
10. "Verständnis von Folien ist nun [nach Aufgaben 4&7] klarer
    als nach der ersten Aufgabe"

    "Ab den Zuordnungsaufgaben habe ich [den Aufbau von Folien]
    besser verstanden"
11. Fehler:

```
 - <h2>Quellcode</h2> Überschrift vergessen
 - <p> statt <code> für Codeblock verwendet
```

12. "das Schreiben ist sehr strukturiert"
    "Struktur hat sich gut eingeprägt,
    der Rest ist einfaches HTML"

13. "ich vermute in Markdown wäre es
    schneller für mich zu schreiben gewesen"

14. Bei Beispiel 1 Pfad von Object-Def aus "entlang gelaufen"
    "diese [t:Variable] verstehe ich nicht, ..., ah Moment"
    -> hat inferiert, dass 't:' Type bedeutet

    Fehler:
     - statt Variablendefinition Typdefinition ausgewählt

15. "die ersten drei waren einfach"
    "ich bin einfach den Pfad entlang gelaufen"
    "ich wusste nicht, was ich mit [t:Variable] anfangen sollte"

16. Fehler:
     - lang.author.projekt.Starter (Race Pfad Element vergessen)
     - Race.RaceState...

    Nach Hinweis auf Fehler:
    Nochmal als fehlerhaft aufgezeigte Pfade zur Kontrolle
    durchgegangen und festgestellt, dass Einrückung des
    Quellcodes nicht korrekt gesehen wurde und
    Fehler korrigiert

17. Einrückungen im vorgelegten Quelltext (2 Spaces) schwer zu lesen
    "War davon ausgegangen,
    dass 'object RaceState' in Klasse sein muss"

    "Ich wusste nicht, ob für 4 't' der richtige Buchstabe ist
    und habe geraten"

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 5

Teilnehmerinformationen

Teilnehmer	5
Background	Frontend Entwickler
Markdown Kenntnisse	mein Notizmittel
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	ja

Testparameter

Reihenfolge 1	B (html)	A (md)		
Syntaxbeispiele 4	1 md	2 md	3 html	4 html
Syntaxbeispiele 7	1 md	2 md	3 html	4 html
Syntax 11	html			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (B.html)	10	1	
1 (A.md)	35	1	
4	40	1	0
7	20	1	0
11	300	1	0
14	240	2	1
17	300	2	1

Beobachtungen

1. `Beide Aufgaben ohne Probleme gelöst`
2. `"HTML ist einfacher für mich zu lesen, es ist strukturierter"`
   `"Tabellen in Markdown können echt mist formatiert sein"`

   `Inhalt von Aufgabe A nicht klar verständlich`
3. `"nach meinem Bauchgefühl ist die Tabelle minimal besser"`

   `Sectionaufbau war logisch`
   `"Grid, in dem sich Folien anordnen nicht logisch"`
4. `"Intuitiv verlaufen Slides horizontal"`
   `nach dieser Aussage 1.md und 2.md korrekt zugeordnet`

5. "HTML Beispiele waren sehr klar durch explizite
   Schachtelung der Sections"

   "Meine Annahme war, dass die einfachste Markdown Variante
   default Slides, also horizontal angeordnete erzeugt"

6. "Die vier Beispiele ließen sich in zwei offensichtliche
   Gruppen [1&2, 3&4] aufteilen"

   "3 & 4 waren ziemlich eindeutig"

   "1 & 2 nicht so sehr"

7.

8. "man muss schon wissen, was ’‘‘‘’ macht und
   dass ’plaintext’ für das Highlighting ist"

9. "ich war mir beim Zuordnen absolut sicher"

10. hat sicheres Verständnis
    "aber nur weil ich schon vorherige Erfahrungen
    mit reveal.js hatte"

11. "Ich bin mir nicht sicher, wie die Konventionen für <h1>
    und <h2> hier sind"

12. "musste das einfach nur runterschreiben"
    "Knackpunkte für mich waren:
     1. wie funktioniert genau die Grid Beziehung
     2. ist ein Codeblock nach Konvention hier
        <pre> oder <pre><code>"

13. "HTML ist für mich das Optimum der guten Struktur wegen"

    "es ist für mich einfacher zu erkennen und hat eine
    logische Struktur"

14. "Ich habe ’package ...’ erstmal als
    Metazeile für Compiler ignoriert"

    "[t: muss man sich zusammenreimen"
    "Mein Gedanke ist, dass ’t’ Typ Variable bedeutet"

    Fehler:
     - 4. zu Typ Definition ausgelöst

    Nach Hinweis zu anderer Option (Var-Definition) aufgelöst
    "Immer noch unklar, was das bedeutet"

15. [Erklärung der Bedeutung von t:]

    "das ist ja eigentlich nur XPath für Java
    mit Extension für Variable"

"für den Teil [t: Syntax] wäre ich in die Doku gegangen"

16. bei 2. komplettes Package wie aus 14.1 übernommen
    "kann ich mir das mit dem Package aussuchen?"

