Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Entwicklung einer Webanwendung zur Erstellung, Verwaltung und Ausführung von Programmen in Curry

Lasse Kristopher Meyer

Wintersemester 2013/2014
(eingereicht im März 2014)

Betreut durch Prof. Dr. Michael Hanus und M. Sc. Björn Peemöller

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

1.1. Motivation

Für die Entwicklung komplexer Software-Systeme oder -Komponenten greifen Programmierer häufig auf Werkzeuge wie integrierte Entwicklungsumgebungen (integrated development environments, IDEs) zurück. Diese bieten neben einem Texteditor, welcher zur effizienten Quelltexterzeugung und -bearbeitung in der Regel diverse Funktionalitäten wie Syntax-Highlighting oder Code-Autovervollständigung bereitstellt, oft auch Infrastruktur für die Projektverwaltung und die Organisation der Softwarebestandteile (Module, Klassen). Um den Medieneinsatz bei der Entwicklung und dem zugehörigen, fortwährenden Testprozess gering zu halten und einen möglichst hohen Arbeitsfluss zu gewährleisten, sind IDEs außerdem zumeist mit Compilern oder Interpretern ausgestattet, die die unmittelbare Ausführung des verfassten Quelltextes erlauben.
Auch im Kontext unversitärer Lehre im Bereich der Informatik ist der Einsatz von intergrierten Entwicklungsumgebungen nicht unüblich und im Rahmen von Programmierpraktika und der Simulation komplexer Softwareentwicklung (z. B. begleitend zur Lehre der Softwaretechnik) häufig sogar erwünscht und als Teil des Lehrstoffs zu betrachten. Im Bereich der Lehre *grundlegender* Programmiertechniken hingegen, wo nicht die Konstruktion vielschichtiger Software-Systeme, sondern die Vermittlung der fundamentalen Prinzipien und Konzepte von Programmiersprachen und -paradigmen im Vorderung steht, erscheint der Einsatz umfangreicher Werkzeuge wie IDEs oft unverhältnismäßig, insbesondere in Hinblick auf die Komplexität der in diesen Lehrveranstaltungen typischerweise von Studenten zu bearbeitenden Programmieraufgaben. So umfassen diese üblicherweise die Erstellung unabhängiger, kompakter Programme von geringem Umfang, die beispielhaft ein bestimmtes Programmierkonzept oder eine bestimmte Programmiertechnik umsetzen. Für die Bearbeitung solcher Aufgaben erweisen sich IDEs nun unter Umständen als weniger gut geeignet, da sie aufgrund ihres großen Funktionsumfangs mitunter zu schwerfällig agieren oder eine (im Verhältnis zur Problemstellung) zu komplexe Bedienbarkeit aufweisen. Weiterhin sind IDEs zumeist für die Entwicklung mit einer bestimmten Programmiersprache konzipiert, während es insbesondere im Lehrbereich grundlegender Programmiertechniken durchaus möglich ist, dass im Laufe einer Lehrveranstaltung auf verschiedene Programmiersprachen zurückgegriffen wird, um bspw. die Konzepte verschiedener Programmierparadigmen (funktional, objektorientiert, etc.) zu behandeln. In einem solchen Fall wäre es wünschenswert auf ein flexibles und leicht handhabbares Werkzeug zurückgreifen zu können, welches die vorteilhaften Funktionen integrierter Entwicklungsumgebungen (z B. das Bereitstellen eines geeigneten Quelltexteditors, die Ausführung von Code mit einer integrierten Ausführungseinheit und die Verwaltung der erstellen Programme) unterstützt, auf die Erstellung von unabhängigen Programmen mit geringer Komplexität zugeschnitten ist und dabei bezüglich der Kompatibilität mit Programmiersprachen möglichst uneingeschränkt erweiterbar ist.

Um den unmittelbaren Zugriff auf ein solches Werkzeug zu gewährleisten, liegt dessen Realisierung als Webanwendung nahe. Dies hat zunächst den Vorteil der Plattformunabhängigkeit, ermöglicht aber insbesondere eine noch vielseitigere Einsetzbarkeit der Software. Die Ausführung im Webbrowser und die Verwaltung bzw. Bereitstellung von Programmen auf Basis einer zentralen Datenbank kann bspw. auch Entwickler unterstützen, die ausgewählte Komponenten ihrer Software (Funktionen, Module) testen möchten oder auf der Suche nach bewährten Lösungen für bestimmte, wiederkehrende Problemstellungen sind. Nicht zuletzt ist die Möglichkeit zur Bereitstellung selbst verfasster Programme auch für Dozenten interessant. So wäre die Nutzung einer entsprechenden Webanwendung als Plattform zur Verteilung von Lösungen oder Beispielprogrammen aus Vorlesungen vorstellbar. Über die integrierte Ausführungsfunktion könnten die Programme dann direkt von den Studenten getestet und nachvollzogen werden. Es zeigt sich also, dass ein derartiges, webbasiertes Werkzeug durch seine vielfältigen Anwendungsmöglichkeiten auch eine potentielle Schnittstelle zwischen Studierenden, Dozenten und Entwicklern darstellen kann.

1.2. Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer Webanwendung, welche entsprechend der Ideen aus dem letzten Abschnitt als zentrale Plattform für die Erstellung, Bearbeitung, Verwaltung, Bereitstellung und Ausführung von Programmen bzw. Quelltexten beliebiger Programmiersprachen im Browser dienen soll. Gemäß der anfänglichen Konzeption als integrierte Entwicklungsumgebung für *Programme mit geringem Umfang*, trägt diese Webanwendung dabei den Namen *Smap* (ein Kofferwort für „**sma**ll **p**rograms“).

Die konkrete Implementierung der Webanwendung Smap erfolgt unter Verwendung der deklarativen, funktional-logischen Programmiersprache Curry [Han12] bzw. des in Curry implementierten Webframeworks Spicey [HK12]. Curry ist eine experimentelle Programmiersprache, die im Wesentlichen als Plattform für die Erforschung und Lehre funktional-logischer Programmiersprachen dient (eine ausführliche Einführung in die Programmiersprache Curry erfolgt in Abschnitt 2.1). Das in Curry implementierte und für die Entwicklung mit Curry konzipierte Spicey-Framework (siehe Abschnitt 2.2) basiert auf Konzepten der Webprogrammierung in Curry und wurde bspw. für die Implementierung der Moduldatenbank des Instituts für Informatik der Christian-Albrechts-Universität zu Kiel[1] verwendet. Die Entwicklung der Webanwendung Smap soll nun eine weitere Möglichkeit zur Erprobung der Konzepte webbasierter Programmierung in Curry und des Spicey-Frameworks bei der Umsetzung komplexer, nichttrivialer Webanwendungen bieten und als zusätzliche Grundlage für deren stetige Verbesserung und Weiterentwicklung dienen.

[1] `https://mdb.ps.informatik.uni-kiel.de/show.cgi`

1.3. Struktur der Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert: In Kapitel 2 werden zunächst die zum Verständnis der nachfolgenden Kapitel notwendigen Grundlagen erarbeitet. Dies umfasst eine Einführung in die allgemeine und in die webbasierte Programmierung mit Curry, eine grundlegende Beschreibung des Webframeworks Spicey, sowie Beschreibungen des Front-End-Frameworks Bootstrap und des JavaScript-Editors CodeMirror. Kapitel 3 befasst sich anschließend mit dem Entwurf der Anwendung. Dazu werden die Anforderungen an die Anwendung ermittelt und strukturiert, die zentralen funktionalen Komponenten identifiziert und erläutert und das der Anwendung zugrunde liegende Datenmodell hergeleitet. Aufbauend auf den Ergebnissen des Entwurfs erfolgt daraufhin in Kapitel 4 eine ausführliche Beschreibung der Implementierung der Webanwendung. Im letzten Kapitel werden abschließend die erarbeiteten Ergebnisse zusammengefasst und bewertet, sowie Ideen für zukünftige Erweiterungen der entwickelten Software diskutiert.

2. Grundlegende Software und Konzepte

In diesem Kapitel werden die bei der Entwicklung der Webanwendung Smap verwendete Software vorgestellt und grundlegende Konzepte beschrieben, deren Kenntnis zum Verständnis der folgenden Kapitel notwendig ist. Der erste Abschnitt dieses Kapitels gibt einen Überblick über die Programmiersprache Curry. Neben einer Beschreibung der wesentlichen Konzepte und Eigenschaften der Sprache erfolgt dabei insbesondere eine kurze Einführung in die Grundlagen der deklarativen Webprogrammierung mit Curry. Der nächste Abschnitt befasst sich mit dem Curry-Webframework Spicey, welches – wie einleitend dargelegt – zur Implementierung der Anwendung verwendet wurde. In den letzten beiden Abschnitten werden abschließend das Front-End-Framework Bootstrap und der in JavaScript implementierte Texteditor CodeMirror vorgestellt.
Für das Verständnis dieses Kapitels werden grundsätzliche Kenntnisse der Konzepte der Programmiersprache Haskell[1], der Auszeichnungsprache HTML[2], der Stylesheet-Sprache CSS[3] und der Skriptsprache JavaScript[4] vorausgesetzt.

2.1. Die Programmiersprache Curry

2.1.1. Eigenschaften und Konzepte

Curry[5] ist eine experimentelle, funktional-logische Programmiersprache, welche von einem internationalen Team als gemeinsame Plattform für die Erforschung, Lehre und Anwendung funktional-logischer Sprachen entwickelt wird [Han12]. Curry verknüpft typische Eigenschaften funktionaler Programmiersprachen (verschachtelte Ausdrücke, Lazy Evaluation und Funktionen höherer Ordnung) mit den wichtigsten Eigenschaften logischer Programmiersprachen (logische Variablen, partielle Datenstrukturen und Lösungsstrategien mittels Suchverfahren). Dazu greift Curry teilweise auch auf Techniken der nebenläufigen Programmierung zurück (bspw. bei der nebenläufigen Auswertung von Constraints mit Synchronisation auf logischen Variablen).

Curry-Programme bestehen grundsätzlich aus einer Menge von Typ- und Funktionsdeklarationen. Typdeklarationen (auch Datentypen oder Datenstrukturen) beschreiben dabei

[1] http://www.haskell.org/haskellwiki/Haskell
[2] http://www.w3schools.com/html/
[3] http://www.w3schools.com/css/
[4] http://www.w3schools.com/js/
[5] http://www.curry-language.org/

Wertebereiche, deren Elemente die Ergebnisse der Anwendung ihrer Konstruktoren sind. Funktionsdeklarationen (auch Regeln) repräsentieren die Operationen auf diesen Wertebereichen. Syntaktisch – und bezüglich der funktionalen Eigenschaften auch semantisch – weist Curry dabei eine große Ähnlichkeit zu der funktionalen Programmiersprache Haskell auf. Folgender Code beschreibt etwa die Deklaration einer Funktion, die das Quadrat einer gegebenen Zahl vom Typ `Int` berechnet:

```
square :: Int -> Int
square x = x * x
```

Curry ist eine stark getypte Programmiersprache – jedes Objekt eines Curry-Programms besitzt also einen eindeutigen Typ. Typangaben (bzw. Signaturen; siehe erste Zeile des obigen Codebeispiels) können in der Regel jedoch ausgespart werden, da diese über einen Typinferenz-Mechanismus rekonstruiert werden können. Bei der Auswertung eines zu der linken Seite einer Regel passenden Ausdrucks wird dieser stets durch den Ausdruck der rechten Seite der Regel ersetzt, wobei etwaige Variablenbindungen übernommen werden. Im obigen Fall wird bspw. der Aufruf `square 8` durch den Ausdruck `8 * 8` ersetzt, welcher wiederum zum Wert `64` ausgewertet wird. Im Allgemeinen entspricht das Ausführen eines Curry-Programms dem sukzessiven Ersetzen und Vereinfachen eines Ausdrucks bis zur Errechnung eines Werts (ein Ausdruck, der keine Funktions- oder Operatorenaufrufe enthält) oder einer Lösung (im Falle logischer Berechnungen). Dazu verwendet Curry die Lazy-Evaluation-Strategie, d. h., dass Ausdrücke nur ausgewertet werden, wenn sie zur Berechnung des Endergebnisses notwendig sind und dass jeder Ausdruck höchstens ein einziges Mal berechnet wird. Im Gegensatz zu rein funktionalen Sprachen wie Haskell erlaubt Curry auch die Definition nichtdeterministischer Funktionen mittels überlappender Regeln:

```
curryParadigm = "functional"
curryParadigm = "logic"
```

In diesem Fall wird der Ausdruck `curryParadigm` entweder zu `"functional"` oder zu `"logic"` ausgewertet, da Curry im Gegensatz zu Haskell nach dem Anwenden der ersten Regel auch nach weiteren Lösungen sucht.

Die Definition von Datentypen erfolgt wie in Haskell über das Aufzählen aller Konstruktoren zusammen mit den Typen ihrer Argumente:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Der rekursive Datentyp `Tree a` repräsentiert hier einen Binärbaum, dessen Blätter Ausdrücke eines beliebigen Typs speichern können. Die Verarbeitung von Werten eines solchen Datentyps erfolgt in der Regel über Pattern Matching. Dabei wird für jeden Konstruktor des Datentyps eine Regel definiert. Mögliche Argumente der Konstruktoren werden dabei im einfachsten Fall in Form von Variablen notiert, die auch in den rechten Seiten der Regeln auftauchen dürfen:

```
mapTree :: (a -> b) -> (Tree a) -> (Tree b)
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) = Node (mapTree f l) (mapTree f r)
```

Die Funktion `mapTree` ist eine mittels Pattern Matching induktiv definierte Funktion,

welche eine gegebene Funktion `f` des polymorphen Typs `a -> b` auf alle Blattwerte eines Binärbaums des Typs `Tree a` anwendet und den dadurch erzeugten Binärbaum des Typs `Tree b` zurückliefert. Dazu wird die Funktion `f` im Falle eines Blattes sofort auf dessen Wert angewendet, während im Falle eines inneren Knotens die nachfolgenden Teilbäume rekursiv behandelt werden. `mapTree` ist sowohl ein Beispiel für eine polymorphe Funktion als auch für eine Funktion höherer Ordnung. Der polymorphe Charakter wird durch die Verwendung der Typvariablen `a` und `b` gekennzeichnet, welche besagen, dass der Typ der Blattwerte des gegebenen Binärbaums mit dem Argumenttyp und der Typ der Blattwerte des erzeugten Baumes mit dem Ergebnistyp der Funktion `f` übereinstimmen muss. Unter Berücksichtigung dieser Einschränkungen kann die konkrete Belegung der Typvariablen `a` und `b` allerdings beliebig sein. Funktionen höherer Ordnung zeichnen sich hingegen dadurch aus, dass sie andere Funktionen als Argumente erwarten (wie hier die Funktion `f` vom Typ `a -> b`) oder als Ergebnis liefern.

Abgesehen von den nichtdeterministischen Funktionen sind die bisher genannten Konzepte typische Eigenschaften funktionaler Programmiersprachen. Zusätzlich unterstützt Curry die Verwendung logischer Variablen und die Auswertung von Ausdrücken mit partiell instantiierten Argumenten (als typische Eigenschaften logischer Programmiersprachen). Eine logische Variable ist dabei eine Variable, die *nur* in der rechten Seite einer Regel vorkommt und somit zunächst ungebunden ist. Solche Variablen werden in Curry mit dem Suffix `where` *vs* `free` deklariert, wobei *vs* eine durch Kommas getrennte Liste von Bezeichnern ungebundener Variablen ist:

```
last :: [a] -> a
last l | xs++[e] =:= l = e where xs,e free
```

Die Funktion `last` berechnet das letzte Element einer Liste von Werten mit beliebigem Typ (gekennzeichnet durch den Typ `[a]`), falls ein solches existiert. Der Ausdruck `xs++[e] =:= l` repräsentiert hierbei eine Bedingung vom Typ `Success`. Diese ist dann erfüllt, wenn die ungebundenen Variablen `xs` und `e` derart instantiiert werden können, dass `xs` dem Anfangsstück der Argumentliste `l` entspricht (ohne das letzte Element) und `e` den Wert des letzten Elements von `l` annimmt. Kann Curry eine solche Instantiierung vornehmen (z. B. durch eine nichtdeterminisstische Suche nach möglichen Werten), dann wird das letzte Element `e` als Ergebnis geliefert. Im Falle der leeren Liste `[]` schlägt die Berechnung hingegen fehl. Im Allgemeinen ist eine bedingte Regel der Form `l | c = r` dann reduzierbar (im Sinne der Ersetzung eines zu `l` passenden Ausdrucks durch einen der rechten Seite `r` entsprechenden Ausdruck), wenn die Bedingung `c` lösbar ist.

Für die Auswertung von Ausdrücken, die logische Variablen enthalten (wie bspw. `xs++[e] =:= l` aus dem obigen Beispiel), verwendet Curry zwei verschiedene Ansätze, was zu folgender grundsätzlicher Klassifizierung von Funktionen führt: Sogenannte starre Funktionen (rigid functions) basieren auf dem Ansatz, zu suspendieren, falls ein für die weitere Auswertung benötigtes Argument ungebunden ist. Sollte das entsprechende Argument bei der nebenläufigen Auswertung eines anderen Ausdrucks instantiiert werden, kann die Berechnung wieder aufgenommen werden. Andernfalls schlägt die ursprüngliche Auswertung fehl. Demgegenüber stehen flexible Funktionen (flexible functions), welche stattdessen versuchen, passende Werte für ungebundene Argumente zu erraten. Die erste Strategie (auch Residuation genannt) wird z. B. bei von arithmethischen Operatoren wie $*$ verwendet,

während der Operator `=:=` aus dem obigen Beispiel entsprechend der zweiten Strategie (auch Narrowing genannt) auswertet.

Für die in [Han12] beschriebene Spezifikation der Programmiersprache Curry wurden mit der Zeit verschiedene Implementierungen entwickelt. Die Implementierung PAKCS[6] übersetzt Curry-Code beispielsweise in äquivalenten Prolog-Code. Die an der Universität Kiel entwickelte Implementierung KiCS2[7] basiert hingegen auf der Übersetzung in Haskell-Code. Für die Entwicklung der Webanwendung Smap wurde aus Kompatibilitätsgründen mit dem Spicey-Framework hauptsächlich die PAKCS-Distribution verwendet.

2.1.2. Webprogrammierung in Curry

Im Umfang der PAKCS- und der KiCS2-Distribution sind zahlreiche Bibliotheken enthalten, die die Entwicklung von Webanwendungen mit Curry unterstützen sollen. Von besonderem Interesse sind hierbei die Bibliotheken `HTML` und `WUI`, welche die Grundlage für die Entwicklung mit dem Spicey-Framework bilden [HK12].

Basierend auf dem CGI-Standard [Net04] stellt die Bibliothek `HTML` Funktionalitäten zur serverseitigen Entwicklung dynamischer Webseiten mit der Programmiersprache Curry zur Verfügung. Das Augenmerk liegt dabei insbesondere auf der Abstraktion von der konkreten HTML-Syntax und auf der Bereitstellung einer geeigneten Schnittstelle zur Konstruktion von Formularen [Han01]. Zur Erstellung dynamischer Webseiten definiert die Bibliothek `HTML` den zentralen Datentyp `HtmlExp`:

```
data HtmlExp
  = HtmlText   String
  | HtmlStruct String [(String,String)] [HtmlExp]
  | HtmlCRef   HtmlExp CgiRef
  | HtmlEvent  HtmlExp HtmlHandler
```

Der Konstruktor `HtmlText` beschreibt reinen, unformatierten Text innerhalb eines HTML-Dokuments. `HtmlStruct` ist der zentrale Konstruktor für die Zusammensetzung komplexer HTML-Dokumente und repräsentiert ein HTML-Element mit einem Namen (z. B. `"h1"` oder `"p"`), einer Liste von HTML-Attributen und einer Liste von HTML-Ausdrücken, die von dem Element umschlossen werden. Um die umständliche Verwendung des `HtmlStruct`-Konstruktors zu vermeiden, definiert `HTML` Abkürzungen für einen schnelleren Zugriff auf bestimmte Elemente der HTML-Spezifikation:

```
strong :: [HtmlExp] -> HtmlExp
strong hexps = HtmlStruct "strong" [] hexps
```

Über die Konstruktoren `HtmlCRef` und `HtmlEvent` können die Elemente eines HTML-Formulars beschrieben werden. Ein HTML-Formular enthält typischerweise eine Menge von Eingabeelementen (Textfelder, Checkboxen, etc.) und Elemente, mit denen der Inhalt des Formulars abgeschickt werden kann (z. B. Buttons). Um die Handhabung solcher Forumlarelemente zu vereinfachen, verwendet die Bibliothek `HTML` ein Konzept, welches auf der automatischen Generierung von sogenannten CGI-Referenzen und ihrer Verknüpfung

[6] `http://www.informatik.uni-kiel.de/~pakcs/`
[7] `http://www-ps.informatik.uni-kiel.de/kics2/`

mit speziellen Event-Handlern beruht. Eine CGI-Referenz ist dabei nichts anderes als ein gekapselter `String`-Wert:

```
data CgiRef = CgiRef String
```

CGI-Referenzen sind vom Nutzer nicht direkt erzeugbar, sondern lassen sich ausschließlich über die Instantiierung logischer Variablen generieren. Die Implementierung sorgt dabei dafür, dass jede generierte Referenz einzigartig ist. Der Wert eines mit einer CGI-Referenz versehenen Eingabeelements kann anschließend von einem sogenannten HTML-Handler extrahiert werden. Ein HTML-Handler ist eine Funktion, die CGI-Referenzen mittels einer gegebenen CGI-Umgebung auslesen kann und als Ergebnis ein neues HTML-Formular als I/O-Aktion[8] zurückliefert:

```
type CgiEnv      = CgiRef -> String
type HtmlHandler = CgiEnv -> IO HtmlForm
```

Über die Konstruktoren `HtmlCRef` und `HtmlEvent` lassen sich nun Eingabeelemente mit Referenzen und Submit-Buttons mit entsprechenden Handlern versehen. Beim Abschicken des Formulars über den Submit-Button extrahiert der zugehörige HTML-Handler zunächst den Wert eines Eingabeelements mittels CGI-Referenz und erzeugt als Ergebnis ein neues HTML-Formular:

```
inputForm :: IO HtmlForm
inputForm = return $ form "inputForm"
  [HtmlText "Enter your name:",textfield ref "",button "Submit" handler]
  where
    ref free
    handler env = return $ form "helloForm" [HtmlText $ "Hello, "++(env ref)++"!"]
```

Über die Funktion `textfield` wird ein Textfeld erzeugt, welches mit der CGI-Referenz `ref` verknüpft ist. `button` erzeugt hingegen einen Submit-Button, dessen Event-Handler von der Funktion `handler` des Typs `HtmlHandler` repräsentiert wird. Sowohl `inputForm` als auch die rechte Seite der `handler`-Funktion sind vom Typ `IO HtmlForm`. Der Datentyp `HtmlForm` verknüpft den Inhalt einer Webseite (eine Liste von HTML-Ausdrücken vom Typ `[HtmlExp]`) mit den restlichen, zur Konstruktion eines vollständigen HTML-Dokuments notwendigen Daten wie dem Dokumenttyp und dem Header. Ein vollständiges HTML-Formular (in Form einer I/O-Aktion vom Typ `IO HtmlForm`) kann dann mit dem PAKCS-Tool `makecurrycgi` als CGI-Skript installiert werden. Dieses generiert bei Anfrage die für den Webbrowser verständliche textuelle Darstellung des HTML-Dokuments, wandelt die von der `HTML`-Bibliothek erzeugten, abstrakten CGI-Referenzen in tatsächliche Attribute um und verarbeitet die beim Abschicken des Formulars spezifizierten Aktionen. Auf diese Weise können dynamische Webseiten konstruiert werden, die basierend auf den Benutzereingaben eines Formulars erst zum Zeitpunkt der Anfrage durch den Klienten generiert werden.