    'object' Keyword war unbekannt und hat verunsichert
    "Wie zeige ich jetzt auf einen Zeilenbereich statt
    auf eine einzelne Zeile?"

    Fehler:
     - [t:Variable] bei 4. vergessen

17. Text ist zu wenig eingerückt

    das Lesen von Code auf Papier ist schwierig:
    "Editor markiert sonst Klammern und hat Hilfslinien
    für Einrückung"

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 6

Teilnehmerinformationen

Teilnehmer	6
Background	Frontend Entwickler
Markdown Kenntnisse	technische Dokumentation
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	A (md)	B (html)		
Syntaxbeispiele 4	1 md	2 html	3 md	4 html
Syntaxbeispiele 7	1 md	2 html	3 md	4 html
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.md)	75	1	
1 (B.html)	175	3	
4	34	1	0
7	30	1	0
11	82	2	1
14	90	2	1
17	156	1	0

Beobachtungen

1. `A.md Zusammenhang der Folien unklar`

   `B.html`
   `"Wie ist nochmal eine Wahrheitstabelle aufgebaut?"`
   `nach ca 100 Sekunden Wahrheitstabellen (WHT) verstanden`
   `2 Versuche mit falschem Verständnis von WHT`

2. `"Tabelle ist sehr kompliziert zu lesen"`
   `Fand Markdown Slides übersichtlicher`

3. `"Markdown, definitv!"`

4.

5. `"überhaupt nicht schwer"`

6. "ich war mir sicher"
7.
8. "es war nicht schwer, eine easy Sache"
9. "100% sicher"
10. nach den Aufgaben "jetzt relativ sicher" bzgl. Aufbau
    und Struktur von Folien und Slidedecks
11. <h1> über '===' Syntax ausgedrückt
    Schachtelung bei ersten beiden Folien verstanden,
    dann jedoch dritte Folie auch über '##' und '###'
    unter ersten beiden Folien einsortiert

    Fehler:
     - Folie 3 & 4 mit '##' und '###' begonnen
       -> falsche Struktur

    Fehler konnten nach Aufzeigen der entstandenen
    Struktur sofort korrigiert werden
12. "die Anordnung von Folien ist bei Sicht auf
    den Code nicht klar"
13. "war eine effiziente Art Folien zu schreiben"
14. Hat Pfadelement nach Pfadelement im Code gesucht
    "Was auch immer das [t:Variable] heißt"

    Ist sich unsicher bei 4.
    "Alle anderen Beispiele zeigen auf Variablen
    und das [t:...] könnte Typ bedeuten"

    Fehler:
     - 4 als Typ Definition aufgelöst
15. Keine intuitive Idee, was [t:Variable] bedeutet

    [Erklärung der Bedeutung von [t:...]]
16. Kollision bei 'Race.Result' erkannt und korrekt
    [t:Variable] verwendet
17. Markierung bei Beispiel 1 nicht klar erkennbar
    "es war simples Abarbeiten"
    "habe jetzt grundlegendes Verständnis wie das funktioniert"

Ergebnisse



Lösung Aufgabe 11

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 16

Teilnehmer*in 7

Teilnehmerinformationen

Teilnehmer	7
Background	Azubi 2. LJ Schwerpunkt Frontend
Markdown Kenntnisse	„nicht wirklich“
HTML Kenntnisse	sehr gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	A (html)	B (md)		
Syntaxbeispiele 4	1 html	2 html	3 md	4 md
Syntaxbeispiele 7	1 html	2 html	3 md	4 md
Syntax 11	html			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.html)	142	1	
1 (B.md)	60	1	
4	129	1	0
7	44	1	0
11	201	2	1
14	256	1	0
17	300	2	1

Beobachtungen

1. `A.html Beudeutung Innenzahl nicht klar`

   `in B.md Tabelle direkt gefunden und zur richtigen Zeile gesprungen`

2. `"HTML ist leichter zu lesen als Markdown"`
   `"Markdown ist abstrakter zu lesen"`
   `HTML Folien durch Einrückung visuell sehr geordnet`

3. `HTML mehr Struktur und Einrückungen`

4. `Verhältnis von Ober- und Unterfolien gezählt und darüber statt über Syntax zugeordnet`

   `In HTML <h1>&<h2> verglichen, nicht <section>’s`

5. MD Hierarchie einfacher als HTML
   Unterscheiden von 1&2 am schwierigsten
6. 3&4 sicher zugeordnet
   1&2 "eher geraten"
7. Liste in Markdown intuitiv als Liste erkannt,
   jedoch nicht mit großer Sicherheit in eigene Intuition
8. "Zuordnen was ziemlich leicht"
9. ziemlich sicher
10. "Bin jetzt [nach den Aufgaben]
    besser in das Thema rein gekommen"
11. Syntax für HTML List Items und Listen Tags nicht im Kopf

    "Irgendwie mache ich gerade schon wieder das Gleiche"
    -> Eine Section zu viel begonnen

    "Die Syntax für HTML Codeblöcke müsste ich jetzt nachschauen"

    Zählt zum Überprüfen der Korrektheit alle öffnenden und
    schließenden Tags durch