Eine Erweiterung der von der Bibliothek `HTML` beschriebenen Schnittstelle wird durch die Bibliothek `WUI` spezifiziert. Diese unterstützt den Entwurf grafischer Webobflächen (Web User Interfaces, WUIs) für typsichere Formulare als zusätzliche Abstraktionsebene bei der Entwicklung dynamischer Webseiten [Han06]. Typsicherheit bedeutet in diesem Fall, dass die Eingabelemente eines Formulars nur Werte eines bestimmten Typs akzeptieren und dass

[8] Curry unterstützt ein monadisches I/O-Konzept, welches identisch zu Haskells I/O-Konzept ist [HK12][Han12]. Hierbei ist „`IO` *T*“ der Typ einer I/O-Aktion, die Werte des Typs *T* zurückliefert.

das Absenden des Formulars im Falle einer Typverletzung unterbunden wird. Zu diesem Zweck definiert die Bibliothek `WUI` typisierte Widgets für atomare oder zusammengesetzte Eingabeelemente, aus denen entsprechende Formulare konstruiert werden können. Ein Widget ist dabei immer mit einem eindeutigen Typ (in Form eines Typparameters), einem Rendering (welches das Aussehen des Eingabeelements spezifiziert), einer Fehlermeldung (welche im Falle einer nicht validen Benutzereingabe angezeigt wird) und einer zusätzlichen Bedingung an den Eingabewert verknüpft. Für die Manipulation von `String`- oder `Bool`-Werten stellt die Bibliothek `WUI` beispielsweise folgende Widgets zur Verfügung:

```
wString :: WuiSpec String
wBool   :: WuiSpec Bool
```

Hierbei beschreibt `WuiSpec` *T* den Typ eines Widgets, mit dem Werte des Typs *T* modifiziert werden können. Für die Konstruktion zusammengesetzter Widgets definiert die Bibliothek sogenannte WUI-Kombinatoren. Ein Widget, das aus drei beliebig typisierten Eingabeelementen besteht, kann z. B. über die Funktion `wTriple` spezifiziert werden:

```
wTriple :: WuiSpec a -> WuiSpec b -> WuiSpec c -> WuiSpec (a,b,c)
```

`wTriple` kann nun beispielsweise verwendet werden, um ein Widget zu definieren, welches die Manipulation dreier `String`-Werte erlaubt:

```
wStringTriple :: WuiSpec (String,String,String)
wStringTriple = wTriple wString wString wString
```

Sowohl für atomare als auch für zusammengesetzte Widgets definiert die `WUI`-Bibliothek Standardwerte bezüglich des Renderings, der Fehlermeldung und der zusätzlichen Validierungsbedingung. Mit den Operatoren `withRendering`, `withError` und `withCondition` können diese Standardwerte jedoch beliebig überschrieben werden:

```
wStringWithMaxLength :: Int -> WuiSpec String
wStringWithMaxLength n = wString `withCondition` hasMaxLength
  where hasMaxLength s = (length s) <= n
```

Der Aufruf `wStringWithMaxLength 10` erzeugt beispielsweise ein Widget, welches nur `String`-Werte akzeptiert, deren Länge `10` nicht überschreitet. Dazu wird das Prädikat `hasMaxLength` an das Widget `wString` gekoppelt.

Mithilfe von Widgets können also komplexe, typsichere Formulare mit nahezu beliebig anpassbaren Eingabeelementen konstruiert werden. Für die Einbindung eines WUI-Formulars (in Form eines einzelnen Widgets) in ein vorhandenes HTML-Formular kann beispielsweise die Funktion `wui2html` verwendet werden:

```
wui2html :: WuiSpec a -> a -> (a -> IO HtmlForm) -> (HtmlExp,WuiHandler)
```

`wui2html` erhält ein Widget eines beliebigen Typs `a`, einen initialen Wert für die Eingabeelemente des Widgets und ein Formular, welches die eingegebenen Daten verarbeitet, falls alle Eingaben valide sind, und erzeugt daraus einen HTML-Ausdruck und einen Handler. Hierbei ist der Typ `WuiHandler` ein Wrapper für den Typ `HtmlHandler`. Der HTML-Ausdruck enthält schließlich die HTML-Repräsentation des Widgets, während der Handler mit einem Button verknüpft werden kann, der das Absenden des Formulars triggert.

2.2. Das Webbframework Spicey

2.2.1. Grundlagen

Das Spicey-Framework[9] ist ein von Michael Hanus und Sven Koschnicke in Curry implementiertes und für die Entwicklung mit Curry konzipiertes Webframework, welches im Wesentlichen auf den im letzten Abschnitt beschriebenen Konzepten zur deklarativen Webprogrammierung in Curry basiert. Von seinen Entwicklern wird Spicey als „Entity-Relationship-basiertes“ Webframework bezeichnet, was daher rührt, dass es aus der Spezifikation des einer Anwendung zugrunde liegenden Datenmodells in Form eines sogenannten Entity-Relationship-Modells per Scaffolding[10] den kompletten Quelltext einer initialen, ausführbaren Webanwendung erzeugt, welche die Manipulation der Daten über den Webbrowser ermöglicht. Entity-Relationship-Modelle (auch ER-Modelle) [Che76] sind ein Konzept der semantischen Datenmodellierung, bei dem der für ein System relevante Ausschnitt der Welt durch Entitäten, Attribute und Beziehungen mit Kardinalitätsrestriktionen beschrieben wird. Ein solches Modell kann bspw. dafür verwendet werden, um die Struktur und etwaige Integritätsbedingungen des Datenbankschemas einer Anwendung zu spezifizieren. Typischerweise werden ER-Modelle dazu in sogenannten ER-Diagrammen visualisiert. Abbildung 2.1 zeigt ein minimales Beispiel eines solchen ER-Diagramms in Chen-/(min,max)-Notation [Che76, Abr76].

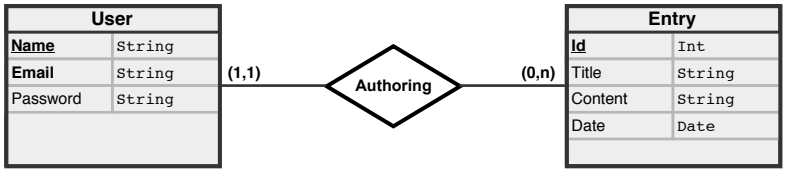


Abbildung 2.1.: Beispiel eines Entity-Relationship-Diagramms

Das ER-Diagramm spezifiziert das Datenmodell eines Web-Blogs, in dem Nutzer mit einem eindeutigen Namen (Primärschlüssel), einer eindeutigen E-Mail-Addresse und einem Passwort beliebig viele (Blog-)Einträge erstellen können. Ein Eintrag besteht dabei aus einer eindeutigen ID, einem Titel, dem Inhalt des Eintrags und dem Datum der Erstellung.

Zur Umsetzung mit dem Spicey Framework wird das ER-Diagramm nun zunächst in einen sogenannten `ERD`-Term übersetzt:

```
(ERD "SpiceyBlog"

  [Entity "User"
    [Attribute "Name"     (StringDom Nothing) PKey   False
    ,Attribute "Email"    (StringDom Nothing) Unique False
    ,Attribute "Password" (StringDom Nothing) NoKey  False]

  ,Entity "Entry"
    [Attribute "Id"       (IntDom    Nothing) PKey   False
```

[9] http://www.informatik.uni-kiel.de/~pakcs/spicey/
[10] http://en.wikipedia.org/wiki/Scaffold_(programming)

```
    ,Attribute "Title"    (StringDom Nothing) NoKey  False
    ,Attribute "Content"  (StringDom Nothing) NoKey  False
    ,Attribute "Date"     (DateDom   Nothing) NoKey  False]]

  [Relationship "Authoring"
    [REnd "User"  "hasAuthor"     (Exactly 1)
    ,REnd "Entry" "isTheAuthorOf" (Between 0 Infinite)]]

)
```

Listing 2.1: Beispiel eines ERD-Terms

Dazu werden sukzessive alle Entitäten, Attribute und Beziehungen mitsamt der spezifizierten Integritätsbedingungen (Eindeutigkeit von Attributen, Kardinalitäten, etc.) auf entsprechende Curry-Datentypen abgebildet (siehe [BHM08] für eine ausführliche Beschreibung dieser Datentypen). Aus dem ERD-Term erzeugt Spicey dann anschließend den kompletten Quelltext einer ausführbaren Anwendung, welche in Form eines CGI-Skripts auf einem Webserver (z. B. Apache) installiert werden kann.

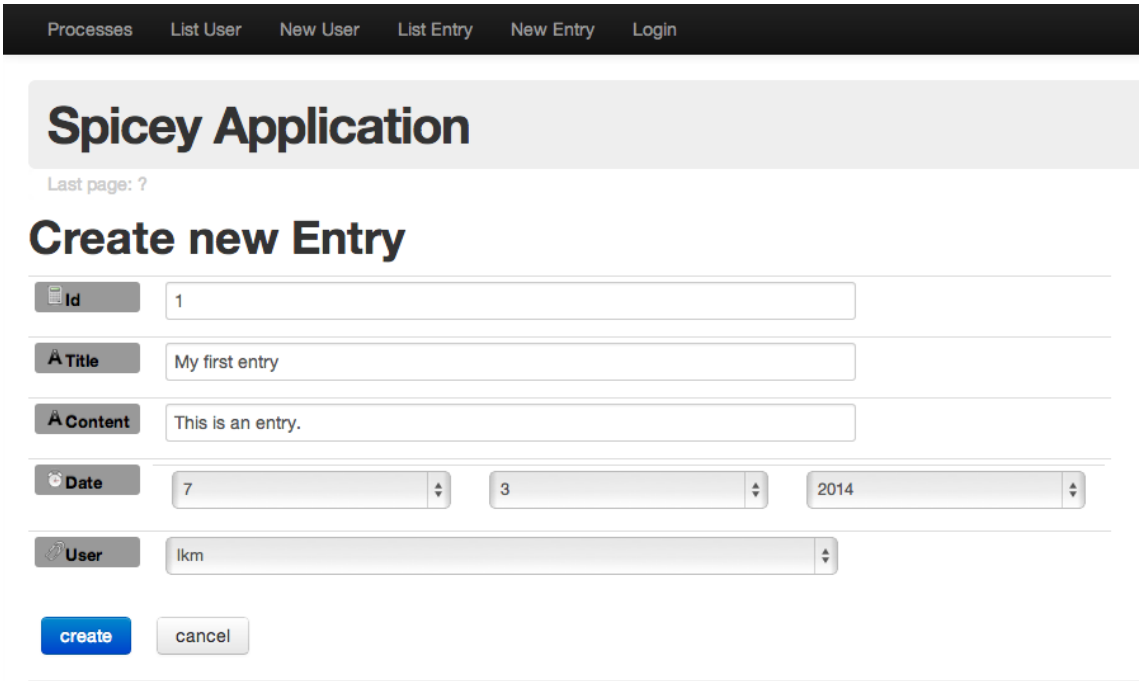


Abbildung 2.2.: Webschnittstelle einer Spicey-Anwendung

Die initiale Anwendung implementiert für alle im ER-Modell spezifizierten Entitäten Webschnittstellen, mit denen die zugrunde liegenden Daten erzeugt und manipuliert werden können. Abbildung 2.2 zeigt bspw. die von Spicey aus dem ERD-Term in Listing 2.1 generierte Webschnittstelle für die Erstellung eines neuen (Blog-)Eintrags. Zusätzlich zu den Weboberflächen für die Datenmanipulation implementiert Spicey außerdem Schnittstellen für viele typische Funktionalitäten herkömmlicher Webanwendungen. Dies umfasst beispielsweise die Verwaltung von Sitzungsdaten (Session Management), Authentifizierung und Autorisierung [HK12].

2.2.2. Struktur einer Spicey-Anwendung

Die Struktur von Spicey-Anwendungen basiert im Wesentlichen auf dem Model-View-Controller-Architekturmuster [KP88]. Das MVC-Muster beschreibt die Aufteilung von (Software-)Systemen in drei funktional getrennte Komponenten, die über bestimmte Schnittstellen miteinander kommunizieren. Diese drei Komponenten sind das Datenmodell (Model), die Präsentationsschicht (View) und die Steuerungsschicht (Controller):

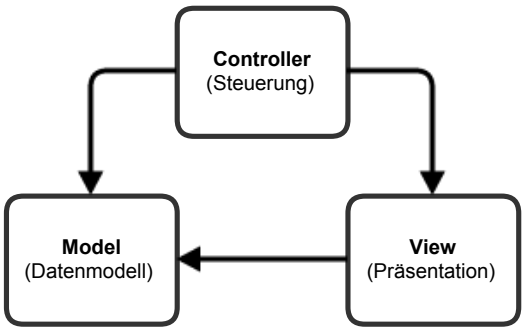


Abbildung 2.3.: Komponenten des Model-View-Controller-Architekturmusters

Datenmodell Das Datenmodell repräsentiert die dem System zugrunde liegende Daten, spezifiziert eine Schnittstelle zu deren Verarbeitung und ist unabhängig von der Steuerung und der Präsentation.

Präsentation Die Präsentationsschicht greift auf die Schnittstelle des Modells zu und stellt die dort beschriebenen Daten dar. Sie definiert das Front-End des Systems, mit dem der Nutzer interagieren kann.

Steuerung Die Steuerungsschicht ist für die Verarbeitung der Benutzeraktionen in der Präsentationsschicht zuständig. Entsprechend dieser Aktionen verändert sie den Zustand des Modells und verwaltet die verfügbaren Präsentationen.

Ziel dieser Aufteilung ist die strikte Trennung der zu den drei Komponenten assoziierbaren Funktionalitäten, um ein möglichst anpassungsfähiges Gesamtsystem zu erhalten. Die Verwendung des MVC-Architekturmusters als Basis des Entwurfs soll die Skalierbarkeit, Modifizierbarkeit und Wartbarkeit eines Systems erhöhen und die Wiederverwendbarkeit einzelner Komponenten gewährleisten. Es wäre beispielsweise vorstellbar, dass eine Anwendung um eine zusätzliche Darstellungsart des zugrunde liegenden Datenmodells ergänzt werden soll (z. B. eine Oberfläche für Android-Geräte, zusätzlich zu einer herkömmlichen Oberfläche für den Webbrowser). Unter Verwendung des MVC-Musters muss hierbei im besten Fall nur die Präsentationsschicht der Anwendung erweitert werden. Steuerung und Modell können bei geeigneter Spezifikation der Schnittstellen von der neuen Darstellungsvariante wiederverwendet werden.

Spiceys Adaption des MVC-Musters spiegelt sich vor allem in der Modularisierung des durch den Scaffolding-Prozess erzeugten Quelltextes wieder [HK12]. So weisen Spicey-Anwendungen eine der MVC-Aufteilung entsprechende Verzeichnisstruktur auf, welche um Konfigurationseinheiten und allgemeine Komponenten ergänzt wird (z. B. Komponenten für

die Umsetzung der Authentifizierung und Autorisierung). Der wesentliche Ausschnitt dieser Verzeichnisstruktur wird zweckmäßig im Rahmen der Implementierung in Abschnitt 4.2 beschrieben. Eine Erläuterung des konkreten Zusammenspiels der drei MVC-Komponenten im Kontext von Spicey-Anwendungen erfolgt im nächsten Abschnitt.

2.2.3. Funktionsweise einer Spicey-Anwendung

Wie bereits in Abschnitt 2.2.1 erwähnt wurde, werden Spicey-Anwendungen in Form von CGI-Skripten installiert. Dieser Vorgang beruht im Wesentlichen auf dem durch die `HTML`-Bibliothek spezifizierten Konzept zur Installation von HTML-Formularen (siehe Abschnitt 2.1.2). Entsprechend diesem Konzept implementiert jede Spicey-Anwendung eine `main`-Funktion vom Typ `IO HtmlForm`, welche die Anfragen des Klienten verarbeitet und als Ergebnis ein HTML-Formular generiert, das anschließend über die CGI-Schnittstelle an den Klienten zurückgesendet wird. Die `main`-Funktion wird vom Modul `Main`[11] bereitgestellt und delegiert die Konstruktion des Formulars an einen Dispatcher (in Form der `dispatcher`-Funktion; siehe Listing 2.2).

```
1  main :: IO HtmlForm
2  main = dispatcher
3
4  dispatcher =
5    do (url,_)    <- getControllerURL
6       controller <- nextControllerRefInProcessOrForUrl url
7                     >= maybe (displayError "Illegal URL!")
8                        getController
9       form <- getForm controller
10      return form
```

Listing 2.2: `Main` (Funktionen `main` und `dispatcher`)

Die `dispatcher`-Funktion ermittelt zunächst den für die Verarbeitung der Anfrage zuständigen Controller. Controller in Spicey sind I/O-Aktionen, die den Inhalt einer View (bzw. eines HTML-Dokuments) zurückliefern:

```
type Controller = IO View
```

Hierbei werden Views von einer Liste von HTML-Ausdrücken repräsentiert, die den Inhalt des HTML-Dokuments beschreiben:

```
type View = [HtmlExp]
```

Für die Ermittlung des zuständigen Controllers werden in Zeile 5 zunächst die Parameter des CGI-Skripts (welche den URL-Pfad repräsentieren) ausgelesen. Auf Basis des Pfads wird in den Zeilen 6 bis 8 anschließend der Controller bestimmt. Die Definition der verfügbaren

[11] Curry-Programme werden (wie Haskell-Programme) in Modulen strukturiert. Ein Modul *M* wird dabei immer in einer Datei `M.curry` abgelegt, deren exportierte Datentypen und Funktionen über das `import`-Schlüsselwort in andere Module eingebunden werden können. Zyklische Abhängigkeiten sind hierbei nicht erlaubt [Han12].

Routen bzw. der unterstützten URLs und deren Mapping auf konkrete Spicey-Controller erfolgt in den Modulen `ControllerMapping` und `RoutesData` im Unterverzeichnis `config/`. In den Zeilen 9 und 10 wird abschließend das eigentliche HTML-Formular aus dem vom Controller zurückgelieferten View-Block des Typs `[HtmlExp]` generiert und als I/O-Aktion vom Dispatcher zurückgegeben. Die Funktion `getForm` bettet den Inhalt der View in das Standard-Rahmenlayout ein und ergänzt den Body des HTML-Dokuments um den Dokumenttyp, das HTML-Wurzelelement und etwaige Kopfdaten (z. B. CSS-Stylesheets).