    Fehler:
     - durch falsche Schachtelung alle Folien
       unter der ersten eingeordnet
     - schließendes Section Tag vergessen

    Korrekturen sind nach Hinweis auf Fehler klar.
12. "ich habe noch nie HTML auf Papier geschrieben"
    "es fiel mir schwer, ohne Syntax Highlighting
    und einem Browser im dem ich meinen Code zwischendurch
    zu checken kann"
13. "es ist nicht umbedingt die effizienteste Methode,
    um Folien zu schreiben"

    "hat sich nicht natürlich angefühlt"

    "ich mache nicht oft genug HTML, um die Syntax für
    Codeblöcke und Listen aus dem Kopf zu kennen"
14. Teilnehmer*in grübelt sehr lange über erstes Beispiel

    [Zwischenfrage: Hast du ein Gefühl, was die Pfade bedeuten]
    "Nein, im Moment nicht"

    Sucht Teile der Pfade direkt als Textfragmente im Code

    Kann 3. aus eigener Kraft lösen und beginnt danach

```
Verständis für Schachtelung zu entwickeln

"wird in 4. ein Variablenname gesetzt?"
Erwartete, dass 4. etwas mit Variablen und
nicht mit Typen zu tun hat
```

15. ```
    Kontextwechsel zwischen Folien und Quellcode schwierig
    Keine Erfahrung in Scala oder Java
    kaum Gefühl für Funktionalität der Pfade entwickelt

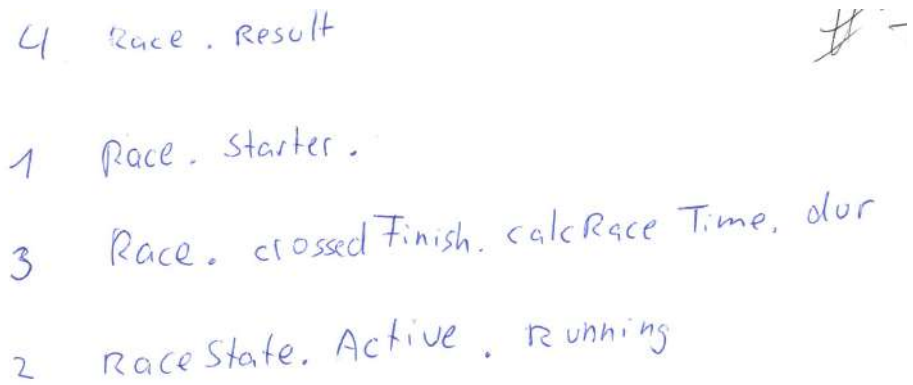
    Hat sich von Element zu Element "durchgehangelt",
    indem die einzelnen Elemente im Text gesucht wurden

    [Konzept der Pfade erklärt]
    ```

16. ```
    bei Angabe von Range für 1.:
    "ich nehme an, dass dann [der Pfad] alles,
    was darunter kommt mit mitnimmt"

    Fehler:
     - Namenskonflikt bei Race.Result nicht gesehen und
       [t: vergessen
    ```

17. ```
    Einrückung des Quelltexts zu klein -> schwer zu lesen
    "hatte als ich bei 3. war schon ein ganz gutes
    Gefühl für das Konzept"
    ```

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	4

Markierte Zeilen in Aufgabe 14
    ```



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 8

Teilnehmerinformationen

Teilnehmer	8
Background	Master Student Digitale Medien
Markdown Kenntnisse	keine (3 Sätze Primer gegeben)
HTML Kenntnisse	gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	B (md)	A (html)		
Syntaxbeispiele 4	1 md	2 html	3 html	4 md
Syntaxbeispiele 7	1 md	2 html	3 html	4 md
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (B.md)	20	1	
1 (A.html)	10	1	
4	20	1	0
7	17	1	0
11	45	1	0
14	67	1	0
17	180	2	2

Beobachtungen

1.
2. `"wenn man versteht was die Aufgabe ist ..."`
   `"Lesen in beiden Fällen nicht schwierig"`
3. `"beides [HTML & MD] gleich gut"`
4.
5. `"Aufgabe ist ziemlich einfach, wenn man weißt,`
   `was die '#' sind"`
6. `"Nachdem ich Beispiel 2 angesehen habe, war der Rest klar"`
   `sicher bei Zuordnung`
7.

8. "Im ersten Moment ist das alles etwas verwirrend"
   "Keywords auf Folien und im Sourcecode jedoch eindeutig"
9. ziemlich sicher
10. hat nach eigenen Aussage Überlick über Konzepte
11.
12. nach Vorübungen war es einfach
13. "habe mich durch Syntax eingeschränkt gefühlt
    in meinen Ausdrucksmöglichkeiten"

    grundsätzlich einfach
14. Pfad Schritt für Schritt aufgelößt
    Kollision für MyClass.Name nicht erkannt
    t: intuitiv als Type interpretiert
15. bei 1. komische Schreibweise
    (kompletter Pfad mit Package hat verwirrt)

    2. war eindeutiger (kein Package)

    "man konnte das [Ergebnis des Pfades]
    aus der Hierarchie des Codes ablesen und dem Pfad folgen"
16. "Wie schreibe ich jetzt mehrere Codezeilen"

    Fehler:
     - 2. RaceState.Active (Element vergessen)
     - 4. [t: vergessen

    Fehler konnten in einem Anlauf ohne Probleme korregiert werden
17. Einrückung des Codes schwer zu lesen

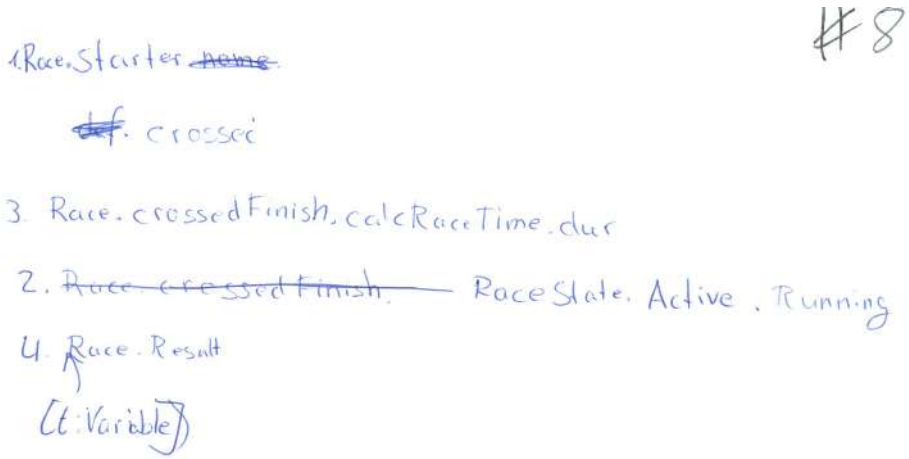
    "[Code] mit den ganzen Klammern und
    Kommentare ganz schon unübersichtlich"

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	4

Markierte Zeilen in Aufgabe 14

Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 9

Teilnehmerinformationen

Teilnehmer	9
Background	Master Student Informatik
Markdown Kenntnisse	einfache Github Readme's
HTML Kenntnisse	gut
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	B (md)	A (html)		
Syntaxbeispiele 4	1 html	2 md	3 md	4 html
Syntaxbeispiele 7	1 html	2 md	3 md	4 html
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (B.md)	12	1	
1 (A.html)	80	1	
4	40	1	0
7	33	1	0
11	130	1	0
14	138	2	1
17	180	2	1

Beobachtungen

1. bei A.html Innenzahlen-Folie getrennt betrachtet
   [Nach Tipp]:
   "das <li> zu nummerierter Liste wird, hatte ich gerade
   nicht so schnell gesehen"

   Hat Aufgabe über gleiche Reihenfolge der Einträge gelöst
   nicht über daraus entstehende gleiche Nummerierung

2. Aufgaben sehr unterschiedlich schwer
   bei B.md durch Fragestellung schon auf
   Tabelle vorbereitet gewesen

   Aufgabenstellung bei A.html erst nicht verstanden
   "dachte mir: Was kann schon an einer Liste

von Zahlen falsch sein"
Der Hinweis "bitte alle Folien beachten" war wichtig

3. "Markdown war einfacher [zu lesen]"
   "Ich war auf Wahrheitstabelle vorbereitet und [Quellcode]
   sah aus wie Tabelle"
4.
5. nicht so schwer
6. Nach Verständnis, was die Bilder darstellen, sehr sicher
7. 3. Über Dateinamen nicht Bildsyntax zugeordnet
   gezögert bei Zuordnung von 1&2
8. nicht schwer
   sehr viel inuitiver als Aufgabe 4
9. "sehr sicher, als ich alle [Beispiele] gesehen habe"
   "Ich war im ersten Moment bei Paragraph und Codeblock
   nicht sicher"
10. "das Konzept Folien in Code auszudrücken
    ist mir aus LaTeX bekannt und inzwischen intuitiv"

    "Vor allem mit den Sprachereferenzen
    aus den vorherigen Aufgaben sicher"
11. Direkt '#' und '##' korrekt Folien und Unterfolien zugeordnet

    Codeblock Syntax war nicht bekannt
    "Was macht plaintext da an dem Block, habe ich noch nie gesehen"
    -> Intuitive Idee: das ist wahrscheinlich optional
12. "ganz fluffig, es waren sehr einfache Folien"

    "eine 'echte' Agenda mit nacheinander auftauchenden
    Punkten ist bestimmt komplizierter"

    "diese statischen Folien waren sehr einfach und
    angenehm zu schreiben"
13. Anordnung von Folien in 2D-Grid ist ungewohnt

    Abstraktion von Quelltext mit Leserichtung TB zu
    Grid mit Leserichtung TBLR ist schwierig
14. 1. "Pfad ist eindetig hierarchisch"

    Pfad Element für Element aufgelößt

    2. "Ist da die Zuweisung der Variable oder ihre Nutzung
    eine Zeile später gemeint?"

    4. "das sieht anders aus!"