Die „Ausführung“ eines Controllers entspricht in Spicey im Allgemeinen der Ausführung einer zugehörigen, möglicherweise nullstelligen Controller-Funktion vom Typ `Controller`, deren letzter Befehl stets aus der Berechnung der nächsten View besteht. Abbildung 2.4 zeigt den allgemeinen Zusammenhang zwischen den MVC-Komponten einer Spicey-Anwendungen (vgl. [Kos08]):

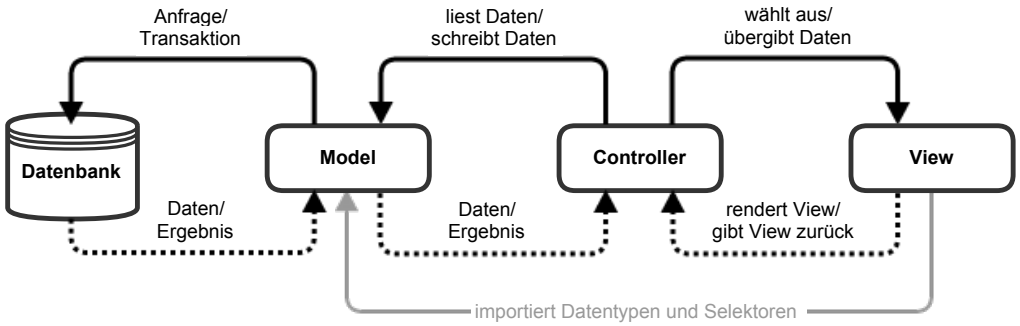


Abbildung 2.4.: Funktionsweise einer Spicey-Anwendung

Die Controller-Funktion wird entweder direkt vom Dispatcher aufgerufen oder als Folge auf das Absenden eines Formulars über einen HTML-Handler (siehe Abschnitt 2.1.2). Im ersten Fall werden typischerweise Daten über das Model aus der Datenbank ausgelesen, damit diese anschließend von der ausgewählten View dargestellt werden können. Im letzteren Fall wird das Datenmodell zusätzlich mit den Daten des Formulars aktualisiert (möglicherweise mit vorheriger Validierung der Eingaben). Auch dazu greift der Controller mittels Model-Schnittstelle auf die Datenbank zu. Das Model wird dabei standardmäßig von einem Modul repräsentiert, welches auf Basis des ER-Modells abstrakte Datentypen, Selektoren und Datenbankoperationen (Anfragen, Transaktionen) für alle spezifizierten Entitäten implementiert (siehe Abschnitt 4.1.2). In einem letzten Schritt wählt der Controller anhand der Ergebnisse des Model-Zugriffs (Daten, Transaktionsfehler, etc.) die nächste darzustellende Weboberfläche aus. Dazu ruft er eine zugehörige View-Funktion auf, welche die Ergebnisse als Eingabe erhält und den Inhalt des HTML-Dokuments als Liste von HTML-Ausdrücken rendert. Diese Liste wird anschließend vom Controller als I/O-Aktion zurückgegeben. Im Gegensatz zu anderen Webframeworks wie z. B. Ruby on Rails[12] oder dem Play-Framework[13] basiert Spiceys View-Konzept also nicht auf einer Template-Engine, sondern ist komplett in Curry implementiert. Um die Daten der Model-Schicht verarbeiten und darstellen zu können importiert die View-Schicht zudem die benötigten Datentypen und Selektoren aus der Model-Schicht.

[12] `http://rubyonrails.org/`
[13] `http://www.playframework.com/`

Die weiteren von Spicey unterstützten Konzepte wie Authentifizierung, Autorisierung und Session Management werden teilweise in Kapitel 4 beschrieben. Eine ausführliche Beschreibung der Implementierung des Spicey Framework inklusive dieser Bestandteile ist in [HK12] zu finden.

2.3. Bootstrap

2.3.1. Grundlagen und Entstehung

Bootstrap[14] ist ein Front-End-Framework für die effiziente Entwicklung und Gestaltung auf HTML5 und CSS3 basierender Webanwendungen, welches 2011 von Twitter[15] als Open-Source-Projekt veröffentlicht wurde [Ott11]. Ursprünglich enstand Bootstrap aus der Idee, eine zentrale Front-End-Bibliothek für die Entwicklung der internen Analyse- und Verwaltungswerkzeugen des Twitter-Dienstes zu entwerfen. Die Entwicklung der Weboberflächen dieser Werkzeuge basierte zunächst auf unterschiedlichen Bibliotheken, was zu Inkonsistenz, geringer Skalierbarkeit und einem hohen Wartungsaufwand führte. Um diesem Umstand entgegenzuwirken wurde die Entwicklung einer eigenen, zunächst internen Bibliothek angestoßen, die alle bisher verwendeten Bibliotheken ersetzen sollte. Daraus enstand (teilweise begleitend zu Twitters erster Hackweek) eine erste Version von Bootstrap, die schließlich im August 2011 von Twitter als Open-Source-Projekt veröffentlicht wurde. Dieses Projekt befindet sich seitdem in anhaltender Entwicklung.

2.3.2. Bestandteile des Frameworks

Bootstrap war von Beginn an für die Unterstützung von HTML5 [W3C14] und CSS3 [W3C11] konzipiert. Basierend auf diesen Techniken definiert Bootstrap Hilfsmittel und Vorlagen für die Gestaltung von Webseiten. Dazu gehören beispielweise ein eigenes Grid-System zur Festlegung der grundlegenden Struktur einer Seite, Vorlagen für einheitliche Typografie und Textformatierung und eine große Menge an wiederverwendbaren Layout-Komponenten. Zusätzlich unterstützt Bootstrap die Verwendung von JavaScript-Plugins zur Erweiterung der Standardkomponenten.

▷ Responsives Grid-System und grundlegende CSS-Styles

Standardmäßig definiert Bootstrap ein zwölfspaltiges Grid-System, welches sich im Sinne des Responsive Webdesign [Mar11] über die Verwendung von CSS-Grid-Klassen an die Auflösung bestimmter Endgeräte anpassen lässt. Dazu unterscheidet Bootstrap zwischen Mobiltelefonen, Tablets sowie niedrig- und hochauflösenden Desktop-PCs. Die Definition der Grid-Klassen erfolgt in den von Bootstrap bereitgestellten CSS-Stylesheets, welche zudem einheitliche Stildefinitionen für alle fundamentalen HTML-Elemente enthalten. Dies umfasst beispielsweise Tabellen, Formularelemente und Buttons.

[14] http://getbootstrap.com/
[15] http://twitter.com/

▷ **Wiederverwendbare Layout-Komponenten**

Zusätzlich zu den Stildefinitionen für die regulären HTML-Elemente definiert Bootstrap auch Schablonen für komplexere, zusammengesetzte Layout-Komponenten. Dazu gehören beispielsweise gruppierte Buttons, Buttons mit Dropdown-Menüs, Tabs, Navigationsleisten, Breadcrumb-Navigation, Paginierungselemente, Labels, Hinweismeldungen und Fortschrittsbalken.

▷ **JavaScript-Plugins**

Viele der von Bootstrap bereitgestellten Komponenten lassen sich unter Zuhilfenahme von JavaScript um zusätzliche Funktionalitäten erweitern. Basierend auf dem JavaScript-Framework jQuery[16] stellt Bootstrap dafür eine Reihe von Plugins zur Verfügung, die beispielsweise die Verwendung zusätzlicher Oberflächenelemente wie Tooltips, Popovers oder Dialogfenster erlauben oder das Ein- und Ausblenden bestimmter Inhalte ermöglichen (z. B. über Dropdown-Menüs, Tabs oder zusammenklappbare Container).

2.3.3. Verwendungsbeispiel

Um die Grundfunktionen von Bootstrap verwenden zu können, genügt zunächst das Einbinden des zentralen Bootstrap-CSS-Stylesheets in das entsprechende HTML-Dokument. Auf diese Weise werden bereits die Stildefinitionen für die regulären HTML-Elemente übernommen. Über die Verknüpfung von HTML-Elementen mit bestimmten CSS-Klassen entsprechend der Bootstrap-Dokumentation lassen sich dann weitere Anpassungen vornehmen oder bestimmte Layout-Komponenten definieren. Die Verwendung der CSS-Klassen ist dabei ein wesentlicher Bestandteil der Entwicklung mit Bootstrap.

```
<h3>Please sign in</h3>
<div>
  <div>
    <form role="form">
      <div>
        <label>Username</label>
        <input type="text" placeholder="Username">
      </div>
      <div>
        <label>Password</label>
        <input type="password" placeholder="Password">
      </div>
      <button type="submit">Sign in</button>
      <button type="reset">Cancel</button>
    </form>
  </div>
</div>
```

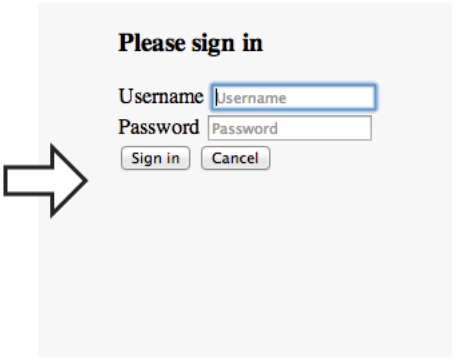


Abbildung 2.5.: Ausschnitt eines HTML5-Dokuments ohne Bootstrap

Abbildung 2.5 zeigt beispielsweise die Spezifikation eines typischen Anmeldeformulars innerhalb eines HTML5-Dokuments und dessen Darstellung im Webbrowser Google Chrome. Hierbei wurde (abgesehen von der Hintergrundfarbe) zunächst keine Modifikation der visuellen Darstellung über CSS-Regeln vorgenommen. Abbildung 2.6 zeigt nun den

[16] http://jquery.com/

gleichen Ausschnitt unter Verwendung des Bootstrap-CSS-Stylesheets (im Kopf des HTML-Dokuments eingebunden), wobei manche HTML-Elemente entsprechend den Gestaltungsvorlagen der Dokumentation mit CSS-Klassen verknüpft wurden:

```
<h3>Please sign in</h3>
<div class="panel panel-default">
  <div class="panel-body">
    <form role="form">
      <div class="form-group">
        <label>Username</label>
        <input type="text" class="form-control" placeholder="Username">
      </div>
      <div class="form-group">
        <label>Password</label>
        <input type="password" class="form-control" placeholder="Password">
      </div>
      <button type="submit" class="btn btn-primary">Sign in</button>
      <button type="reset" class="btn btn-default">Cancel</button>
    </form>
  </div>
</div>
```

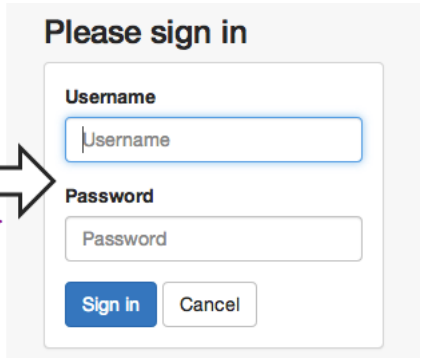


Abbildung 2.6.: Ausschnitt eines HTML5-Dokuments mit Bootstrap

Die ersten beiden `div`-Container definieren eine Panel-Komponente (eine Box mit weißem Hintergund), welche das eigentliche Formular enthält. Die Klassen `form-group` und `form-control` steuern das Erscheinungsbild, die Positionierung und die Abstandsflächen der Formularelemente. Das Aussehen von Buttons (Basisklasse `btn`) kann über Kontextklassen (hier `btn-primary` und `btn-default`) variiert werden. Zudem ist zu erkennen, dass Bootstrap auch globale Modifikationen des Erscheinungbilds vornimmt (z. B. bzgl. Schriftart und Schriftgröße).
Alle von Bootstrap vorgegebenen Einstellungen können an die Bedürfnisse des Entwicklers angepasst werden. Für die Konfiguration des zentralen Stylesheets stellt die Dokumentation dafür beispielsweise eine „Customize“-Option zur Verfügung. Alternativ können Nutzer auch direkt den Quellcode von Bootstrap, der im Wesentlichen in Form von LESS[17]-Stylesheets vorliegt, modifizieren und die benötigten CSS-Dateien per Hand kompilieren.

2.4. CodeMirror

2.4.1. Grundlagen und Entstehung

CodeMirror[18] ist ein in JavaScript implementierter, anpassbarer Texteditor für den Webbrowser, der seit 2007 von Marijn Haverbeke entwickelt wird. CodeMirror entstand aus dem Vorhaben, eine Erweiterung für die HTML-`textarea`-Komponente zu entwickeln, welche das Erstellen und Bearbeiten von Quelltext im Browser effizient und einfach gestalten sollte. Dazu war zunächst nur die Unterstützung von Syntax-Highlighting und automatischer Code-Einrückung vorgesehen [Hav07]. Mittlerweile verfügt CodeMirror über eine eigene Programmierschnittstelle (API), sodass der Funktionsumfang des Editors durch das Einbinden von Addons nahezu beliebig erweitert werden kann.

Die wichtigsten Features von CodeMirror (Version 3.22) umfassen:

[17] `http://www.lesscss.de/`
[18] `http://codemirror.net/`

- Native (Syntax-Highlighting-)Unterstützung für 60 verschiedene Programmiersprachen. Weitere Sprachen können über ein integriertes Sprachmodus-System hinzugefügt werden.
- Unterstützung für die automatische Einrückung von Code.
- Unterstützung von Autovervollständigung, Code-Folding, Bracket Matching und vieler weiterer Funktionalitäten typischer Quelltext-Editoren durch das Einbinden von Addons.
- Anpassbare Tastenbelegungen (Shortcuts) und Unterstützung der Vim- und Emacs-Modi.
- Bereitstellung einer umfassenden Programmierschnittstelle für die Anpassung der CodeMirror-Instanzen und die Entwicklung von Addons.

2.4.2. Verwendungsbeispiel

Um CodeMirror verwenden zu können müssen zunächst die grundlegenden Skripte und Stylesheets in das entsprechende HTML-Dokument eingebunden werden. Dazu gehören das Basis-Skript `codemirror.js` und das Basis-Stylesheet `codemirror.css`. Zusätzlich wird in der Regel noch mindestens ein Sprachmodus-Skript eingebunden, über welches die Regeln für das Syntax-Highlighting definiert werden (z. B. `curry.js`).
Für die Erzeugung einer CodeMirror-Instanz gibt es grundsätzlich mehrere Möglichkeiten. Die gängigste Variante ist die Verwendung der `CodeMirror.fromTextArea`-Methode. Diese erhält als erstes Argument den DOM-Knoten einer bereits im HTML-Dokument enthaltenen `textarea` und als zweites optionales Argument ein Konfigurationsobjekt, über das zusätzliche Einstellungen vorgenommen werden können. Die Verwendung der Methode wird in Abbildung 2.7 demonstriert:

Abbildung 2.7.: Erzeugung einer CodeMirror-Instanz

Die `fromTextArea`-Methode ersetzt hier die `textarea` mit der ID `editor` durch eine CodeMirror-Instanz mit den im Konfigurationsobjekt beschriebenen Eigenschaften. Das Objekt spezifiziert in diesem Fall die Darstellung der Zeilennummern im Editor, die Verwendung des (zuvor mittels eines Skripts eingebundenen) `curry`-Sprachmodus und des `smap`-Themes (ein zusätzliches CSS-Stylesheet, welches die Einfärbung der in `curry.js`

definierten Schlüsselworte festlegt) und die Hervorhebung der momentan ausgewählten Zeile. Die `styleActiveLine`-Option wird dabei von einem Addon bereitgestellt, welches über das Skript `active-line.js` in das HTML-Dokument eingebunden werden muss.
Als Resultat wird der rechts in Abbildung 2.7 dargestellte Editor erzeugt. Das Erscheinungsbild des Editors (Größe, Ränder, Ecken, etc.) kann anschließend durch das Überschreiben oder Erweitern der CSS-Klasse `CodeMirror` modifiziert werden.

3. Entwurf der Webanwendung

Dieses Kapitel befasst sich mit dem Entwurf der Webanwendung Smap. Dazu werden zunächst die zentralen funktionalen und nichtfunktionalen Anforderungen an die zu entwickelnde Software ermittelt. Unter Zuhilfenahme der Ergebnisse dieser Anforderungsanalyse werden dann im nächsten Abschnitt die funktionalen Komponenten der Anwendung identifiziert und mittels des konzeptionellen Aufbaus der Software und ihrer Funktionsweise zueinander in Beziehung gesetzt und erläutert.

Wie in Abschnitt 2.2 erwähnt wurde, dient die Spezifikation des einer Anwendung zugrunde liegenden Datenmodells in Form eines Entity-Relationship-Modells als Einstiegspunkt für die Entwicklung einer Spicey-Applikation. Aus diesem Grund wird im letzten Abschnitt dieses Kapitels aus den Anforderungen und den funktionalen Komponenten der Anwendung ein solches konzeptionelles Datenmodell hergeleitet, in einem Entity-Relationship-Diagramm dargestellt und dessen Bestandteile erläutert. Die tatsächliche Umsetzung dieses Modells mit Spicey bildet dann im folgenden Kapitel die Grundlage für die Implementierung des Entwurfs.

3.1. Zentrale Anforderungen an die Webanwendung

3.1.1. Funktionale Anforderungen

Ziel ist die Entwicklung einer Anwendung, die die folgenden Spezifikationen und Schnittstellen unter Berücksichtigung der Rahmenbedingungen (siehe Abschnitt 3.1.2) implementiert:

▷ **Programme als zentrale Objekte der Anwendung**

Die zentralen Objekte der zu entwickelnden Webanwendung sind Programme. Ein Programm repräsentiert zunächst einen Quelltext – verfasst in einer bestimmten Programmiersprache – der eine Folge von Anweisungen enthält und die Funktionalität des Programms beschreibt. Der Quelltext kann dabei möglicherweise in mehreren Versionen vorliegen.
Zu einem Programm ist weiterhin eine Menge von Metadaten gegeben, die das Programm identifizieren und beschreiben. Dazu gehören beispielsweise ein Titel und eine Beschreibung, aber auch Informationen über die Programmiersprache, in der der Quelltext verfasst wurde, und Angaben über den Autor des Programms. Außerdem enthalten die Metadaten eine Menge von Schlagwörtern (Tags), die das Programm klassifizieren.

▷ **Erstellung von Programmen**

Nutzer der zu entwickelnden Webanwendung können Programme erstellen. Dazu muss der Nutzer zunächst aus einer Liste die gewünschte Programmiersprache wählen. Die Anwendung stellt dem Nutzer daraufhin eine Weboberfläche zur Verfügung, die die Eingabe von Quelltext dieser Programmiersprache ermöglicht.
Nach Eingabe des Quelltextes darf ein befugter Nutzer den Quelltext in Form eines Programms abspeichern. Dazu müssen möglicherweise zunächst noch Angaben zu den Metadaten des Programms gemacht werden. Anschließend wird das Programm in der Datenbank abgelegt.

▷ **Bearbeitung von Programmen**

Hat ein Nutzer ein neues Programm erstellt, liegt der entsprechende Quelltext zunächst in einer einzigen Version vor. Möchte ein Nutzer nun eines seiner Programme bearbeiten, kann er die aktuellste Version des zugehörigen Quelltextes in eine geeignete Weboberfläche laden und dort verändern. Das Abspeichern des bearbeiteten Quelltextes bewirkt das Anlegen einer neuen Version des Quelltextes zu dem entsprechenden Programm.

▷ **Verwaltung von Programmen**

Die von Nutzern erstellten Programme sind im Regelfall auch allen anderen Nutzern zugänglich. Zu diesem Zweck stellt die Anwendung eine Suchfunktion zur Verfügung, die Programme mit bestimmten Attributen und Eigenschaften filtert und die Resultate sortiert und geordnet in einer Listenansicht darstellt. Die Suchfunktion erlaubt dabei insbesondere auch das Filtern von Programmen mit bestimmten Tags.
Da die Listenansicht nicht alle Daten zu einem Programm enthalten kann, ist für jedes Programm-Objekt noch eine Einzelansicht abrufbar, die alle zu dem jeweiligen Programm bekannten Informationen darstellt oder zugänglich macht. Dies schließt beispielsweise den Quelltext und Daten zur Versionshistorie des Programms ein.

▷ **Ausführung von Programmen**

Die Anwendung erlaubt darüber hinaus das Ausführen von Programmen. Zur Ausführung eines Programms, dessen Quelltext in einer bestimmten (unterstützten) Programmiersprache verfasst wurde, wählt der Nutzer zunächst ein Ausführungssystem[1] aus einer begrenzten Menge von Implementierungen der Sprache. Nach der Ausführung wird dem Nutzer das Ergebnis sichtbar gemacht, welches zudem Informationen darüber enthält, ob die Ausführung des Programms mit dem gewählten Ausführungssystem erfolgreich war

[1] Der Begriff „Ausführungssystem“ wird im Kontext dieser Arbeit als Synonym für die Implementierung einer Programmiersprache verwendet. Beispiele für solche Ausführungssysteme sind die bereits in Abschnitt 2.1.1 erwähnte Curry-Implementierung PAKCS oder der Glasgow Haskell Compiler (GHC, `https://www.haskell.org/ghc/`) für die Programmiersprache Haskell.

oder fehlgeschlagen[2] ist.
Um Programme auszuführen, ist es nicht notwendig, diese vorher zu speichern. Stattdessen erlaubt es die Anwendung, Programme während ihrer Erstellung oder während der Bearbeitung des Quelltextes auszuführen und zu testen. Natürlich können aber auch bereits gespeicherte Programme ausgeführt werden.

▷ **Löschen von Programmen**

In der Datenbank abgelegte Programme können von befugten Nutzern wieder gelöscht werden. Da diese Operation die zugehörigen Daten endgültig und unwiderruflich aus der Datenbank entfernt, muss der Nutzer die Durchführung der Aktion bestätigen.

▷ **Registrierung und Authentifizierung von Nutzern**

Für den Zugriff auf bestimmte Funktionalitäten der Anwendung müssen sich Nutzer authentifizieren. Dazu stellt die Anwendung ein Webformular zur Verfügung, mit der sich Nutzer unter Angabe bestimmter Informationen im System registrieren können. Diese Informationen beinhalten dabei zumindest den für die Identifikation des Nutzers und die Authentifizierung notwendigen Benutzernamen und ein Passwort.

▷ **Rollensystem und Autorisierung**

Zur Verwaltung der Autorisierung und Erteilung von Zugriffsrechten auf Operationen implementiert die Anwendung ein Rollensystem, welches auf der Authentifizierung von Nutzern basiert und zwischen folgenden Benutzergruppen unterscheidet:

- **Gäste** sind Nutzer, die sich nicht authentifiziert haben.
- **Reguläre Nutzer** (auch Standard-Nutzer) sind Nutzer, die sich authentifiziert haben, aber über keine speziellen Rechte verfügen.
- **Administratoren** sind authentifizierte Nutzer, die über spezielle (erweiterte) Rechte verfügen.

Abbildung 3.1 zeigt die initiale Spezifikation der Zugriffsrechte auf Programm-Operationen. Diese sieht vor, dass Gäste zwar die Weboberfläche zur Erstellung von Programmen nutzen können (und somit auch Zugriff auf die Ausführungskomponente haben), Programme aber nicht gespeichert werden dürfen. Authentifizierten Nutzern ist es hingegen erlaubt, Programme in der Datenbank abzulegen, zu bearbeiten (also neue Versionen anzulegen) und wieder zu löschen, wobei die Anwendung der letzten beiden Operationen bei regulären Nutzern auf die Menge der selbst erstellten Programme beschränkt ist. Demgegenüber ist es Administratoren per Spezifikation erlaubt, beliebige Programme zu bearbeiten und zu löschen.

[2] Eine fehlgeschlagene Ausführung kann sowohl von Syntaxfehlern (also Fehlern, die vom Compiler erkannt werden) verursacht werden, als auch von Laufzeitfehlern.

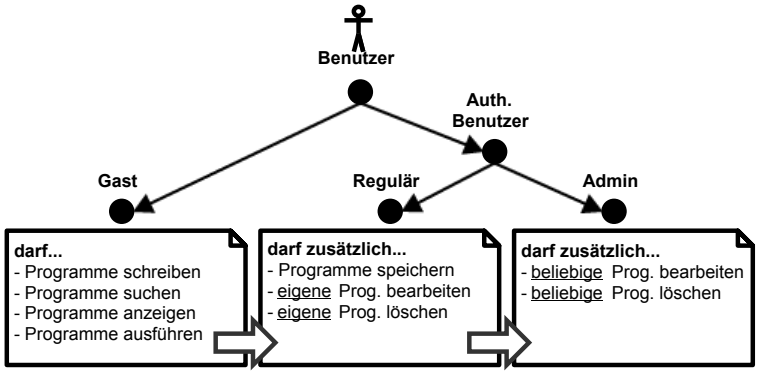


Abbildung 3.1.: Spezifikation des Rollensystems

Ferner sind Administratoren für die Verwaltung der Anwendung zuständig. Dies schließt insbesondere die Verwaltung der unterstützten Programmiersprachen und Ausführungssysteme ein. Nur Administratoren können diesen Funktionsumfang durch das Hinzufügen neuer Programmiersprachen und Ausführungssysteme erweitern.

3.1.2. Nichtfunktionale Anforderungen

Neben der Realisierung der funktionalen Anforderungen sollen für die Umsetzung der zu entwickelnden Anwendung folgende Rahmenbedingungen berücksichtigt werden:

▷ **Layout, Handhabung und Benutzbarkeit**

Gemessen am Funktionsumfang sollte die zu entwickelnde Anwendung ein möglichst simples, strukturiertes und bedienerfreundliches Layout besitzen. Beim Entwurf und der Entwicklung der Präsentationsschicht sollten deshalb die folgenden Punkte besonders beachtet werden:

- Verwendung eines übersichtlichen (statischen) Rahmenlayouts für die primäre Navigation.
- Verwendung ähnlicher Basislayouts für die Bestandteile einer funktionalen Komponente.
- Pflegen einer möglichst schmalen Präsentationsschicht zur Vermeidung von unnötigem Navigationsaufwand.
- Gezielte Bereitstellung von Hilfeleistung in Form von Hinweisen, Hilfstexten und Tooltips zur Förderung der intuitiven Bedienbarkeit.

▷ **Skalierbarkeit, Flexibilität des Datenmodells**

Um Änderungen des Problemumfangs (insbesonderes Erweiterungen der Funktionalitäten) möglichst problemlos bewältigen zu können, sollte die Anwendung umempfindlich gegenüber Modifikationen des Datenmodells sein.

▷ **Entwicklung in Curry**

Wie einleitend in Abschnitt 1.2 erläutert, soll die Anwendung als Webanwendung unter Verwendung der logisch-funktionalen Programmiersprache Curry (siehe Abschnitt 2.1) und des in Curry implementierten Webframeworks Spicey (siehe Abschnitt 2.2) realisiert werden.

3.1.3. Tabellarische Zusammenfassung

Identifikator	Kurzbeschreibung
	Programme als zentrale Objekte
F1.1	Verknüpfung mit Quelltext (in möglicherweise mehreren Versionen)
F1.2	Verknüpfung mit Metadaten (Titel, Beschreibung, Sprache, Autor, ...)
F1.3	Verknüpfung mit Tags
	Erstellung von Programmen
F2.1	Erstellung nur mit unterstützten Programmiersprachen
F2.2	Eingabe von Quelltext/Programmen in geeigneter Weboberfläche
F2.3	Speichern von Programmen
	Bearbeitung von Programmen
F3.1	Bearbeitung von Programmen in geeigneter Weboberfläche
F3.2	Speichern neuer Versionen zu Programmen
	Verwaltung von Programmen
F4.1	Erweiterte Suchfunktion
F4.2	Listenansicht für Programme (z. B. für die Resultate von Suchen)
F4.3	Einzelansicht für Programme (mit Zugriff auf alle assoziierten Daten)
	Ausführung von Programmen
F5.1	Ausführung nur mit unterstützten Ausführungssystemen
F5.2	Ausführung während der Erstellung/Bearbeitung (mit Ergebnisklassifikation)
F5.3	Ausführung bereits gespeicherter Programme (mit Ergebnisklassifikation)
	Löschen von Programmen
F6.1	Löschen von Programmen (mit Bestätigung der Aktion durch den Nutzer)
	Registrierung und Authentifizierung
F7.1	Registrierung von Benutzern
F7.2	Authentifizierung von Benutzern
	Rollensystem und Autorisierung
F8.1	Implementierung des Rollensystems (nach geg. Spezifikation)
NF1	Simples, bedienerfreundliches Layout/Schmale Präsentationsschicht
NF2	Skalierbarkeit, Flexibilität des Datenmodells
NF3	Entwicklung in Curry

Abbildung 3.2.: Zusammenfassung der Anforderungen

3.2. Identifikation der funktionalen Komponenten

Die Ergebnisse der Anforderungsanalyse aus dem letzten Abschnitt zeigen, dass die zu entwickelnde Anwendung hauptsächlich grundlegende Operationen für den Zugriff und die Verarbeitung von Programm-Datensätzen bereitstellen soll. Diese (manchmal auch mit dem Akronym *CRUD* [Mar83] bezeichneten) Operationen umfassen:

(1) Die Erstellung und Speicherung von Programmen (**c**reate).

(2) Das Suchen und Anzeigen von in der Datenbank abgelegten Programmen (**r**ead).

(3) Das Bearbeiten von Programmen (**u**pdate).

(4) Das Löschen von Programmen (**d**elete).

Demgegenüber steht als weitere zentrale Anforderung die Ausführung von Programmen. Obwohl die Ausführung von Programmen zunächst keiner CRUD-Operation zugeordnet werden kann, lässt sie sich praktikabel mit den Punkten **(1)** und **(3)** assoziieren. Einerseits wird die Möglichkeit, Programme während der Erstellung und Bearbeitung ausführen zu können, exakt in **F5.2** (siehe Tabelle 3.2) gefordert und andererseits ist die in Anforderung **F5.3** geforderte Funktionalität zur Ausführung bereits gespeicherter Programme sinnvoll mit der Funktionalität zur Bearbeitung von Programmen (bzw. deren Quelltexten) kombinierbar, indem Programme aus der Datenbank zur Ausführung in die gleiche Umgebung geladen werden, in der auch die Bearbeitung möglich ist. Dies hat zwei Vorteile:

(1) Erstens haben Nutzer so direkt die Möglichkeit den Quelltext von Programmen anderer Nutzer zu Testzwecken (bzw. zum „Ausprobieren“) beliebig zu modifizieren, da die entsprechende Weboberfläche die Bearbeitung von Quelltext sowieso nativ unterstützt.

(2) Zweitens sind für den Nutzer auf diese Weise für die Ausführung und Bearbeitung von Programmen dieselben Schritte zu unternehmen. Die Anwendung unterscheidet zunächst nicht, ob ein Benutzer ein Programm nur zur Ausführung in die entsprechende Weboberfläche lädt oder ob er den Quelltext bearbeiten und als neue Version abspeichern will (und möglicherweise weiß der Nutzer das auch selbst noch nicht). Stattdessen wird von der Anwendung nur geprüft, ob der Nutzer die Berechtigung *hätte*, eine neue Version zu dem geladenen Programm anzulegen und gegebenfalls eine entsprechende Option auf der Weboberfläche eingeblendet.

Die Erstellung, Bearbeitung und Ausführung von Programmen lässt sich zweckmäßig also in eine einzige funktionale Komponente integrieren, die für diese Funktionen zunächst auch nur eine zentrale Weboberfläche zur Verfügung stellen muss, da sich Erstellung und Bearbeitung von Programmen bezüglich der Benutzerschnittstelle kaum unterscheiden (präsentationstechnisch unterscheiden sich beide Fälle beinahe nur durch den initialen Inhalt des Quelltext-Eingabebereichs der entsprechenden Weboberfläche). Die auf diese Weise definierte Komponente ist zudem *exklusiv* für die Eingabe und Ausführung von Programmcode zuständig und diesbezüglich unabhängig von anderen funktionalen Komponenten der Anwendung, was einerseits dem Prinzip der Modularisierung entspricht und andererseits der Wartbarkeit, Änderbarkeit und Skalierbarkeit der Anwendung zuträglich ist.

Damit ein Nutzer ein in der Datenbank abgelegtes Programm überhaupt bearbeiten oder ausführen kann, muss es ihm von der Anwendung zugänglich gemacht werden. Für diesen Zweck sieht die Spezifikation eine Suchfunktion und entsprechende Listen- und Einzelansichten für die in der Datenbank abgelegten Programme vor (siehe Punkt **(2)**). Diese Funktionalitäten werden in einer zweiten funktionalen Komponente zusammengefasst, die die Suchfunktion implementiert und die entsprechende Weboberfläche für die Listenansicht rendert, über die Nutzer zu den Einzelansichten navigieren können. Zusätzlich ist die Komponente auch für die Umsetzung und Integration der Löschfunktion zuständig (siehe Punkt **(4)**).

Die Motivation für die Aufteilung der CRUD-Operationen auf diese beiden funktionalen Komponenten ist die Idee, die Arbeitsabläufe bei der Benutzung der zu entwickelnden Webanwendung (Erstellung und Bearbeitung von Programmen, Programmausführung, Programmverwaltung, etc.) möglichst analog zu den entsprechenden Arbeitsabläufen zu gestalten, die Nutzer typischerweise von ihrem lokalen Betriebssystem kennen. Dabei repräsentiert die funktionale Komponente zur Erstellung, Bearbeitung und Ausführung von Programmen einen *Quelltext-Editor mit integrierter Shell* (lokal üblicherweise als integrierte Entwicklungsumgebung realisiert; siehe Abschnitt 1.1), mit dem man Programme erzeugen, öffnen und mittels integrierter Shell zeitgleich ausführen kann. Gespeicherte Programme werden auf der *Festplatte* (repräsentiert durch die Datenbank) persistiert und können über die grafische Benutzerschnittstelle eines *Dateimanagers* (repräsentiert durch die zweite funktionale Komponente) verwaltet werden. Ein solcher Dateimanager arbeitet dabei üblicherweise mit den Metadaten der zu verwaltenden Inhalte und erlaubt das Suchen, Auflisten und Löschen der Inhalte.
Aufgrund dieser Analogie wird die erste funktionale Komponente der zu entwickelnden Anwendung von nun an als **interaktiver Editor** (oder kurz **SmapIE**) bezeichnet und die zweite funktionale Komponente als **Browser** (in Anlehnung an den englischen Begriff *file browser*). Diese Bezeichnungen werden dabei sowohl auf Implementationsebene als auch in den Weboberflächen verwendet.

Der interaktive Editor und der Browser repräsentieren die zentralen funktionalen Komponenten der Software. Zusätzlich wird die Anwendung noch zweckmäßig um die drei folgenden Komponenten ergänzt:

(1) Eine erste Komponente, die alle der **Authentifizierung** und Benutzerverwaltung zugeordneten Funktionalitäten implementiert.

(2) Eine zweite Komponente, die Funktionen zur **Administration** der Anwendung implementiert.

(3) Eine dritte Komponente, die das Rollensystem implementiert und alle Funktionalitäten verwaltet, die zur **Autorisierung** gehören.

Diese Komponenten ergeben sich direkt aus den entsprechenden Anforderungen des letzten Abschnitts (siehe **`F7.x`** und **`F8.x`**). Da die Spezifikation zunächst keine speziellen Funktionalitäten für die Verwaltung von Benutzern und Benutzerkonten (z. B. benutzerspezifische Einstellungen, Bearbeitung der Registrierungsdaten o. ä.) vorsieht, wird die Erstellung neuer Nutzer mittels Registrierung in die Authentifizierungs-Komponente integriert. Die Aufteilung von Administration und Autorisierung in zwei unabhängige Komponenten

ist ebenfalls sinnvoll, da Zugriffskontrolle in *allen* funktionalen Komponenten benötigt wird, die Interaktion mit dem Benutzer erlauben, während die Administration von diesen Komponenten unabhängig ist (siehe nächster Abschnitt).

3.3. Aufbau und Funktionsweise der Webanwendung

Abbildung 3.3 zeigt den groben konzeptionellen Aufbau der zu entwickelnden Webanwendung und gibt zusätzlich zu jeder Komponente die von ihr implementierten zentralen funktionalen Anforderungen aus Tabelle 3.2 an.

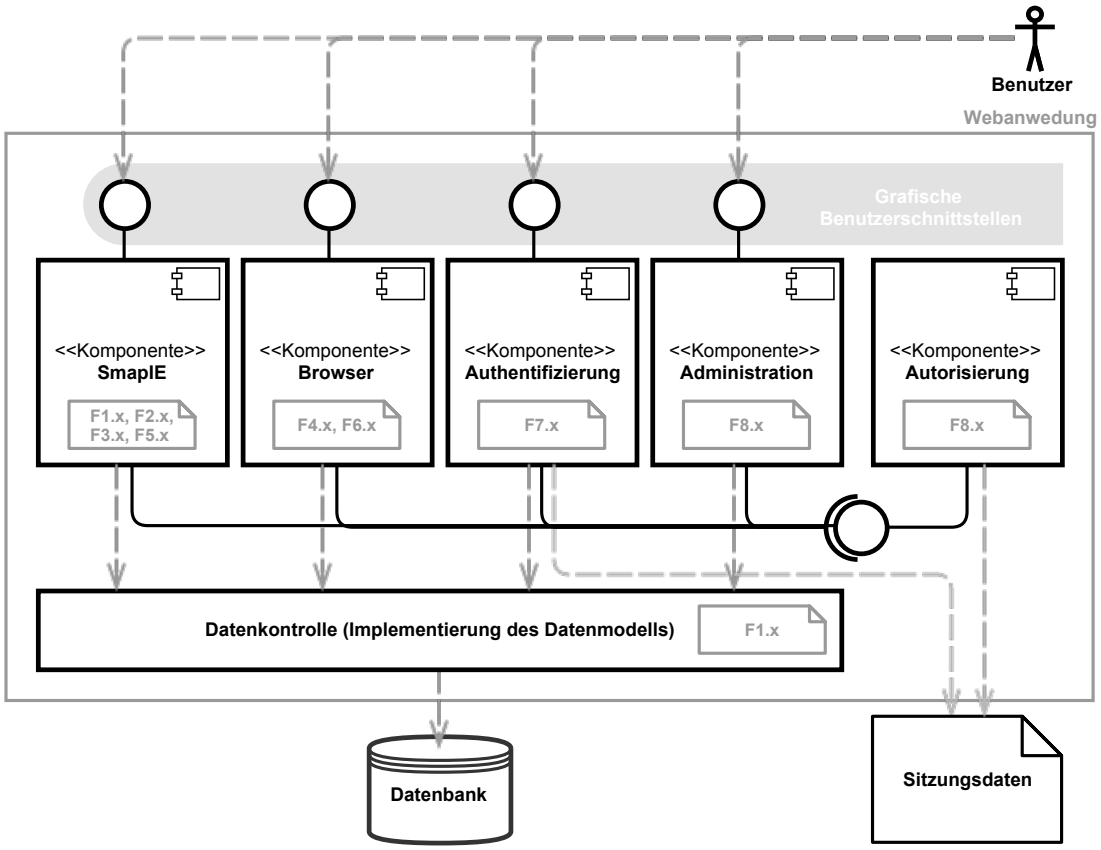


Abbildung 3.3.: Konzeptioneller Aufbau der Webanwendung

Das Front-End der Anwendung wird im Wesentlichen von den durch die Komponenten bereitgestellten grafischen Benutzeroberflächen (Weboberflächen, Views) gebildet. Auf diese kann der Nutzer über den Webbrowser zugreifen. Ausgenommen davon ist die Autorisierung-Komponente. Diese stellt statt einer grafischen Schnittstelle für den Nutzer eine funktionale Schnittstelle zur Verfügung, über die die anderen Komponenten die Zugriffsrechte auf die von ihnen implementierten Operationen erfragen können. Dazu werden serverseitig Sitzungsdaten verwaltet, die bei der Authentifizierung eines Benutzers angelegt und durch die Autorisierungskomponente ausgelesen werden.
Benutzeraktionen in den Weboberflächen (bzw. HTTP-Requests) werden in der Regel

von der logischen Einheit (oder Datenverarbeitungseinheit) der jeweiligen Komponente verarbeitet. Diese greift über eine (im Moment noch nicht näher spezifizierte) Datenkontrollschicht, die unter anderem die Implementierung des noch zu ermittelnden Datenmodells beinhaltet, auf die Datenbank zu und rendert als Antwort eine Weboberfläche, die wiederum (per HTTP-Response) an den Webbrowser des Benutzers geschickt wird.

In den folgenden Unterabschnitten werden nun die Entwürfe zum Aufbau und zur Funktionsweise der Komponenten vorgestellt, mit denen der Nutzer direkt interagiert. Dabei wird teilweise auf Layout-Skizzen zurückgegriffen, die die wichtigsten Elemente der Weboberflächen hervorheben sollen.

3.3.1. SmapIE

Abbildung 3.4 zeigt eine Skizze der zentralen Arbeitsoberfläche des interaktiven Editors *SmapIE* (im folgenden kürzer als *Editor* bezeichnet) im Webbrowser.

Abbildung 3.4.: Rahmenlayout und interaktiver Editor (Entwurf)

Die Navigationsleiste (A) ist das primäre Navigationselement der Anwendung und als Bestandteil des Rahmenlayouts in jeder Weboberfläche der Anwendung enthalten. Zusammen mit der Fußleiste begrenzt die Navigationsleiste den Teil des HTML-Dokuments, der den eigentlichen Inhalt der momentan angezeigten Webseite repräsentiert.

▷ **Erstellung von Programmen**

Um ein neues Programm zu erstellen, wählt der Nutzer von einem beliebigen Standpunkt innerhalb der Anwendung über das Dropdown-Menü Ⓑ die gewünschte Sprache. Daraufhin wird die in Abbildung 3.4 dargestellte Weboberfläche als neue „Instanz“ des Editors geöffnet und mit einer entsprechenden *Ausführungsumgebung* (einem Paar bestehend aus der Sprache und der Liste zugehöriger Ausführungssysteme) verknüpft. Über das Eingabefeld Ⓒ kann nun der Quelltext eingegeben werden.

▷ **Ausführen von Programmen**

Die Ausführung des im Eingabefeld Ⓒ befindlichen Quelltextes durch einen externen Prozess kann über den Button Ⓔ veranlasst werden. Die tatsächliche Implementierung der Ausführung bleibt dem Nutzer verborgen und ist unter Berücksichtigung der Schnittstellenvereinbarung zwischen der Datenverarbeitungseinheit der SmapIE-Komponente und der Ausführungseinheit (ein Curry-Modul) theoretisch austauschbar. Eine flexible Lösung stellt beispielsweise das Senden des Quelltextes per HTTP-Post-Request an ein CGI-Skript dar, welches unter einer gegebenen URL erreichbar ist und das Ergebnis der Ausführung als Antwort an die Ausführungseinheit zurückschickt (dies entspricht auch der tatsächlichen Implementierung der Ausführung und wird in Abschnitt 4.3.2 beschrieben). Steht das Ausführungsergebnis zur Verfügung, wird es dem SmapIE-Controller übergeben, welcher es beim Neuladen der Weboberfläche in Textfeld Ⓓ lädt. Die Ausführung wird dabei mit demjenigen Ausführungssystem durchgeführt, welches vom Nutzer über das Select-Menü in der rechten oberen Ecke von Textfeld Ⓓ ausgewählt wurde.