Kollision erkannt
"Ich könnte mir vorstellen, das t: für Typ steht"

Fehler:
- 4. Typ Defintition statt Var Defintion

15. "4. t: ist für mich nicht logisch"
"muss das vorne stehen?"

hierarchische Struktur ist logisch
"Vom Lesen her wäre 'Typ=Variable' logischer gewesen"

zu Typdefinition vorweg: "Ich habe ganz viele 'vars' in dem
Code hier, es wäre einfacher, wenn MyClas... vorne stehen würde"

"das die Packet Definition optional ist, finde ich schön"

16. Verwirrt über 1: "in allen vorherigen Beispielen
ist immer nur eine Zeile definiert"
-> intuitiv "Race.Starter" gewählt

Pfade immer von Race oder RaceState aus aufgebaut

"Ich muss mir beim Aufschreiben der Pfade gar nicht die
Bedeutung/Semantik anschauen, das gefällt mir gut"

"Eigentlich hangel ich mich an den Blöcken aus
Klammern und Einrückungen entlang und nehme immer
den ersten Bezeichner aus der Zeile als Pfadelement"

"Bei Blöcke nehme ich an, dass ich da auf die Zeile mit
der öffnenden Klammer zeigen muss"

Fehler:
- [t: bei 4 vergessen

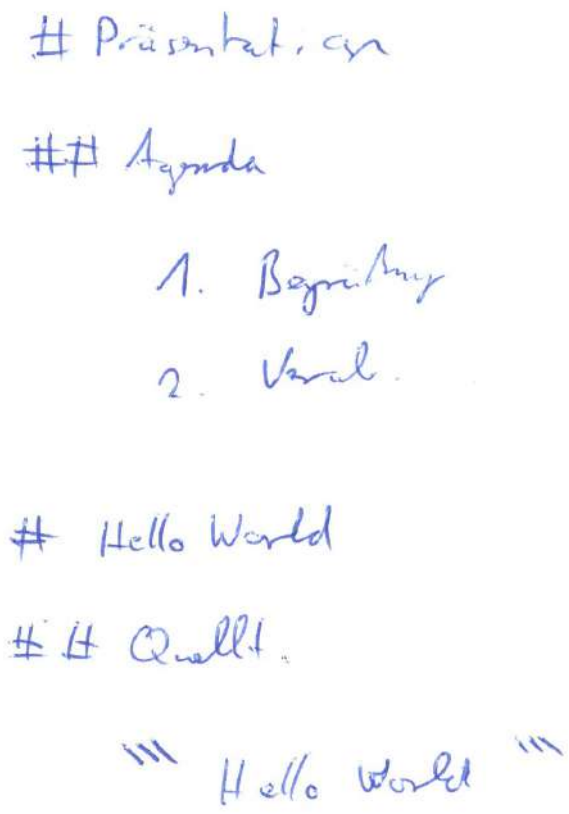
17. Blockkdefinition ist uklar

"die Syntax für 4 ist nicht glücklich gewählt"

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11

Lösung Aufgabe 16

Teilnehmer*in 10

Teilnehmerinformationen

Teilnehmer	10
Background	Master Student Informatik
Markdown Kenntnisse	Readme's schreiben, Abgaben für die Uni
HTML Kenntnisse	vorhanden, ein bisschen eingestaubt
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	A (html)	B (md)		
Syntaxbeispiele 4	1 md	2 html	3 md	4 html
Syntaxbeispiele 7	1 md	2 html	3 md	4 html
Syntax 11	html			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (A.html)	46	1	
1 (B.md)	10	1	
4	50	1	0
7	15	1	0
11	182	1	0
14	150	2	1
17	180	2	1

Beobachtungen

1. `A.html [Wie bist du bei der Lösung vorgegangen?]`
   1. `Bilder angesehen, welche Zahl ist da drin`
   2. `Paarweise mit Einträgen in den ersten beiden Listen verglichen`

   `"Meine Annahme war, dass die auf allen Folien in der gleichen Reihenflolge sind"`
2. `"es war einfach"`
   `"A etwas schwieriger, weil mehr Inhalt [als B]"`
   `"in B einfach Tabelle gesehen und einfach Zeile rausgesucht"`
3. `Markdown`
4. `1 zugeordnet, bei 2: "Wundert mich, dass nach <h1>A</h1> kein </section> ist"`

5. "nicht wirklich schwer, straight forward"
   "Meine Intuition bei 2 war, da das immer weiter geschatelt ist,
   dass da ein Baum bei rauskommen sollte"
6. "1,3 und 4 konnte ich durch ihre Struktur sicher zuordnen"
   2 per Ausschlussverfahren
7.
8. relativ einfach
   "Das Bild und der Codeblock sind
   visuell auf Sldies relativ ähnlich"
9. "ja, war mir sicher"
10. grundlegendes Verständnis
    "bei HTML weniger, da sind Tags,
    die mir manchmal nicht einfallen"
11. Zuerst Überschriften ohne Tags eingetragen,
    aber selbstständig korregiert
    Liste zuerst mit <li> Tags begonnen,
    gestockt und Syntax für Listen erfragt
    Syntax für Codeblock erfragt
12. Schwierigkeiten sich an HTML Tags zu erinnern

    [gab es Probleme bei der abstraktion Code zu Slides?]
    "Ich dachte andauernd wieder an Bäume nicht Gitter"
13. "geht so, eher nicht"
    Viele Tags
    "am Ende war es eher eine Aufgabe von:
    'welche Tags muss ich jetzt noch schließen?'"
14. Pfad Element für Element aufgelößt
    bei 3. "Ah, das ist alles immer so geschachtelt!"
    "Was bedeutet [t:Variable]?"