▷ **Speichern von Programmen**

Button Ⓕ öffnet ein Dropdown-Menü über das ein zuvor authentifizierter Nutzer den Quelltext als neues Programm speichern kann (für Gäste wird dieser Menüpunkt nicht angezeigt). Dazu wird aus Sicherheitsgründen (z. B. zum Schutz vor Spambots) zunächst geprüft, ob der zu speichernde Quelltext mit dem aktuell ausgewählten Ausführungssystem erfolgreich ausführbar ist. Ist dies nicht der Fall, wird der Nutzer über ein Dialogfenster informiert und erhält die Chance, den fehlerhaften Code weiter zu bearbeiten. Kann der Quelltext erfolgreich ausgeführt werden, wird der Nutzer zu einem WUI-Formular (siehe Abschnitt 2.1.2) weitergeleitet, welches die Eingabe eines Titels, einer Beschreibung, einer Sichtbarkeit und einer Liste von klassifizierenden Tags erwartet. Die Sichtbarkeit ist dabei eine Erweiterung der zentralen Anforderungen an Programme um die Eigenschaft, nur für den Autor des Programms (*privat*) sichtbar zu sein oder für alle Nutzer der Anwendung (*öffentlich*). Anschließend wird das Programm in der Datenbank abgelegt.

▷ **Bearbeitung und Ausführung von gespeicherten Programmen**

Die Bearbeitung und das Ausführen von bereits gespeicherten Programmen verläuft analog zur Erstellung von Programmen, mit dem Unterschied, dass die Weboberfläche über einen

Hyperlink aufgerufen wird, dessen Ziel-URL die ID des Programms enthält. Falls ein Programm mit der entsprechenden ID in der Datenbank existiert, wird das Eingabefeld © initial mit der aktuellsten Version des Quelltextes gefüllt, welche dann bearbeitet und ausgeführt werden kann. Statt einer Option zum Speichern eines neuen Programms zeigt das Dropdown-Menü von Button Ⓕ in diesem Fall eine Option zum Anlegen einer neuen Version an, falls der Nutzer dazu berechtigt ist (falls er also der ursprüngliche Autor oder ein Administrator ist). Auch hier gilt: Der Quelltext darf nur als neue Version des Programms gespeichert werden, wenn er erfolgreich ausführbar ist. Zum Abschluss des Speichervorgangs ist noch die Angabe einer Versionsnachricht notwendig, in der der Nutzer die vorgenommenen Änderungen am Quelltext kurz beschreiben kann.

3.3.2. Browser

Die *Browser*-Komponente umfasst mehrere Weboberflächen:

(1) Das *Dashboard* ist eine Art Startseite für den Browser, welche einen schnellen Überblick über das momentane Angebot an Programmen bietet und beispielsweise als Startpunkt für die (erweiterte) Programmsuche verwendet werden kann. Es dient vor allem als benutzerfreundliche Navigationshilfe und stellt daher keine exklusiven Funktionalitäten zur Verfügung.

(2) Die *Listenansicht* für Programme ist die zentrale Weboberfläche für die Darstellung von Suchergebnissen jeglicher Art und der Ausgangspunkt für die Verwaltung der in der Datenbank abgelegten Programme.

(3) Die *Einzelansicht* enthält alle Informationen zu einem gespeicherten Programm (insbesondere die Metadaten) und ermöglicht zudem bestimmte Interaktionen mit dem Programm. Von der Einzelansicht aus kann ein Programm beispielsweise zur Bearbeitung und Ausführung in den interaktiven Editor geladen oder gelöscht werden.

(4) Die *Tag-Übersicht* listet lexikographisch geordnet alle jemals von Nutzern verwendeten Tags auf. Dies kann z. B. bei der Erstellung eines neuen Programms oder der Auswahl eines Suchbegriffs für die Programmsuche hilfreich sein.

Im Kontext der zentralen Anforderungen an die zu entwickelnde Anwendung sind vor allem die Listenansicht und die Einzelansicht für Programme bedeutend (siehe **F4.2** und **F4.3** in Tabelle 3.2). Diese sollen daher nachfolgend näher beschrieben werden.

▷ Suchen und Auflisten von Programmen

Abbildung 3.5 zeigt den Entwurf für die Listenansicht von Programmen als zentrale Weboberfläche für die Darstellung von Suchergebnissen. Die Listenansicht folgt im Regelfall auf die direkte Verwendung der Suchfunktion und erhält deren benannte Parameter über den Query String der momentan zu verarbeitenden URL. Dies hat den Vorteil, dass die Suche theoretisch auch über die URL gesteuert werden kann und Suchergebnisse zwischen Nutzern austauschbar sind.

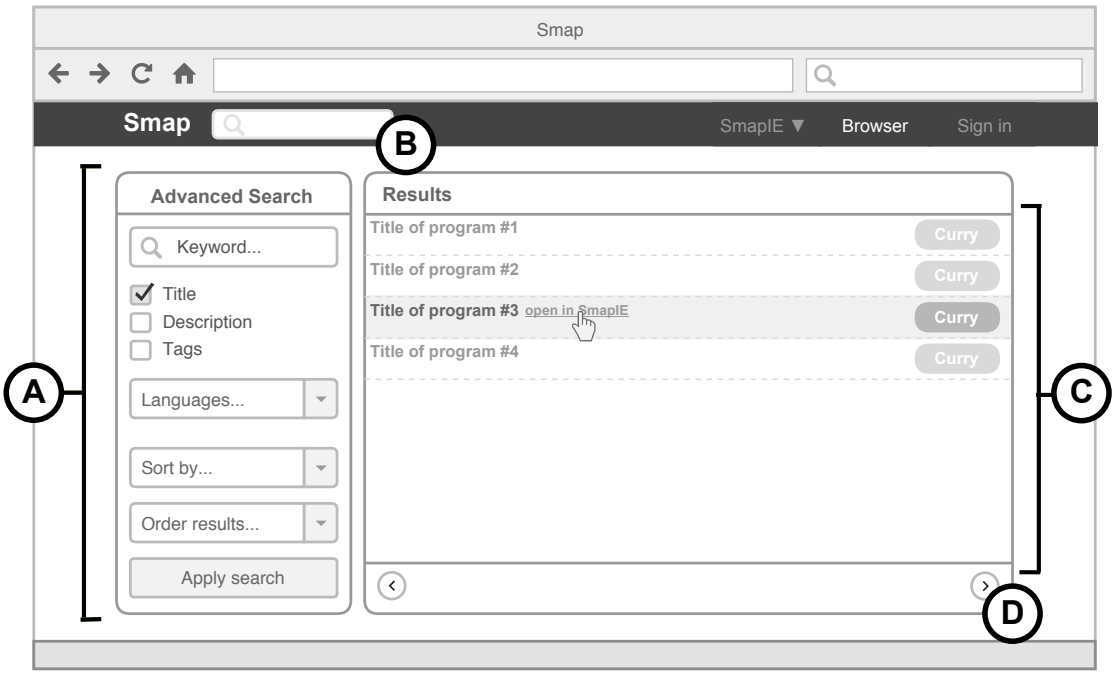


Abbildung 3.5.: Listenansicht im Browser (Entwurf)

Neben der direkten Eingabe der Suchparameter über die URL oder dem Folgen eines entsprechenden Hyperlinks gibt es für den Nutzer noch zwei weitere Möglichkeiten eine allgemeine Suche zu initiieren:

(1) Die erste Möglichkeit ist die Verwendung des Formulars (A) für die erweiterte Suche (im Folgenden als *Search Panel* bezeichnet), welches sowohl auf der Weboberfläche für das Dashboard als auch für die Listenansicht in der Seitenleiste zu finden ist. Hier lassen sich (von oben nach unten) folgende Einstellungen vornehmen:

Filtern nach Schlüsselwort Filtert alle Programme, bei denen das gegebene Schlüsselwort in mindestens einem von möglicherweise mehreren definierten Attributen enthalten ist. Die Auswahl der zu untersuchenden Attribute wird über die Checkboxen unterhalb des Eingabefeldes vorgenommen. Hierbei stehen der Titel, die Beschreibung und die Liste aller Tags eines Programms zur Auswahl, wobei bei letzterem geprüft wird, ob mindestens ein Tag in der Liste enthalten ist, das zu dem Schlüsselwort identisch ist.

Filtern nach Programmiersprache Filtert alle Programme, deren Quelltext in der über das Select-Menü ausgewählten Programmiersprache verfasst wurde.

Sortierung der Ergebnisse Sortiert die Ergebnisliste mit der über das Select-Menü ausgewählten Sortierung (z. B. nach Erstellungsdatum oder Titel).

Anordnung der Ergebnisse Legt die Ordnung der Ergebnisliste bezüglich der gewählten Sortierung fest (aufsteigend oder absteigend).

(2) Die zweite Möglichkeit ist die Verwendung des Eingabefeldes Ⓑ, welches auch als „*Quick Search*“-Feld bezeichnet wird, weil es über die Navigationsleiste jederzeit zur Durchführung einer „schnellen“ Suche zur Verfügung steht. Die Verwendung der Quick Search ist äquivalent zur der Suche via Search Panel, wobei das gegebene Schlüsselwort im Titel, in der Beschreibung und in den Tags gesucht wird. Für die Sortierung und Anordung werden in diesem Fall zweckmäßig Standardwerte definiert.

Die Suchparameter im Query String werden von der Datenverarbeitungseinheit der Browser-Komponente extrahiert und in eine Anfrage für die Datenkontrollschicht umgewandelt. Das Ergebnis dieser Anfrage wird letztendlich als Inhalt des Feldes Ⓒ in der Listenansicht dargestellt. Dazu kann die Ergebnisliste ab einer gewissen Länge auf mehrere Seiten verteilt werden. Das Durchblättern dieser Seiten erfolgt über die Navigationselemente in der Fußleiste Ⓓ.
Von der Listenansicht kann nun einerseits über den Titel des Programms zur Einzelansicht navigiert werden, andererseits können Programme bereits von hier zur Bearbeitung und Ausführung in den interaktiven Editor geladen werden. Dazu erscheint bei der Auswahl eines Listenelements ein entsprechender Hyperlink.

Aus der Einführung der Sichtbarkeitseigenschaft in Abschnitt 3.3.1 folgt auch die Notwendigkeit für eine Behandlung der Sichtbarkeit bei Programm-Suchen. Deshalb wird für allgemeine Suchvorgänge festgelegt, dass nur sichtbare (öffentliche) Programme in der Ergebnisliste enthalten sein können. Zusätzlich erhalten authentifizierte Nutzer einen autorisierten Zugriff auf eine Listenansicht ihrer selbst erstellten Programme (inklusive der privaten Programme), die sinnvollerweise ebenfalls von der Browser-Komponente verwaltet wird. Selbiges gilt auch für die Favoritenliste eines Nutzers (siehe nächster Abschnitt).

▷ Anzeigen einzelner Programme

Abbildung 3.6 zeigt den Entwurf für die Einzelansicht für Programme im Browser. Auf der Einzelansicht werden einerseits alle zu einem Programm bekannten Informationen zusammengetragen und dargestellt, andererseits erlaubt sie dem Nutzer Interaktionen abseits der Bearbeitung und Ausführung des Quelltextes.
Alle dem Nutzer verfügbaren Optionen werden dazu im Panel Ⓐ in der Seitenleiste zusammengefasst. Von hier aus kann der Nutzer den Quelltext der aktuellesten Version des Programms in den interaktiven Editor laden oder das Programm löschen (falls er dazu befugt ist). Wie in Abbildung 3.6 zu sehen ist, können Programme über das Optionen-Panel außerdem zu der *Favoritenliste* eines Nutzers hinzugefügt (und auch wieder entfernt) werden. Die Favorisierung von Programmen durch Nutzer stellt eine Erweiterung der in Abschnitt 3.1.1 beschriebenden zentralen funktionalen Anforderungen an die zu entwickelnde Anwendung dar und dient folgenden Zwecken:

(1) Programme, die vom Nutzer als besonders hilfreich erachtet oder häufig verwendet werden, können der Favoritenliste hinzugefügt werden. Auf diese Favoritenliste erhält der Nutzer (ähnlich wie bei der Liste der selbst erstellten Programme) einen autorisierten Zugriff, was die Navigation zu den entsprechenden Programmen erleichtert.

(2) Die Favorisierung von Programmen erlaubt eine weitere (für Nutzer möglicherweise interessante) Klassifizierung von Programmen: die Beliebtheit. Es ist dabei anzunehmen, dass häufig favorisierte Programme eine hohe Qualität aufweisen und somit auch für andere Nutzer interessant sind. Resultate von Suchvorgängen können deshalb auch nach Beliebtheit (also der Anzahl an Nutzern, die ein Programm ihrer Favoritenliste hinzugefügt haben) sortiert werden.

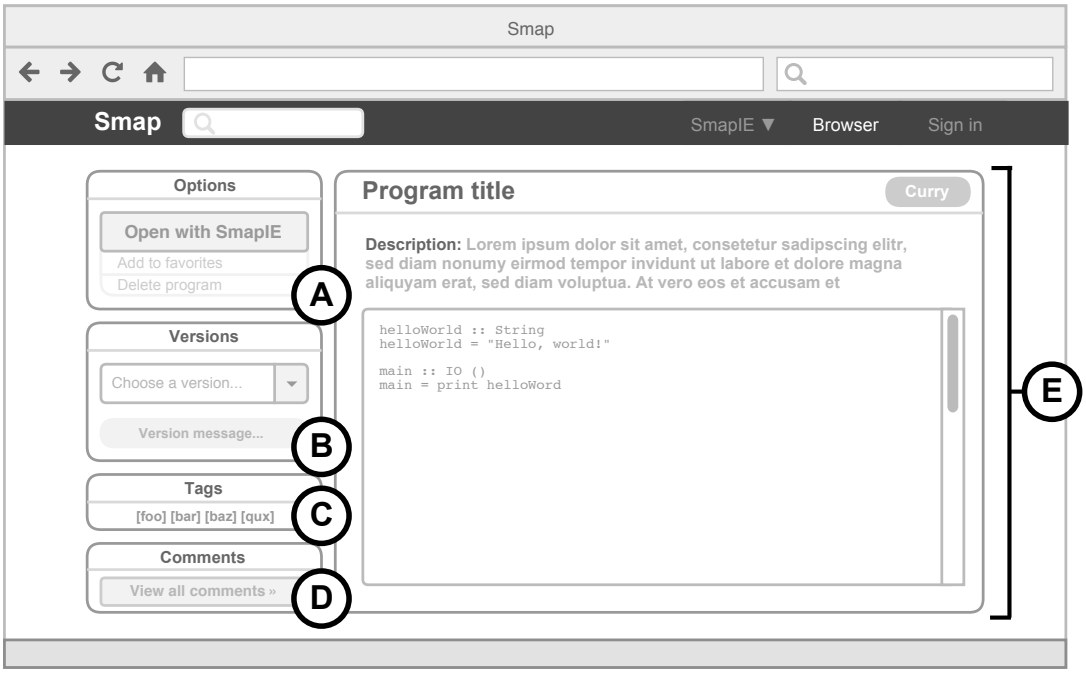


Abbildung 3.6.: Einzelansicht im Browser (Entwurf)

Über Panel (B) erhält der Nutzer Zugriff auf die Versionshistorie des entsprechenden Programms. Über das Select-Menü kann dazu eine Version ausgewählt werden, deren Quelltext dann im Haupt-Panel (E) dargestellt wird. Die entsprechende Versionsnachricht wird unterhalb des Menüs angezeigt.
Eine Liste aller mit dem Programm verknüpften Tags ist in Panel (C) zu finden. Über den Namen eines hier aufgeführten Tags kann eine Suche gestartet werden, die alle Programme liefert, die ebenfalls mit dem entsprechenden Tag verknüpft sind. Da die allgemeine Suchfunktion über den Query String der URL gesteuert werden kann (siehe letzter Abschnitt), muss jeder Name dabei nur einen Hyperlink repräsentieren.
Mittels Panel (D) können die *Kommentare* zu einem Programm abgerufen werden. Kommentare sind textuelle Nachrichten, über die (registrierte) Nutzer ihre Meinung, Verbesserungsvorschläge, Hinweise o. ä. zu einem in der Datenbank abgelegten Programm abgeben können. Wie die Favorisierung von Programmen ist die Kommentarfunktion eine Erweiterung der ursprünglichen funktionalen Anforderungen, die vor allem praktischen Nutzen hat.
Die „primären“ Metadaten (Titel, Beschreibung, Programmiersprache, ...) zu dem betrachteten Programm, sowie der Quelltext der momentan ausgewählten Version werden im

Haupt-Panel (E) zusammengefasst. Hierbei ist der Inhalt des Textfeldes, welches den Quelltext enthält, nicht zur Bearbeitung freigeschaltet. Wie in Abschnitt 3.2 beschrieben, ist die Eingabe und Bearbeitung von Quelltext *ausschließlich* im Editor erlaubt. Die Einzelansicht des Browsers bietet lediglich eine Vorschau auf den bearbeitbaren und ausführbaren Code eines Programms und erlaubt zudem das Durchblättern älterer Versionen.

3.3.3. Authentifizierung

Die Registrierung eines Nutzers erfolgt über ein simples WUI-Formular unter Angabe eines Benutzernamens, einer E-Mail-Adresse und eines Passworts. Nach erfolgreicher Registrierung (falls sowohl Benutzername als auch E-Mail-Adresse nicht schon mit einem anderen Benutzerkonto verknüpft sind) kann sich der entsprechende Nutzer in der dafür vorgesehenen Weboberfläche mithilfe seines Benutzernamens und seines Passworts authentifizieren.

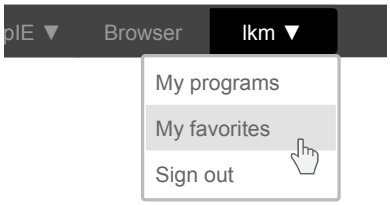


Abbildung 3.7.: Benutzermenü (Entwurf)

Nach erfolgreicher Authentifizierung (falls ein Benutzerkonto mit der gegebenen Kombination aus Benutzername und Passwort existiert) ist über die Navigationsleiste das in Abbildung 3.7 skizzierte Benutzermenü aufrufbar. Über dieses haben Nutzer Zugriff auf die Liste der selbst erstellten Programme und ihre Favoritenliste, welche, wie bereits in Abschnitt 3.3.2 beschrieben, mittels Listenansicht im Browser dargestellt werden. Auch die Abmeldung von Nutzern erfolgt über das Benutzermenü.

Falls Nutzer ihr Passwort verlieren oder vergessen sollten, kann über die Authentifizierungs-Komponente die Erzeugung eines neuen Passworts angestoßen werden, welches dann per E-Mail an den entsprechenden Nutzer gesendet wird. Diesem Zweck dient auch die Angabe der E-Mail-Adresse bei der Registrierung .

3.3.4. Administration

Die Administrations-Komponente stellt zunächst nur WUI-Formulare für das Hinzufügen neuer Programmiersprachen und Ausführungssysteme zur Verfügung. Auf diese kann der authentifizierte Nutzer mit Administrationsrechten ebenfalls über das Benutzermenü in der Navigationsleiste zugreifen.

Alle weiteren Privilegien von Administratoren werden von der Autorisierungs-Komponente verwaltet. Diese definiert für autorisierte Operationen entsprechende *Zugriffstypen*, mithilfe derer geprüft wird, ob ein Nutzer die erforderlichen Rechte zur Durchführung einer Aktion hat. Das Ergebnis einer solchen Prüfung kann dann beispielsweise verwendet werden, um bestimmte Elemente (z. B. die Option zum Löschen eines Programms) in Weboberflächen ein- oder auszublenden.

Die konkrete Funktionsweise der Autorisierungs-Komponente wird im Rahmen der Implementierung in Abschnitt 4.3.1 beschrieben. Selbiges gilt für die bisher nicht näher

spezifizierte Datenkontrollschicht, deren Implementierung auf dem im nächsten Abschnitt beschriebenen Datenmodell basiert.

3.4. Herleitung des Datenmodells als ER-Modell

Die grundlegenden Entitätstypen, ihre Attribute und die Beziehungen des ER-Modells lassen sich unmittelbar aus den zentralen funktionalen Anforderungen an die zu entwickelnde Anwendung (Abschnitt 3.1.1) und den Beschreibungen des Aufbaus und der Funktionsweise der einzelnen Komponenten in den Abschnitten 3.3.1 bis 3.3.4 ableiten. Abbildung 3.8 zeigt das zugehörige Entity-Relationship-Diagramm in Chen-/(min,max)-Notation [Che76, Abr76]:

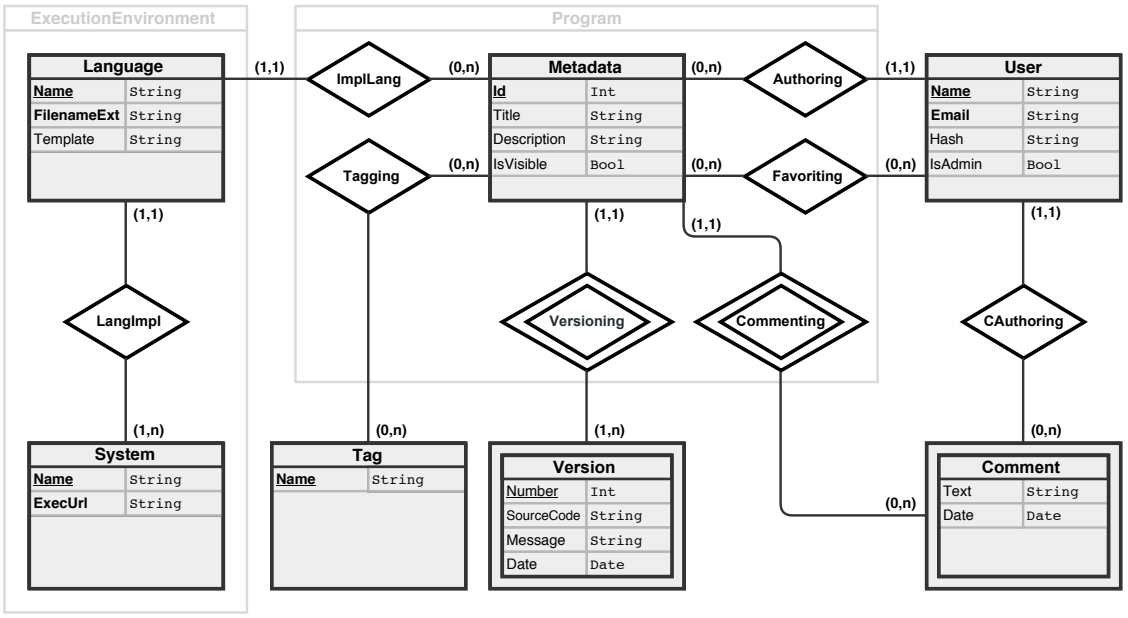


Abbildung 3.8.: Datenmodell der Webanwendung (ER-Diagramm)

▷ **Die `Metadata`-Entität**

Jedes Programm wird durch genau eine `Metadata`-Entität identifiziert, welche zudem alle atomaren Metainformationen des entsprechenden Programms repräsentiert. Dies umfasst die ID (`Id`), den Titel (`Title`), die Beschreibung (`Description`) und die Sichtbarkeit (`IsVisible`) des Programms. Alle weiteren (zusammengesetzten) Attribute eines Programms werden wiederum über die Beziehungen der identifizierenden `Metadata`-Entität realisiert. Dazu gehören:

- Die Programmiersprache, in der das Programm bzw. der Quelltext des Programms verfasst wurde (`ImplLang`-Beziehung).
- Der Autor des Programms (`Authoring`-Beziehung).

- Der Quelltext des Programms. Dieser liegt in mindestens einer Version vor, kann theoretisch aber auch in beliebig vielen Versionen vorliegen (`Versioning`-Beziehung).
- Die Tags des Programms (`Tagging`-Beziehung).
- Eine (möglicherweise leere) Menge an Kommentaren, die zu dem Programm verfasst wurden (`Commenting`-Beziehung).
- Eine (möglicherweise leere) Menge an Nutzern, zu deren Favoritenliste das Programm hinzugefügt wurde (`Favoriting`-Beziehung).

Die Zusammenfassung der `Metadata`-Entität mit all diesen Beziehungen stellt die Abbildung der Programm-Spezifikation in das ER-Modell dar. Dies wird durch den mit `Program` betitelten grauen Kasten verdeutlicht.

▷ Die `Version`-Entität

`Version`-Entitäten repräsentieren die Versionen des Quelltextes eines Programms. Sie sind immer genau einem Programm (bzw. der identifizierenden `Metadata`-Entität) zugeordnet und können ohne ein solches nicht existieren. Aus diesem Grund werden `Version`-Entitäten als *schwache Entitäten* mit einem *partiellen Schlüssel*, der Versionsnummer, modelliert. Zur Identifikation einer Version muss deshalb neben der Versionsnummer (`Number`) auch immer die ID der zugehörigen `Metadata`-Entität betrachtet werden. `Version`-Entitäten enthalten neben der Versionsnummer auch den eigentlichen Quelltext (`SourceCode`) des Programms (in der entsprechenden Version) sowie das Datum der Erstellung der Version (`Date`) und die zugehörige Versionsnachricht (`Message`).

▷ Die `Tag`-Entität

`Tag`-Entitäten repräsentieren die mit Programmen assoziierbaren Tags. Ein Tag ist eindeutig über seinen Namen (`Name`) identifizierbar und kann beliebig vielen Programmen (bzw. identifizierenden `Metadata`-Entitäten) zugeordnet sein.

▷ Die `Comment`-Entität

`Comment`-Entitäten repräsentieren Kommentare zu Programmen und bestehen aus dem Inhalt des Kommentars (`Text`), dem Datum der Erstellung (`Date`) und einer Verknüpfung zum Autor des Kommentars (`CAuthoring`-Beziehung). Wie `Version`-Entitäten können `Comment`-Entitäten nicht ohne das zugehörige Programm (bzw. ohne die identifizierende `Metadata`-Entität) existieren und wurden deshalb ebenfalls als schwache Entitäten modelliert.

▷ **Die `User`-Entität**

Registrierte Benutzer werden durch die `User`-Entität repräsentiert. Zu einem Nutzer gehört ein eindeutiger Name (`Name`, Primärschlüssel), eine eindeutige E-Mail-Adresse (`Email`), ein verschlüsseltes Passwort (`Hash`) und die Information darüber, ob der entsprechende Nutzer ein Administrator ist (`IsAdmin`). Jeder Nutzer kann beliebig viele Programme und Kommentare erstellen und beliebig viele Programme zu seiner Favoritenliste hinzufügen.

▷ **Die `Language`-Entität**

`Language`-Entitäten repräsentieren die von der Anwendung unterstützten Programmiersprachen. Jedes Programm ist mit genau einer Programmiersprache verknüpft und zu jeder Programmiersprache können beliebig viele Programme erstellt werden. Weiterhin sind `Language`-Entitäten Bestandteil der in Abschnitt 3.3.1 erwähnten Ausführungsumgebungen (*execution environments*). Diese bestehen aus genau einer Programmiersprache und einer Menge an zugehörigen Ausführungssystemen (`LangImpl`-Beziehung), die zur Ausführung der Programme, deren Quelltext in der entsprechenden Sprache verfasst wurde, zur Verfügung stehen. Zu jeder Sprache sollte dabei mindestens ein solches System existieren, andernfalls kann Quellcode der entsprechenden Sprache nicht ausgeführt (und folglich auch nicht gespeichert) werden.
Zu jeder von der Anwendung unterstützten Programmiersprache wird ein eindeutiger Name (`Name`, Primärschlüssel), eine eindeutige Dateiendung[3] (`FilenameExt`, z. B. `".curry"`) und eine Quelltext-Schablone (`Template`) gespeichert. Die Quelltext-Schablone wird dem Nutzer nach der Auswahl einer Programmiersprache zur Erstellung eines neuen Programms als Hilfestellung im Quelltext-Eingabefeld angezeigt (siehe Abbildung 3.4, ©) und sollte verdeutlichen welche grundsätzlichen Bausteine zur Ausführung des Programms vorhanden sein müssen. Dabei kann es sich z. B. um das Gerüst einer `main`-Funktion handeln.

▷ **Die `System`-Entität**

Die Ausführungssysteme einer Programmiersprache werden von der `System`-Entität repräsentiert. Zu jedem Ausführungssystem existiert genau eine Programmiersprache, ein eindeutiger Name (`Name`, Primärschlüssel) und eine eindeutige URL (`ExecUrl`) unter der das Skript liegt, welches die eigentliche Ausführung des Quelltextes übernimmt (siehe Abschnitt 3.3.1).

[3] Die Dateiendung wurde dem Modell in Hinblick auf zukünftige Erweiterungen (z. B. eine Download-Funktion für Quelltext) hinzugefügt. In der aktuellen Implementierung wird sie jedoch noch nicht verwendet.

4. Implementierung der Webanwendung

Dieses Kapitel befasst sich mit der Implementierung der Webanwendung Smap und der Umsetzung des Entwurfs. Nach der Umsetzung des in Abschnitt 3.4 ermittelten Entity-Relationship-Modells mit dem Spicey-Framework erfolgt zunächst die Abbildung des konzeptionellen Aufbaus der Webanwendung (siehe Abschnitt 3.3) in das MVC-Architekturmuster. Im Rahmen dieses Abschnitts werden die einzelnen Schichten, ihre horizontalen und vertikalen Abhängigkeiten, Funktionen und ihre grundlegende Implementierung beschrieben. Auf Basis dieser Beschreibung wird im letzten Abschnitt schließlich die Implementierung der funktionalen Komponenten (siehe 3.2 und 3.3) und ihrer Kernfunktionalitäten erläutert.

4.1. Umsetzung des ER-Modells mit dem Spicey Framework

In Abschnitt 2.2.1 wurde die Generierung eines Spicey-Projekts aus einem gegebenen ER-Modell bereits anhand eines minimalen Beispiels demonstriert. Dazu wurde in einem ersten Schritt das zugehörige ER-Diagramm in einen `ERD`-Term übersetzt, aus dem dann im nächsten Schritt mittels Scaffolding eine initiale ausführbare Anwendung generiert wurde, mit der über den Webbrowser auf die zugrunde liegenden Daten zugegriffen werden konnte.
In diesem Abschnitt wird nun zunächst die Umwandlung des in Abschnitt 3.4 ermittelten ER-Diagramms in einen entsprechenden `ERD`-Term beschrieben. Anschließend werden kurz die zentralen Datentypen der Anwendung und die wichtigsten Operationen der durch das sogenannte `erd2curry`-Tool [Han14, BHM08] während des Scaffolding-Prozesses erzeugten Datenbankschnittstelle erläutert.

4.1.1. Darstellung des ER-Diagramms als `ERD`-Term

Ähnlich wie im Beispiel aus Abschnitt 2.2.1 lässt sich das im Systementwurf ermittelte ER-Diagramm in Abbildung 3.8 nahezu direkt in einen entsprechenden `ERD`-Term übersetzen. Dazu wird zunächst jeder Entitätstyp mitsamt der Attribute auf einen entsprechenden `Entity`-Wert abgebildet und für jede Beziehung ein entsprechender `Relationship`-Wert definiert. Listing 4.1 zeigt beispielhaft den Ausschnitt des resultierenden `ERD`-Terms, der die `CAuthoring`-Beziehung und die zugehörigen Entitätstypen spezifiziert:

```
(...)

,Entity "User"
  [Attribute "Name"        (StringDom Nothing     ) PKey   False
  ,Attribute "Email"       (StringDom Nothing     ) Unique False
  ,Attribute "Hash"        (StringDom Nothing     ) NoKey  False
  ,Attribute "IsAdmin"     (BoolDom   (Just False)) NoKey  False]

,Entity "Comment"
  [Attribute "Text"        (StringDom Nothing     ) NoKey  False
  ,Attribute "Date"        (DateDom   Nothing     ) NoKey  False]

(...)

,Relationship "CAuthoring"
  [REnd "User"     "hasCAuthor"      (Exactly 1)
  ,REnd "Comment"  "isTheCAuthorOf"  (Between 0 Infinite)]

(...)
```

Listing 4.1: `Smap.erdterm` (`CAuthoring`-Beziehung und Entitätstypen)

Die im ER-Diagramm spezifizierten Attribut-Eigenschaften (Primärschlüssel-Eigenschaft, Eindeutigkeit) und die festgelegten Kardinalitäten für Beziehungstypen werden direkt in die `ERD`-Term-Darstellung übernommen. Die schwachen Entitätstypen (hier der `Comment`-Entitätstyp) sind wie in der Darstellung mittels ER-Diagramm daran erkennbar, dass sie kein Attribut mit Primärschlüssel-Eigenschaft besitzen, wobei partielle Schlüssel auf reguläre Attribute ohne Schlüsseleigenschaft abgebildet werden. Die einzige Ausnahme von dieser Regel und von der direkten Abbildung der ER-Diagramm-Spezifikation eines Entitätstyps auf den entsprechenden `Entity`-Wert stellt der `Metadata`-Entitätstyp dar:

```
(...)

,Entity "Metadata"
  [Attribute "Title"       (StringDom Nothing     ) NoKey  False
  ,Attribute "Description" (StringDom Nothing     ) NoKey  True
  ,Attribute "IsVisible"   (BoolDom   Nothing     ) NoKey  False]

(...)
```

Listing 4.2: `Smap.erdterm` (Entitätstyp `Metadata`)

Wie man sieht, wird das `Id`-Attribut des `Metadata`-Entitätstyps und damit der Primärschlüssel für Programme zunächst nicht im `ERD`-Term abgebildet. Stattdessen wird hierfür auf den internen Schlüssel des durch den Scaffolding-Prozess erzeugten `Metadata`-Datentyps zurückgegriffen (siehe nächster Abschnitt).

Die Abbildung aller übrigen Entitätstypen und Beziehungen erfolgt analog zu der Vorgehensweise in Listing 4.1. Die resultierende Spezifikation des Datenmodells als `ERD`-Term ist in `Smap.erdterm` zu finden.

4.1.2. Generierte Datentypen und Operationen

Bei der Projekterzeugung mit Spicey wird der im letzten Abschnitt konstruierte `ERD`-Term von dem Tool `erd2curry` zunächst auf einen neuen `ERD`-Term abgebildet, der jeden Entitätstyp um einen internen Primärschlüssel ergänzt, für jede `1`-zu-`n`-Beziehung ein Fremdschlüssel-Attribut hinzufügt und für alle `n`-zu-`m`-Beziehungen entsprechende Entitätstypen (im Folgenden als `n`-zu-`m`-Entitätstypen bezeichnet) anlegt [BHM08]. Die Definition des `Metadata`-Entitätstyps aus Listing 4.2 wird dabei beispielsweise wie folgt erweitert:

```
(...)

,Entity "Metadata"
  [Attribute "Key"                (IntDom    Nothing) PKey  False
  ,Attribute "Title"              (StringDom Nothing) NoKey False
  ,Attribute "Description"        (StringDom Nothing) NoKey True
  ,Attribute "IsVisible"          (BoolDom   Nothing) NoKey False
  ,Attribute "LanguageImplLangKey" (KeyDom "Language") NoKey False
  ,Attribute "UserAuthoringKey"    (KeyDom "User"    ) NoKey False]

(...)
```

Listing 4.3: `Smap_ERDT.term` (Entitätstyp `Metadata`)

Aus diesem erweiterten `ERD`-Term wird nun in einem nächsten Schritt das Curry-Modul `Smap` generiert, welches für alle Entitätstypen entsprechende abstrakte Datentypen, Akzessoren und die grundlegenden Datenbankoperationen (CRUD-Operationen) definiert. Die Spezifikation des (erweiterten) `Metadata`-Entitätstyps aus Listing 4.3 wird beispielsweise auf folgenden abstrakten Datentyp abbgebildet:

```
data Metadata = Metadata
  ERDGeneric.Key -- internal primary key
  String         -- title
  String         -- description
  Bool           -- isAdmin
  ERDGeneric.Key -- foreign key (ImplLang relation)
  ERDGeneric.Key -- foreign key (Authoring relation)
```

Listing 4.4: `Smap` (Abstrakter Datentyp `Metadata`)

Ebenso wird mit allen anderen, im ursprünglichen ER-Modell spezifizierten Entitätstypen und den `n`-zu-`m`-Entitätstypen `Favoriting` und `Tagging` verfahren, wobei letztere lediglich aus den jeweiligen Fremdschlüsselattributen bestehen:

```
data Tagging = Tagging
  ERDGeneric.Key -- foreign key (Metadata entity)
  ERDGeneric.Key -- foreign key (Tag entity)
```

Listing 4.5: `Smap` (Abstrakter Datentyp `Tagging`)

Wie zu erkennen ist, ermöglicht der interne Primärschlüssel (in Spicey durch den Typ `Key` aus dem Modul `ERDGeneric` repräsentiert) den einheitlichen Zugriff auf Entitäten beliebigen Typs und die einheitliche Verwaltung der Fremdschlüsselbeziehungen. Zusätzlich wird für alle Entitätstypen mit einem solchen Schlüsselattribut (also für alle Entitätstypen außer den n-zu-m-Entitätstypen) ein entsprechendes Attribut in der Datenbank angelegt, welches bei der Erzeugung einer neuen Entität automatisch und eindeutig generiert wird.[1] Um außerhalb des Moduls `Smap` zwischen den Schlüsseln verschiedener Entitätstypen unterscheiden zu können, generiert Spicey für jeden regulären Entitätstyp zudem einen entitätsspezifischen, abstrakten Schlüssel-Datentypen, der den eigentlichen internen Schlüssel kapselt:

```
data MetadataKey = MetadataKey ERDGeneric.Key
```

Über diese Schlüssel-Datentypen kann auf Anwendungsebene indirekt auf die Werte der internen Schlüssel zugegriffen werden. Zusätzlich wird der entitätsspezifische Schlüssel von den automatisch generierten Datenbankoperationen zur typsicheren Identifikation von konkreten Entitäten verwendet. Eine solche Operation ist beispielsweise die Funktion zur Erzeugung einer neuen `Version`-Entität in der Datenbank:

```
newVersionWithMetadataVersioningKey
  :: Int -> String -> String -> Time.CalendarTime -> MetadataKey
  -> KeyDatabase.Transaction Version
```

Neben den vier Attributen des `Version`-Entitätstyps wird auch der Schlüssel der zugehörigen `Metadata`-Entität erwartet. Vor dem Erzeugen der neuen `Version`-Entität prüft die Funktion, ob die betreffende `Metadata-Entität` überhaupt existiert. Im Allgemeinen gilt, dass die mit dem Modul `Smap` erzeugten Datenbankoperationen „sicher“ im Bezug auf die durch den `ERD`-Term spezifizierten Integritätsbedingungen (Eindeutigkeit von Attributen, Fremdschlüsselbeziegungen, etc.) sind. Solche Integritätsprüfungen werden ausschließlich bei Transaktionen des Typs `KeyDatabase.Transaction` *`T`* durchgeführt, wobei neben der Create-Operation eines Entitätstyps auch die Update- und die Delete-Funktionen als Transaktionen realisiert sind. Transaktionen können mit der Funktion `KeyDatabase.runT` als I/O-Aktion ausgeführt werden und liefern entweder einen Wert vom Typ *`T`* (bei der Create-Operation die erzeugte Entität, ansonsten das leere Tupel vom Typ `()`) oder einen Transaktionsfehler vom Typ `TError`.
Analog zu den Transaktionen werden die mit dem Modul `Smap` automatisch generierten Read-Operationen über Anfragen vom Typ `KeyDatabase.Query [`*`T`*`]` realisiert. Beispielsweise wird für jeden Entitätstyp *`En`* eine Funktion `queryAll`*`En`*`s` definiert, die alle in der Datenbank vorhandenen Entitäten dieses Typs abfragt:

```
queryAllTags :: KeyDatabase.Query [Tag]
```

Anfragen können über die Funktion `KeyDatabase.runQ` ausgeführt werden, welche die (möglicherweise leere) Liste der Ergebnisse als I/O-Aktion zurückliefert.

Das von Spicey aus dem `ERD`-Term erzeugte Modul `Smap` definiert also einerseits abstrakte Datentypen für alle Entitätstypen und n-zu-m-Entitätstypen (inklusive Getter- und Setter-

[1] Die aktuelle Implementierung verwendet eine SQLite3-Datenbank (`http://www.sqlite.org/`). Der Zugriff auf diese Datenbank wird hauptsächlich von den von Spicey generierten Modulen `lib/ERDGeneric` und `lib/KeyDatabase` spezifiziert und wird im Rahmen dieser Arbeit nicht weiter erläutert.

Funktionen, mit denen die Attribute einer gegebenen Entität ausgelesen oder verändert werden können) und andererseits sichere Datenbankoperationen in Form von Transaktionen und Anfragen als Schnittstelle zwischen Anwendung und Datenbank. Eine ausführlichere, allgemeine Beschreibung der generierten Schnittstelle ist in [BHM08] zu finden.

4.2. Architektur und Struktur der Webanwendung

Wie in Abschnitt 2.2.2 beschrieben, basiert Spiceys Scaffolding-Mechanismus im Wesentlichen auf dem MVC-Architekturmuster. Das Strukturierungsprinzip des MVC-Musters wird dabei durch die Modularisierung des von Spicey aus einer ERM-Spezifikation generierten Codes widergespiegelt. Genauer gesagt werden von Spicey folgende Verzeichnisse angelegt [HK12]:

`models/` Das `models/`-Verzeichnis enthält die Implementierung des Datenmodells der Spicey-Anwendung. Dies ist zunächst das im letzten Abschnitt beschriebene Modul `Smap`, welches für die im ERM spezifizierten Entitäten abstrakte Datentypen und entsprechende CRUD-Operationen definiert.

`controllers/` Nach dem Scaffolding-Prozess enthält das `controllers/`-Verzeichnis für jede ERM-Entität ein Controller-Modul, welches für die Verarbeitung korrespondierender Benutzerinteraktionen zuständig ist. Mögliche Interaktionen sind dabei zunächst die typischen Operationen auf Entitäten (auflisten, erstellen, bearbeiten und löschen), wobei jede dieser Operationen von mindestens einem Controller implementiert wird. Manche Controller sind dabei direkt (z. B. über eine URL) aufrufbar, während andere erst beim Ausführen entsprechender Aktionen in den zugehörigen Views aufgerufen werden (z. B. der für das Löschen einer Entität zuständige Controller).

`views/` Analog zu den Controller-Modulen wird im `views/`-Verzeichnis für jede ERM-Entität ein View-Modul angelegt. Die in diesem Modul enthaltenen Views werden von den Controllern aus dem korrespondierenden Controller-Modul der Entität aufgerufen und implementieren die Weboberflächen zu den erwähnten Interaktionsmöglichkeiten.

`system/` Das `system/`-Verzeichnis enthält globale, vom Datenmodell unabhängige Module. Hier werden z. B. die Module abgelegt, die Spiceys Authentifizierungs- und Autorisierungsmechanismen implementieren.

Weitere von Spicey generierte Verzeichnisse sind bspw. das `config/`-Verzeichnis, in welchem Routing und Controller-Mapping definiert werden, oder das `scripts/`-Verzeichnis, welches (unter anderem) Shell-Skripte für die Kompilierung und Installation der Anwendung bereitstellt.[2]

[2] Für die Implementierung der Anwendung sind vor allem die Verzeichnisse `models/`, `controllers/`, `views/` und `system/` interessant. Die übrigen durch Spicey generierten Verzeichnisse und Module wurden nur geringfügig modifiziert oder angepasst und werden im Kontext der Implementierung nicht weiter behandelt. Eine Beschreibung der Verzeichnisstruktur und ihrer Komponenten (inklusive dieser Verzeichnisse) ist in Anhang C zu finden.