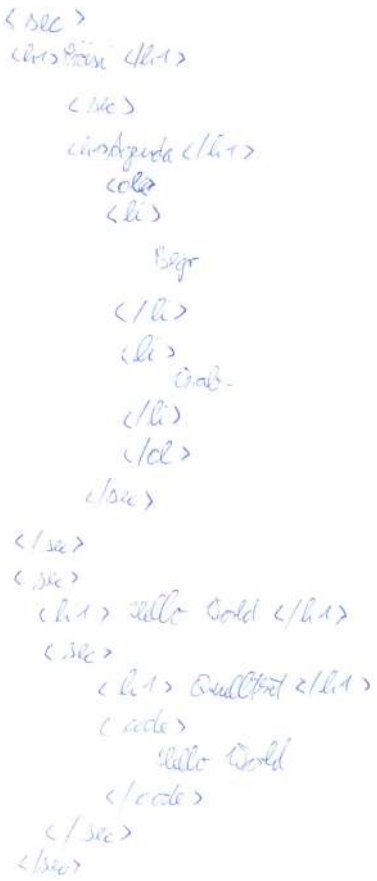
    Kollision für MyClass.Name gefunden
    "das gibt wohl den Typen zurück"

    Fehler:
     - 4 Type-Def statt Var-Def
15. "1 war bis ich das 'package' keyword gefunden habe unklar,
    danach war es nur noch Baumtraversierung"
16. Verwirrung über Bedeutung von 'object' und 'class'
    Kollision für Race.Result nicht gesehen

    Fehler:
     - 4 Pfad ohne [t:Variable]
17. "Auswahl von mehreren Zeilen war in keinem Beispiel bei 14,
    ist aber, wie ich geraten hätte"

```
"das es zwei Race.Result gibt, habe ich übersehen"

"sonst war die Aufgabe nur Baum traversieren
zu der Stelle die ich brauche"
```

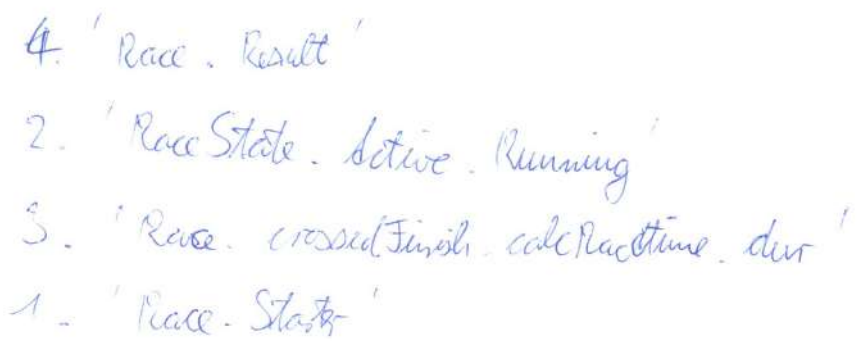
Ergebnisse



Lösung Aufgabe 11

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 16

Teilnehmer*in 11

Teilnehmerinformationen

Teilnehmer	11
Background	Master Student Informatik
Markdown Kenntnisse	Schreiben von Notizen, Uni Abgaben
HTML Kenntnisse	grundlegend, Schule, Softwareprojekt 2
revealJS Kenntnisse	nein

Testparameter

Zeiten, Versuche und Fehler

Beobachtungen

1. `B.md Verständnisprobleme Wahrheitstabelle und Aufgabenstellung Nach Klärung der Probleme Lösung in 10s`

   `A.html zuerst für ca 60s syntaktischen Fehler im HTML gesucht danach Listen Zeile für Zeile verglichen bis Fehler gefunden wurde`

Reihenfolge 1	B (md)	A (html)		
Syntaxbeispiele 4	1 html	2 md	3 html	4 md
Syntaxbeispiele 7	1 html	2 md	3 html	4 md
Syntax 11	html			

Aufgabe	Zeit	Versuche	Fehler
1 (B.md)	100	1	
1 (A.html)	100	1	
4	40	1	0
7	30	1	0
11	150	2	1
14	184	2	1
17	224	1	0

2. "bei A.html musste ich mir erstmal ein
   semantisches Mapping aufbauen, die Tabelle (B.md)
   war sofort als solche erkennbar"

   "bei A.html mussten erstmal Listen verglichen werden, um
   den Zusammenhang zu finden"
3. "definitiv Markdown"
4. zuerst Probleme 3&4 zuzuordnen,
   dann anhand des Verhältnisses zwischen Toplevel
   und geschachtelten Folien gelöst
5. "nicht schwierig"
   "die Anzahl der <h1> gibt vor, wie viele Spalten ich habe"
6. sehr sicher
7. ".png muss wohl ein Bild sein"
   "<p> ist ein einzelner Text"
   "- sind Stichpunkte"
8. "etwas schwieriger als 4"
   "Absatz und Code war nicht ganz einfach"
   "zuerst gedacht, dass 'plaintext' ein Absatz sein könnte,
   dann die Anführungsstriche gesehen"
9. "sicher bis sehr sicher"
10. "habe ein Gefühl dafür [Aufbau Slides]"
    "Aufbau von Grid und wie ich Inhalte in diesem Grid
    strukturiere nicht ganz klar"
11. hat intuitiv wie in Beispielen <section>'s geschachtelt

    Fehler:
     - schließendes </section> vergessen

    Korrektur ohne zögern möglich

12. "ich habe mich mehr um die Syntax
    als um den Inhalt kümmern müssen"

    "Es ist aufwändig schließende Klammern
    und Tags zu checken"

    Die Strukturierung der Folien war eindeutig

    Es ist aufwändiger als HTML

13. "nicht so richtig, habe mich nicht sehr effizient gefühlt"
    viel Syntax
    "habe die meiste Zeit Tags nicht Text geschrieben"

14. Pfade Element für Element aufgelöst
    "jeder Schritt ist genau benannt"
    aus [t:Variable] gefolgert: Type Variable
    Kollision gesehen

    Fehler:
     - Type-Def statt Var-Def markiert

15. "nicht so schwer"
    "man muss den Pfad einfach einmal durchgehen"

16. "Wie gebe ich Bereiche an? Vorher waren es immer
    nur Zeilen"

    "Vielleicht gibt es ein Keyword für 'alles' oder
    eine Klammern Semantik"

    bei object Pfad kompletten Paketnamen vorangestellt
    [Warum volles Paket vorweg?]
    "habe ich vorher [bei 14.1] auch so gesehen,
    das hier ist auch ein object"

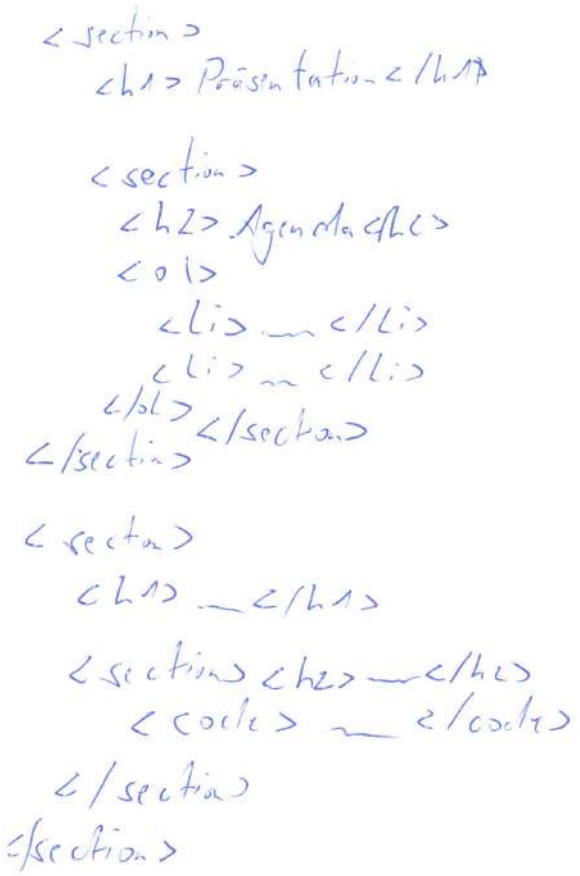
    Kollision bei 4 erkannt und korrekt [t:Var eingesetzt

17. "Bei der Auswahl kompletter Klassen nicht sicher"
    "Mein Educated Guess war: alles darin wird übernommen"
    "Lösen der Aufgabe hat kein Problem dargestellt"

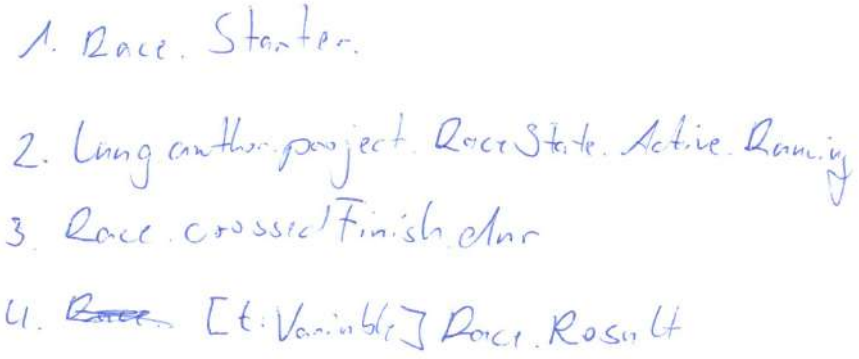
Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14