Für die Adaption des MVC-Musters bezüglich der Implementierung der zu entwickelnden Anwendung stellte sich die strenge Aufteilung der durch Spiceys Scaffolding-Prozess erzeugten Controller- und View-Module nach den im ER-Modell spezifizierten Entitäten als nicht praktikabel heraus. Dies liegt vor allem am komponentenbasierten Entwurf der Anwendung, sowie an der Spezifikation von Programmen und deren Rolle als zentrale zu verarbeitende Daten der Anwendung. Wie die Herleitung des ER-Diagramms und der zugehörigen Entitäten in Abschnitt 3.4 gezeigt hat, lassen sich Programme als die Zusammenfassung der `Metadata`-Entität mit ihren Beziehungen beschreiben. Infolgedessen besteht die Ausprägung eines Programms stets aus einer identifizierenden `Metadata`-Entität und einer Menge von assoziierten Entitäten entsprechend dieser Beziehungen. Es liegt deshalb z. B. nahe, Programme als zusammengesetzte Daten zu implementieren, statt alle ihre assoziierten Entitäten einzeln zu verarbeiten. Zusätzlich lässt sich der im Entwurf spezifizierte komponentenbasierte Aufbau (insbesondere das Front-End; siehe Abschnitt 3.3 und Abbildung 3.3) nahezu direkt auf die MVC-Schichtenarchitektur abbilden.

Abbildung 4.1 zeigt die letztlich implementierte, unter Berücksichtigung dieser Aspekte entwickelte Abbildung der Entwurfs-Spezifikation auf Spiceys MVC-Struktur einschließlich der die einzelnen Schichten bildenden Model-, Controller- und View-Module:

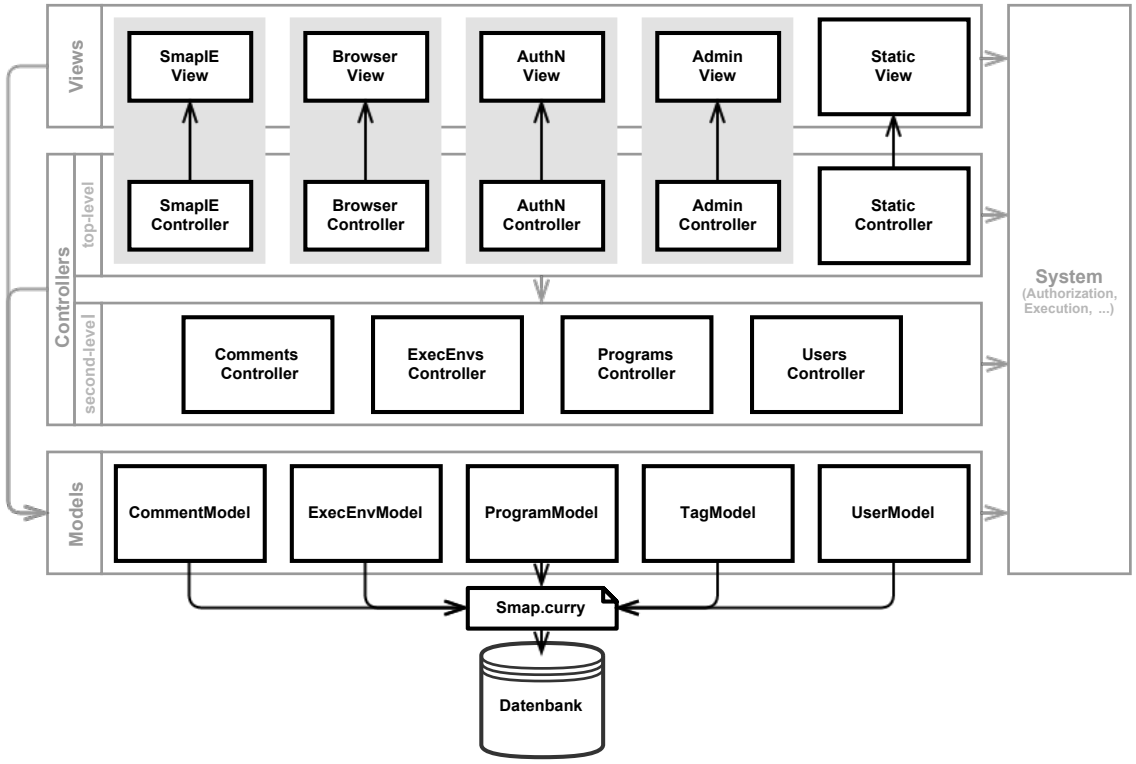


Abbildung 4.1.: Architektur der Webanwendung

Die grundlegende Implementierung der horizontalen MVC-Schichten und ihre Bedeutung im Kontext der Webanwendung Smap wird nun in den folgenden Unterabschnitten beschrieben. Die Bestandteile der vertikalen System-Schicht, welche die Basis für die Implementierung der Model-, Controller- und View-Schichten und der funktionalen Komponenten

(siehe Abschnitt 4.3) bildet, werden dabei an geeigneter Stelle eingeführt.

4.2.1. Models

Die Model-Schicht definiert die Datenbankschnittstelle der Anwendung und stellt die zentralen Operationen zur Datenmanipulation bereit. Sie repräsentiert die untere Hälfte der in Abschnitt 3.3 erwähnten Datenkontrollschicht (siehe Abbildung 3.3) und implementiert einen Teil der zu den zentralen Daten der Anwendung gehörenden Geschäftslogik. Dabei abstrahiert die Model-Schicht einerseits von den Transaktionen und Anfragen des im letzten Abschnitt beschriebenen Moduls `Smap` und andererseits von der strengen Aufteilung nach den ERM-Entitäten. Die Model-Schicht definiert dazu beispielsweise den zusammengesetzten abstrakten Datentyp `Program`, der – wie in der Spezifikation vorgesehen – das zentrale zu verarbeitende Datum der Anwendung repräsentiert. Im Sinne des MVC-Paradigmas ist die Model-Schicht zudem unabhängig von der Steuerungs- und von der Präsentationsschicht (siehe Abschnitt 2.2.2).
Wie Abbildung 4.1 andeutet, kommuniziert die Model-Schicht über das Modul `Smap` mit der Datenbank. `Smap.curry` ist dabei (konzeptionell) selbst nicht Teil dieser Schicht und wird *ausschließlich* von dieser referenziert. Dies hat den Vorteil, dass eine Änderung des Datenmodells, welche üblicherweise mit der Generierung einer neuen Version der Datenbankschnittstelle durch Spicey einhergeht, nur von der Model-Schicht abgefangen werden muss. Deren Schnittstelle für die Controller- und View-Schicht sollte dabei möglichst unverändert bleiben. Damit `Smap.curry` in einem solchen Fall problemlos gegen die modifizierte Version ausgetauscht werden kann, sollten zudem keine Modifikationen an dessen Inhalt vorgenommen werden. Zusätzliche, spezielle Operationen, die den Zugriff auf die Daten vereinfachen, und abstrakte Datentypen (wie der oben erwähnte `Program`-Datentyp) werden ausschließlich in dem entsprechenden Modul innerhalb der Model-Schicht definiert.

▷ Das Modul `ProgramModel`

Das Modul `ProgramModel` definiert und implementiert die Schnittstelle für die Verarbeitung von Programm-Objekten im Sinne des Entwurfs und ist damit das wohl wichtigste Modul der Model-Schicht. Analog zu den von Spicey generierten Datentypen für die ERM-Entitäten werden Programme als abstrakter Datentyp realisiert:

```
data Program = Program
  Metadata  -- program metadata (identifying Metadata entity)
  Language  -- source code language (ImplLang relation)
  User      -- author (Authoring relation)
  [Version] -- program versions (Versioning relation)
  [Tag]     -- program tags (Tagging relation)
  [Comment] -- program comments (Commenting relation)
  [User]    -- program favoriters (Favoriting relation)
```

Listing 4.6: `models/ProgramModel` (Abstrakter Datentyp `Program`)

Der `Program`-Datentyp bündelt die identifizierende `Metadata`-Entität mit allen assoziierten Entitäten entsprechend der im ER-Modell spezifizierten Beziehungen. Dies entspricht exakt der Definition von Programmen aus Abschnitt 3.4 (siehe auch Abbildung 3.8) und hat den Vorteil, dass alle unmittelbar mit einem Programm verknüpften Daten ohne zusätzliche Datenbankzugriffe aus dem entsprechenden `Program`-Objekt extrahiert werden können. Zu diesem Zweck exportiert das Modul `ProgramModel` eine Reihe von Selektoren, die den Zugriff auf bestimmte Attribute eines gegebenen Programms erlauben. Die (gekapselte) ID eines Programms lässt sich bspw. mit folgender Funktion auslesen:

```
programKey :: Program -> ProgramKey
programKey (Program mdata _ _ _ _ _ _) = metadataKey mdata
```

Der Schlüssel-Datentyp `ProgramKey`[3] ist dabei zweckmäßig identisch zum Schlüssel-Datentyp von `Metadata`-Entitäten:

```
type ProgramKey = MetadataKey
```

Neben Selektoren für die Attribute der identifizierenden `Metadata`-Entität (`programKey`, `programTitle`, etc.) und für die assoziierten Entitäten (`programAuthor`, `programTags`, etc.) gibt es auch spezielle Selektoren mit semantischem Charakter wie z. B. die Funktion `programLatestVersion`, welche die aktuellste Version eines Programms zurückliefert:

```
programLatestVersion :: Program -> Version
```

Es ist anzumerken, dass der Datentyp `Program` wie die Datentypen der ERM-Entitäten abstrakt ist und dessen Attribute deshalb *ausschließlich* über Selektoren ausgelesen werden können. Dies hat unter anderem den Vorteil, dass die Definition des Programm-Datentyps grundsätzlich beliebig erweiterbar ist.

Für die Verarbeitung von `Program`-Objekten stellt das Modul `ProgramModel` unter anderem folgende Funktionen zur Verfügung:

```
createProgram      :: (...)      -> IO (Either Program TError) -- shortened
updateProgram      :: Program    -> IO (Either () TError)
deleteProgram      :: Program    -> IO (Either () TError)
getProgramByKey    :: ProgramKey -> IO (Maybe Program)
```

Diese Funktionen implementieren wesentliche CRUD-Operationen für den zusammengesetzten `Program`-Datentyp, um die Handhabung von Programm-Objekten in der Steuerungsschicht so einfach und allgemein wie möglich zu gestalten. Wie zu erkennen ist, entspricht der Rückgabetyp der ersten drei Operationen dem Rückgabewert von Transaktionen (bzw. dem Typ der Funktion `runT`; siehe Abschnitt 4.1.2). Dies liegt daran, dass die von der Model-Schicht exportierten CRUD-Operationen Transaktionen und Anfragen kapseln, um etwaige, von der Implementierung des Datenbankzugriffs abhängige Operationen von der Controller-Schicht zu trennen und die Schnittstelle zu vereinfachen. Listing 4.7 zeigt beispielhaft die um die lokalen Definitionen gekürzte Implementierung der Funktion `createProgramm`, welche ein neues Programm mit gegebenen Metainformationen, einer Sprache, einem Autor, einer ersten Version des Quelltextes, dem Erstellungsdatum und einer Menge von Tags (als Liste von Tagnamen vom Typ [String]) in der Datenbank speichert.

[3] Aus Konsistenzgründen wurde statt `ProgramId` der Bezeichner `ProgramKey` gewählt.

```
1  createProgram
2    :: (String,String,Bool,Language,User,String,CalendarTime,[String])
3    -> IO (Either Program TError)
4  createProgram (title,descr,visible,lang,author,code,date,tagNames) =
5    do tagsOld <- getAllTagsWithNameIn
6       runT $ createMetadataT
7              |»= (\mdata -> createVersionT (1,"Program created.",mdata)
8                   |»= (\version -> returnT (mdata,[version])))
9              |»= (\(mdata,vers) -> mapT newTag (getNewTagNames tagsOld)
10                  |»= (\tagsNew -> let tags = tagsNew++tagsOld
11                                   in mapT_ (addTaggingT mdata) tags
12                      |» (returnT $ Program mdata lang author vers tags [] [])))
13   where ... -- shortened (getAllTagsWithNameIn, createMetadataT, ...)
```

Listing 4.7: `models/ProgramModel` (Funktion `createProgram`)

Die Erzeugung eines neuen Programms bedeutet in diesem Fall die Erzeugung einer neuen `Metadata`-Entität, einer `Version`-Entität, einer Menge von `Tag`-Entitäten (falls mindestens ein neues Tag darunter ist) und entsprechender `Tagging`-Entitäten, um die Tags mit dem Programm zu verknüpfen. Die lokal definierten Funktionen `createMetadataT` (Zeile 6) und `createVersionT` (Zeile 7) sind Transaktionen, die zunächst die `Metadata`- und `Version`-Entitäten persistieren. Hierbei erhält die erste Version des Programms die Versionsnummer `1` und die Standardnachricht `"Program created."`. In den Zeilen 9 bis 11 wird für jedes neue Tag eine Entität angelegt und die Verknüpfung der Tags mit der `Metadata`-Entität vorgenommen. Die Transaktion `Smap.newTag`, welche eine neue `Tag`-Entität mit einem gegebenen Namen in der Datenbank persistiert, erzeugt einen Transaktionsfehler, falls bereits eine `Tag`-Entität mit demselben Namen existiert (siehe Abschnitt 4.1.2). Aus diesem Grund filtert die lokale Funktion `getNewTagNames` mit Hilfe der bereits bekannten Tags (`tagsOld`; Zeile 5) die Namen der neuen Tags aus der Liste `tagNames`. Die bei den Transaktionen erzeugten Entitäten werden in Zeile 12 zusammen mit der Sprache und dem Autor zu dem resultierenden `Program`-Objekt zusammengesetzt. Die Ausführung der durch die Operatoren `(|»=)` und `(|»)` sequentiell gekoppelten Transaktionen mit der Funktion `runT` (Zeile 6) liefert entweder dieses `Program`-Objekt oder einen von einer der Transaktionen erzeugten Transaktionsfehler vom Typ `TError`.

Neben den oben genannten Operationen stellt das Modul `ProgramModel` insbesondere auch eine Schnittstelle für die Durchführung komplexer Programm-Anfragen zu Verfügung. Die Schnittstelle besteht im Wesentlichen aus der Funktion `getAllProgramsWith`, die die Sucheinstellungen in Form eines Datenobjekts vom Typ `ProgramQuery` entgegennimmt und die Ergebnisliste als I/O-Aktion zurückliefert:

```
getAllProgramsWith :: ProgramQuery -> IO [Program]
```

Der abstrakte Datentyp `ProgramQuery` repräsentiert eine Anfrage an die Datenbank, deren Parameter über eine Reihe von Attributen festgelegt werden können:

```
data ProgramQuery = ProgramQuery
  (Maybe ProgramKey) -- program key
  String             -- keyword
  (Bool,Bool,Bool)   -- targets (title, descr., tags)
```

```
  Bool               -- only visible programs?
  String             -- implementation language name
  String             -- name of the author
  [String]           -- list of favoriter names
  (Sorting Program)  -- a program sorting
  OrderingType       -- order of the result list
```

Listing 4.8: `models/ProgramModel` (Abstrakter Datentyp `ProgramQuery`)

Für die Durchführung einer neuen Anfrage definiert `ProgramModel` das `ProgramQuery`-Objekt `defaultProgramQuery`, welches zunächst Standardwerte für alle Parameter definiert und über Akzessoren modifiziert werden kann. Folgender Ausdruck liefert beispielsweise alle Programme des Nutzers mit dem Namen `"lkm"`:

```
getAllProgramWith $ withExactAuthorName "lkm" $ defaultProgramQuery
```

Die konkrete Durchführung einer komplexen Anfrage wird von der (nicht exportierten) Funktion `runProgramQuery` realisiert:

```
1 runProgramQuery :: ProgramQuery -> IO [Program]
2 runProgramQuery progQuery@(ProgramQuery _ _ _ _ _ _ _ (sType,sFunc) oType) =
3   do mdatas   <- runQ queryAllMetadatas
4      progs    <- metadatasToPrograms mdatas
5      filter   <- getFilter progQuery
6      ordering <- getOrdering sType oType
7      return $ mergeSort (ordering sFunc) $ filter progs
```

Listing 4.9: `models/ProgramModel` (Funktion `runProgramQuery`)

In der aktuellen Implementierung werden dazu zunächst alle `Metadata`-Entitäten aus der Datenbank ausgelesen (Zeile 3) und anschließend von der Funktion `metadatasToPrograms` in korrespondierende `Program`-Objekte umgewandelt (Zeile 4). Die Funktion `getFilter` in Zeile 5 erzeugt eine Funktion vom Typ `[Program] -> [Program]`, welche eine Liste von Programmen entsprechend den im `ProgramQuery`-Objekt spezifizierten Sucheinstellungen filtert. Die gefilterte Liste wird anschließend mit Mergesort sortiert (Zeile 7). Die Art der Sortierung (der erste Parameter der `mergeSort`-Funktion) wird durch die letzten beiden Parameter des `ProgramQuery`-Objekts festgelegt. Die Typen `Sorting a` und `OrderingType` (und die Funktion `getOrdering`) werden vom Modul `system/Models` bereitgestellt:

```
type Sorting a    = (SortingType,a -> a -> Bool)
data SortingType  = LEQ
data OrderingType = Ascending | Descending
```

Die zweite Komponente des `Sorting`-Paars definiert hierbei die dem Sortierungstyp in der ersten Komponente entsprechende Relation zwischen zwei Werten des Typs `a`. Aus einem `SortingType` und einem `OrderingType` berechnet die Funktion `getOrdering` wiederum eine Funktion vom Typ `(a -> a -> Bool) -> (a -> a -> Bool)`, die diese Relation geeignet modifiziert (also ggf. „umdreht“). Für die Sortierung der gefilterten `Program`-Listen stellt das Modul `ProgramModel` verschiedene Sortierungsrelationen zur Verfügung. So gibt es beispielsweise Relationen für die Sortierung nach Erstellungsdatum oder dem Titel des Programms (siehe Abschnitt 3.3.2):

```
leqProgramFirstVersionDate :: Sorting Program
leqProgramTitle            :: Sorting Program
```

Das `ProgramQuery`-Konzept wird von der Browser-Komponente für die Implementierung komplexer Programm-Suchen benutzt (siehe Abschnitt 4.3.3). Die Kapselung des Datentyps beruht auf dem Gedanken, die konkrete Implementierung der Anfragen in späteren Erweiterungen leicht austauschen zu können (z. B. durch eine Implementierung, die auf „echten“ SQL-Anfragen beruht und die Filterung und Sortierung auf Datenbankebene durchführt).

▷ Sonstige Models

Analog zum `Program`-Model definieren `CommentModel`, `TagModel` und `UserModel` sichere Standard-Operationen zum Erzeugen, Manipulieren und Auslesen der jeweiligen zugrunde liegenden ERM-Entitäten auf Basis der vom Modul `Smap` bereitgestellten Transaktionen und Anfragen. Über das `UserModel` können zusätzlich `Favoriting`-Entitäten für gegebene Nutzer und Programme erzeugt und gelöscht werden:

```
addFavoriting    :: (User,Program) -> IO (Either () TError)
removeFavoriting :: (User,Program) -> IO (Either () TError)
```

Das `ExecEnvModel` fasst Sprachen und Ausführungssysteme entsprechend der `LangImpl`-Beziehung zu Ausführungsumgebungen (Execution Environments; siehe Abbildung 3.8) zusammen:

```
type ExecEnv = (Language,[System])
```

Ausführungsumgebungen des Typs `ExecEnv` werden vor allem vom interaktiven Editor SmapIE verwendet, welcher zu einer geladenen Sprache stets auch alle Ausführungssysteme kennen muss (siehe Abschnitt 4.3.2). Um den Zugriff auf Ausführungsumgebungen zu vereinfachen stellt das `ExecEnvModel` ensprechende Model-Operationen zur Verfügung. Die Funktion `getExecEnvByLanguageName` gibt bspw. das `ExecEnv`-Paar zurück, dessen Sprache durch den gegebenen Namen identifiziert wird:

```
getExecEnvByLanguageName :: String -> IO (Maybe ExecEnv)
```

Existiert keine solche Sprache (und somit auch keine zugehörige Ausführungsumgebung) wird der Wert `Nothing` vom Typ `Maybe ExecEnv` als I/O-Aktion zurückgegeben. Zusätzlich zu den Operationen auf den `ExecEnv`-Typ definiert das `ExecEnvModel` aber auch Operationen auf einzelne `Language`- und `System`-Entitäten (z. B. `createLanguage` und `createSystem`).