Lösung Aufgabe 11



Lösung Aufgabe 16

Teilnehmer*in 12

Teilnehmerinformationen

Teilnehmer	12
Background	Master Student Informatik
Markdown Kenntnisse	Git Wikis, Uni Abgaben
HTML Kenntnisse	gering, aus Schulzeiten, keine Nutzung in 5 Jahren
revealJS Kenntnisse	nein

Testparameter

Reihenfolge 1	B (html)	A (md)		
Syntaxbeispiele 4	1 md	2 md	3 html	4 html
Syntaxbeispiele 7	1 md	2 md	3 html	4 html
Syntax 11	md			

Zeiten, Versuche und Fehler

Aufgabe	Zeit	Versuche	Fehler
1 (B.html)	90	2	
1 (A.md)	96	1	
4	71	1	0
7	73	1	0
11	64	2	1
14	170	2	1
17	220	1	0

Beobachtungen

1. `B.html Nicht sicher über Aufbau von Tabellen in HTML`

   `A.md hat zuerst erwartet, dass Innenzahl`
   `ein Mapping von Form zu Form darstellt`
2. `Struktur bei Markdown einfacher zu lesen`
   `"es ist übersichtlicher und ich habe mehr Erfahrung damit"`
   `Viele Tags in HTML, die von Struktur ablenken`
   `Inhalt in beiden Folien verständlich`
3. `Markdown`
4.

5. "relativ simpel, nachdem ich 3 verstanden hatte"

   "Markdown war für einfache Strukturen [1&2] überhaupt
   kein Problem"

6. sehr sicher

7. Liste erkannt: "das hat viele Abschnitte"
   Bild an <image> Tag erkannt

8. "Hat Spaß gemacht"
   "Habe zuerst etwas an der Liste gehangen"
   "musste etwas rein kommen"
   "der Rest [1,2,3] war straight forward"
   "Aufgabe war auch mit geringen HTML Kenntnissen zu lösen"

9. "ja, sicher"

10. "ich habe einen ganz guten Überblick darüber"

11. "Wie ging nochmal nummerierte Liste?"

    Fehler
     - # statt ## für eine Unterfolie verwendet

    "Es passiert einem ziemlich oft mal ein # zu
    vergessen, wenn man mehrere Überschriften hat"

    "Man merkt das [Vergessen von #] meist sehr schnell,
    nachdem man das [den Quellcode] durch einen
    Konverter jagt"

12. "Syntax für nummerierte Liste musste ich drüber nachdenken"

    "sonst ist das alles sehr simples Markdown,
    um einfache Folien zu schreiben"

13. "war effizient"
    "musste mich wenig mit Formatierung beschäftigen"
    "man kann schnell aufschreiben, was man sagen will"

    "gut im Quelltext zu lesen, man braucht keinen Editor
    mit Autocompletion und so"

14. Package gesucht und "Pfad Definition für
    Definition durchgegangen"

    Namenskollision nicht gesehen

    [t:Var Bedeutung unklar "Pfad ist wohl nur der
    Typ der Variable"
    -> "MyClass.Name würde also nur String auswählen"

    Fehler:

```
 - 4 definiert nur Typ der Variable -> 'String'
```

15. `"[t:Variable] ist nicht intuitiv ohne weitere Erklärung"`
    `"der Rest war straight forward"`
    `"man kann Definitionen einfach abgehen"`
    `"[Pfade sind] einfach zu verfolgen"`

16. `Beispiel 1: "Meine Intuition sagt Race.Starter`
    `sollte es eigentlich sein"`

    `"Da ich nichts darunter definiere, erwarte ich,`
    `dass alles darunter geprinted wird"`

    `Beispiel 2 mit komplettem Package, weil Beispiel`
    `14.1 auch object war und komplettes Package enthielt`

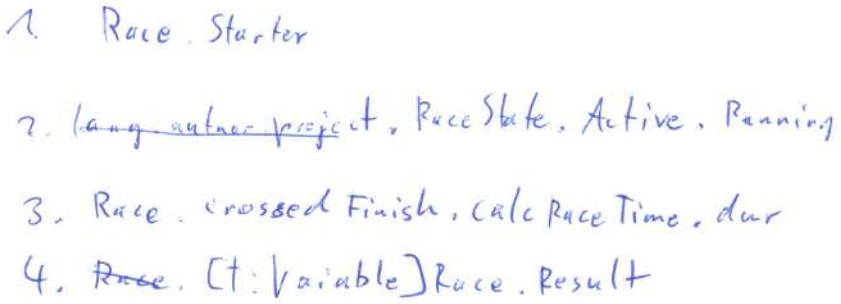
17. `Verwirrung, wie object außerhalb von class funktioniert`
    `konnte Pfade entlang ihrer Einrückung so hinschreiben`
    `"auch 4 war nach Erklärung meines Fehlers [aus 14] einfach"`

Ergebnisse

Pfad	Lösung	Teilnehmerantwort
1	47	47
2	35	35
3	18	18
4	4	9

Markierte Zeilen in Aufgabe 14

Lösung Aufgabe 11



Lösung Aufgabe 16