4.2.2. Controllers

Wie bereits in Abschnitt 2.2.3 über die Funktionsweise von Spicey-Anwendungen erläutert wurde, werden Controller in Spicey durch I/O-Aktionen repräsentiert, die den Inhalt des nächsten anzuzeigenden HTML-Dokuments zurückgeben:

```
type Controller = IO [HtmlExp]
```

Analog zu dem im letzten Abschnitt erwähnten Modul `Models` enthält die System-Schicht auch ein Modul `Controllers`, welches unter anderem diesen Typ, allgemeine Controller für die Darstellung von Fehlerseiten und Funktionen für die Konstruktion von HTML-Formularen (insbesondere `getForm`) zur Verfügung stellt. Listing 4.10 zeigt bspw. die Controller-Funktion `showErrorPage`, welche zur Rückgabe einer allgemeinen Fehlerseite verwendet werden kann:

```
showErrorPage :: String -> String -> Maybe String -> Controller
showErrorPage title controllerMsg mInternalMsg =
  return [renderErrorPage (title,controllerMsg,mInternalMsg)]
```

Listing 4.10: `system/Controllers` (Funktion `showErrorPage`)

Die Funktion `renderErrorPage` erzeugt hier aus einem gegebenen Titel und zwei Fehlermeldungen den Inhalt der darzustellenden View in Form einer `HtmlExp`. Das Modul `Controllers` wird von allen Modulen der Controller-Schicht importiert. Wie in Abbildung 4.1 zu sehen ist, werden diese Module konzeptionell in zwei separate Schichten eingeteilt, deren Aufgaben in den folgenden beiden Abschnitten erläutert werden.

▷ Second-Level-Controller

Second-Level-Controller (auch Entitäten-Controller) kapseln im Wesentlichen die in der Model-Schicht definierten `create`-, `update`- und `delete`-Operationen (bzw. Transaktionen) der jeweiligen Entitäten, um diese in den Top-Level-Controllern (auch Komponenten-Controller) möglichst generisch (wieder-)verwenden zu können. Sie bilden die obere Hälfte der in Abbildung 3.3 dargestellten Datenkontrollschicht und implementieren Teile der Geschäftslogik des Datenmodells, wie z. B. die Validierung von Eingaben via Datenbankabgleich. Außerdem stellen sie eine Schnittstelle für die Handhabung von (Transaktions-) Fehlern bereit. Ein gutes Beispiel für einen Second-Level-Controller ist die Controller-Funktion `doCreateUser`[4] des Moduls `UsersController`, die für die Erzeugung eines neuen Nutzers zuständig ist:

```
1 doCreateUser
2   :: ((String,String,String,String) -> Controller,Maybe Alert) -- name not unique
3   -> ((String,String,String,String) -> Controller,Maybe Alert) -- mail not unique
4   -> (User                          -> Controller,Maybe Alert) -- success case
5   -> (String,String,String,String) -- input data (name, email, password, password conf.)
6   -> Controller                     -- returned controller
```

Listing 4.11: `controllers/UsersController` (Signatur `doCreateUser`)

Beim Erzeugen eines neuen Nutzers muss zunächst geprüft werden, ob sowohl Name als auch E-Mail-Adresse nicht schon von einem anderen Nutzer verwendet werden, wozu

[4] Das Präfix `do` kennzeichnet Controller, die für die Verarbeitung von Formularen zuständig sind und aus diesen aufgerufen werden (was für alle Second-Level-Controller gilt). Demgegenüber stehen Controller mit dem Präfix `show`, welche im Allgemeinen direkt über eine URL aufrufbar sind und deren primäre Aufgabe die Darstellung einer Weboberfläche ist.

ein Datenbankzugriff notwendig ist. Insbesondere im Falle der Registrierung sollte der Nutzer bei einem entsprechenden Fehlerfall nicht unbedingt auf eine Fehlerseite weitergeleitet werden, sondern die Gelegenheit erhalten, einen neuen Namen oder eine neue E-Mail-Adresse einzugeben. Um solche Situationen im Allgemeinen beliebig handhaben zu können, erwarten Second-Level-Controller für jeden Fehlerfall und den Erfolgsfall ein Paar bestehend aus einem Controller (bzw. einer Controller-Funktion), an den die weiteren Steuerungsmaßnahmen delegiert werden, und einer optionalen (Warn-)Meldung in Form eines `Alerts`. Letztere werden im Modul `system/Alerts` definiert und bestehen aus einem `AlertType` und einer `String`-Nachricht mit dem Inhalt der Meldung:

```
type Alert     = (AlertType,String)
data AlertType = ErrorAlert | InfoAlert | SuccessAlert | WarningAlert
```

Das Setzen eines `Alerts` bewirkt dessen Darstellung auf der nächsten darzustellenden Weboberfläche. Im Kern entsprechen `Alerts` damit den von Spicey standardmäßig unterstützten Seitennachrichten (Page Messages), welche in [HK12] beschrieben werden.
`doCreateUser` wählt nun auf Basis der Eindeutigkeitstests einen der Controller aus den Zeilen 1 bis 3 und führt diesen aus. Gegebenenfalls wird dabei der zugehörige `Alert` gesetzt. Im Falle der erfolgreichen Erzeugung des Nutzers wird die von der Transaktion zurückgegebene `User`-Entität dabei an den nächsten Controller übergeben. Andernfalls werden die ursprünglichen Eingabedaten (Zeile 4) weitergereicht.

Second-Level-Controller haben also keine Kenntnis über die konkreten funktionalen Komponenten, die auf sie zugreifen und können an beliebiger Stelle (und mehrfach) verwendet werden. Als Funktionen höherer Ordnung sind sie über die Controller-Parameter an die jeweilige Zugriffsart anpassbar. Dass Second-Level-Controller konzeptionell nicht in das Model integriert wurden, hat dabei vor allem den Grund, dass sie per Definition mit dem Rückgabetyp `Controller` konsequenterweise eher als Controller anzusehen sind.

▷ Top-Level-Controller

Die Top-Level-Controller-Schicht wird bis auf eine Ausnahme von den Steuerungseinheiten der funktionalen Komponenten gebildet (siehe Abbildung 4.1) und implementiert deren Kernfunktionalitäten. Im Allgemeinen greift sie für Anfragen auf das Model und für Transaktionen auf die Second-Level-Controller-Schicht zu. Im Gegensatz zur Second-Level-Controller-Schicht definiert die Top-Level-Controller-Schicht auch direkt über die URL aufrufbare Controller-Funktionen (also Funktionen mit dem Präfix `show`), die die zentralen grafischen Schnittstellen der Webanwendung erzeugen. Jedes Controller-Modul der Top-Level-Controller-Schicht exportiert zu diesem Zweck eine Controller-Funktion, den sogenannten Delegations-Controller, der basierend auf dem URL-Pfad (bzw. den Parametern des CGI-Skripts) einen konkreten Controller des Moduls auswählt.

```
1 staticController :: Url -> Controller
2 staticController url@(path,_) =
3   case path of
4     [""]      -> showLandingPage
5     ["help"]  -> showHelpPage
6     ["about"] -> showAboutPage
```

```
7     _          -> showInvalidUrlErrorPage url -- from module 'Controllers'
```

Listing 4.12: `controllers/StaticController` (Funktion `staticController`)

Listing 4.12 zeigt beispielhaft den Delegations-Controller des Moduls `StaticController`, welches für die Darstellung der statischen Weboberflächen (Landing Page, Hilfsseite und About) der Anwendung Smap zuständig ist und die oben erwähnte Ausnahme bildet, da es zu keiner funktionalen Komponente gehört. Wie bereits in Abschnitt 2.2.3 erwähnt, erfolgt die Auswahl eines Controllers durch den Dispatcher über die Module `ControllerMapping` und `RoutesData`. Im Falle der Webanwendung Smap wird dabei der Delegations-Controller eines Top-Level-Controllers gewählt und auf die zuvor abgefragte URL in Form eines `Url`-Wertes angewendet. Die Schnittstelle für den Zugriff und die Verarbeitung von URLs wird durch das Modul `system/Url` bereitgestellt. Der Typ `Url` ist dabei wie folgt definiert:

```
type Url = ([String],[(String,String)])
```

Die erste Komponente beschreibt die Bestandteile des Pfades, während in der zweiten Komponente die Name-Wert-Paare des Query Strings als Liste von `String`-Paaren gespeichert werden.
Der Delegations-Controller wählt nun auf Basis des Pfads den eigentlichen auszuführenden Controller aus. Im Beispiel aus Listing 4.12 wird in Zeile 5 bspw. der Pfad `/help` auf den Controller `showHelpPage` gemappt, welcher die View-Funktion `helpPage` aus dem Modul `views/StaticView` aufruft und die entsprechende Weboberfläche als I/O-Aktion zurückgibt. Auf analoge Weise wird auch bei der Landing Page (Route: `/`) und der About Page (Route: `/about`) verfahren.

Die konkrete Umsetzung der restlichen Komponenten-Controller ist Teil der in Abschnitt 4.3 beschriebenen Implementierung der funktionalen Komponenten und wird in diesem Zusammenhang erläutert.

4.2.3. Views

Eine View ist im Allgemeinen eine Funktion mit dem Rückgabewert `View` (siehe Abschnitt 2.2.3), welche von einem Controller auf die darzustellenden Daten und etwaige, für die Verarbeitung von Formulareingaben zuständige Controller angewendet wird. Diese Funktionen sind analog zu den Controller-Funktionen in den Modulen der View-Schicht organisiert (siehe Abbildung 4.1). Sinnvollerweise existiert dabei für jeden Top-Level-Controller, einschließlich des `StaticControllers`, ein View-Modul, welches die Oberflächen der entsprechenden Komponente zur Verfügung stellt.

▷ Allgemeine Implementierung

Die Konstruktion der Views basiert im Allgemeinen auf der durch das Modul `HTML` bereitgestellten Schnittstelle (siehe Abschnitt 2.1.2). Für die Entwicklung der Weboberflächen wurde zudem auf das Front-End-Framework Bootstrap (siehe Abschnitt 2.3) zurückgegriffen, um ein einheitliches, simples und übersichtliches Erscheinungsbild zu gewährleisten.

Abbildung 4.2 zeigt beispielhaft die Weboberfläche für die Startseite (Landing Page) der Webanwendung Smap:

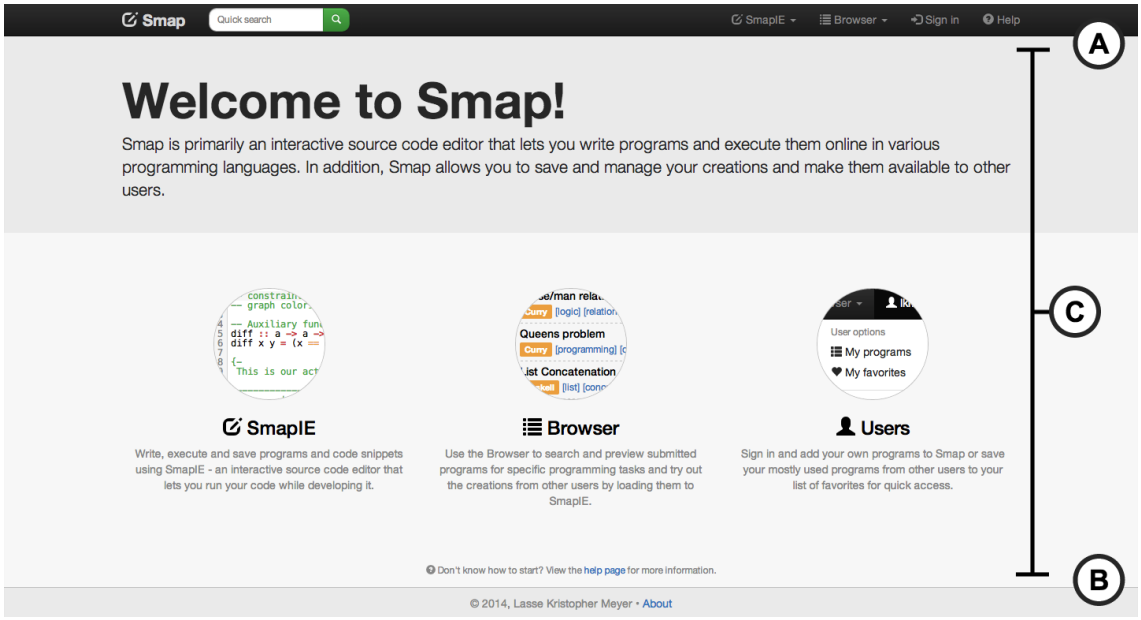


Abbildung 4.2.: Startseite der Webanwendung Smap

Das Standard-Rahmenlayout, in welches jede View von der Funktion `getForm` eingebettet wird, besteht aus der Navigations- und der Fußleiste (Ⓐ und Ⓑ), deren zugehörige HTML-Ausdrücke von dem Modul `system/SmapHtml` bereitgestellt werden. `SmapHtml` definiert eine Reihe anwendungsspezifischer HTML-Komponenten für das Rahmenlayout und die allgemeine Formularkonstruktion, wie beispielweise die Formularparameter für die einzubindenen CSS- und JavaScript-Skripte, aber auch für den eigentlichen Inhalt der Views (z. B. Buttons und Icons). Der Bereich Ⓒ zwischen Navigations- und Fußleiste wird von dem eigentlichen Inhalt der View gebildet. Im Beispiel aus Abbildung 4.2 wird dieser Inhalt von der View `landingPage` aus dem Modul `StaticView` berechnet:

```
landingPage :: View
landingPage =
  [jumbotron
    [container
      [h1 []  [b [] [text "Welcome to Smap!"]]
      ,p [] [text headerText]]]
  ,(container `withId` "landing-page")
    [row
      [...] -- shortened (HTML expressions of info columns)
    ,(row `withId` "help-line")
      [helpIcon,text $ ' ':helpText]]]
  where ... -- shortened (headerText, helpText, ...)
```

Listing 4.13: `views/StaticView` (Funktion `landingPage`)

Um die Konstruktion Bootstrap-kompatibler Weboberflächen mit dem `HTML`-Modul zu er-

leichtern, stellt das Modul `lib/Bootstrap3` Konstruktoren für einige der von Bootstrap spezifizierten Gestaltungsvorlagen und Komponenten bereit (z. B. `jumbotron`, `container` und `row` aus Listing 4.13). Ansonsten wurde hauptsächlich mit den Schnittstellen der Module `HTML` und `lib/Html5` gearbeitet. Letzteres erweitert das Modul `HTML` unter anderem um einige HTML5-Elemente und definiert zusätzliche Operatoren für die Verknüpfung von HTML-Elementen mit CSS-Klassen oder IDs (z. B. `withId`).

▷ WUI-Formulare

Für die Erstellung neuer Entitäten (Programme, Nutzer, Sprachen, etc.) und für die Implementierung des Authentifizierungsformulars wird auf das in Abschnitt 2.1.2 beschriebene WUI-Konzept der Curry-Bibliothek `WUI` zurückgegriffen. Analog zum Modul `SmapHtml` gibt es für diesen Zweck das Modul `system/SmapWui`, welches die anwendungsspezifischen Grundbausteine für die Konstruktion der benötigten Widgets bereitstellt. Beispielsweise definiert `SmapWui` das folgende Widget für die typsichere Eingabe von `String`-Werten durch ein Textfeld:

```
wSmapString
  :: (String,String,Bool) -- icon name, type attribute value and autofocus
  -> String               -- textfield label
  -> String               -- placeholder attribute value
  -> (String,String)      -- help text and error message
  -> WuiSpec String       -- returned widget
```

Alle von `SmapWui` bereitgestellten Widgets (atomare Widgets sowie WUI-Kombinatoren) erhalten ein anwendungsspezifisches Rendering auf Basis des Bootstrap-Frameworks, welches über die Parameter des Widgets modifiziert werden kann. Die zusammengesetzten Widgets, die letztendlich in den Views verwendet werden (z. B. das Widget `wProgram`, welches alle für die Erstellung eines neuen Programms notwendigen Eingaben spezifiziert oder das Widget `wSignInData`, welches den Nutzernamen und das Passwort zur Authentifizierung entgegennimmt), werden in den View-Modulen der entsprechenden Komponenten definiert. Mithilfe der Funktion `renderWuiForm` aus dem Modul `system/Views`, welches bspw. auch den Typ `View` exportiert, werden diese Widgets dann in der Regel zu einer vollständigen View ergänzt:

```
renderWuiForm
  :: WuiSpec a -> a -> (a -> Controller) -> [HtmlExp] -> [HtmlExp] -> [HtmlExp]
  -> [HtmlExp] -> [HtmlExp] -> [HtmlExp]
```

`renderWuiForm` erwartet dazu neben dem zusammengesetzten Widget vom Typ `WuiSpec a`, den initialen Eingabedaten und dem Controller, der die Formulareingaben verarbeitet, auch eine Reihe von HTML-Komponenten, mit denen das Erscheinungsbild des Formulars angepasst werden kann.

Unter Zuhilfenahme des WUI-Konzepts wird ein Großteil der Validierung von Benutzereingaben von der Präsentationsschicht (bzw. dem Modul `WUI`) übernommen. Dies gilt vor allem für Eingabefehler, die keinen Abgleich mit der Datenbank erfordern, wie bei Pflichteingaben oder Eingaben mit besonderer Formatierung (z. B. E-Mail-Adressen). Alle anderen Validierungsprozesse werden von der Controller-Schicht übernommen, welche die Eingaben

anschließend für die weitere Verarbeitung in der Model-Schicht geeignet formatiert.

4.3. Implementierung der funktionalen Komponenten

In den folgenden Abschnitten wird nun die Implementierung der in Abschnitt 3.3 entworfenen funktionalen Komponenten und ihrer Kernfunktionalitäten erläutert. Hierzu wird zunächst die Autorisierungs-Komponente beschrieben, welche als vertikale Komponente Teil der System-Schicht ist (siehe Abbildung 4.1) und sowohl von der Steuerungs- als auch von der Präsentationsschicht in allen Komponenten verwendet wird.

4.3.1. Autorisierung

Die Autorisierungs-Komponente (bzw. das in 3.1.1 beschriebene Rollensystem) wird durch die von den Modulen `Authorization` und `AuthorizedOperations` im Unterverzeichnis `system/` zur Verfügung gestellten Datentypen und Funktionen implementiert und besteht im Wesentlichen aus den folgenden Elementen:

▷ Der Datentyp `AuthZData`

Der Datentyp `Authorization.AuthZData` repräsentiert den Typ des aktuellen Nutzers und bildet die Grundlage zur Umsetzung des Rollensystems:

```
data AuthZData = Guest | Standard String | Admin String
```

Wie bereits einleitend in Abschnitt 3.3 angedeutet wurde, wird der Typ eines Nutzers über das Auslesen der beim Authentifizierungs-Prozess angelegten Sitzungsdaten ermittelt. Zu diesem Zweck definiert das Modul `Authorization` die Funktion `getAuthZData`, welche die Sitzungsdaten über das Modul `Authentication` (siehe Abschnitt 4.3.4) ausliest und in ein Objekt des Typs `AuthZData` umwandelt:

```
getAuthZData :: IO AuthZData
```

`AuthZData`-Objekte sind das zentrale Kriterium für die Erteilung von Zugriffsrechten auf Nutzeroperationen. Da die Autorisierung einer Operation im Allgemeinen auch von der Identität des Nutzers abhängt (z. B. beim Löschen eines Programms) wird im Falle authentifizierter Nutzer auch der Nutzername gespeichert.

▷ Zugriffstypen und Nutzeroperationen

Autorisierungsanfragen werden immer im Kontext einer bestimmten Operation gestellt und bewertet. Im Falle der Webanwendung Smap wurden diese Operationen, zu denen beispielsweise auch die in Abbildung 3.1 spezifizierten Operationen auf Programme gehören, bereits im Entwurf auf die verbleibenden vier funktionalen Komponenten aufgeteilt.

Auf Basis dieser Aufteilung definiert das Modul `AuthorizedOperations` für jede funktionale Komponente *`Comp`* einen Datentyp *`Comp`*`AccessType`, dessen Konstruktoren die von der Komponente implementierten Operationen repräsentieren, sowie eine Funktion *`comp`*`Operation`, die für einen Wert des Typs *`Comp`*`AccessType` und ein gegebenes `AuthZData`-Objekt entscheidet, ob die entsprechende Operation von dem durch das Objekt beschriebenen Nutzer ausgeführt werden darf. Die Operationen des interaktiven Editors SmapIE werden beispielsweise durch folgenden Zugriffstyp beschrieben:

```
data SmapIEAccessType
  = ShowSmapIE (Maybe Program) | CreateProgram | CreateVersion Program
  | ExecuteProgram
```

Der Zugriffstyp `CreateVersion` *`prog`* beschreibt hierbei die Operation zur Erzeugung einer neuen Version für das Programm *`prog`*. Die Funktion `smapIEOperation` definiert nun die konkreten Zugriffsrechte für diese Operation:

```
1  smapIEOperation :: SmapIEAccessType -> AuthZData -> AccessResult
2  smapIEOperation accessType authzData =
3    case accessType of
4      ... -- shortened (access types ‘ShowSmapIE‘ and ‘CreateProgram‘)
5      CreateVersion prog -> case authzData of
6        Guest         -> AccessDenied createVersDeniedGErr
7        Standard name -> if name ‘authored‘ prog
8                         then AccessGranted
9                         else AccessDenied createVersDeniedSErr
10       Admin    _    -> AccessGranted
11     _ -> AccessGranted
12   where ... -- shortened (error messages, e.g. ‘createVersDeniedErr‘)
```

Listing 4.14: `system/AuthorizedOperations` (Funktion `smapIEOperation`)

Die Zeilen 6 bis 10 spezifizieren, dass die Operation nur vom Autor des Programms und Nutzern mit Administratorrechten durchgeführt werden darf. Allen anderen Nutzern wird der Zugriff hingegen verweigert. Das Ergebnis einer Autorisierungsanfrage wird in Spicey dabei durch den Typ `Authorization.AccessResult` repräsentiert [HK12].

Ein Vorteil der vorgenommenen Implementierung des Rollensystems und der Beschreibung der Zugriffsrechte durch Zugriffstypen ist die simple Erweiterbarkeit. Für die Spezifikation einer neuen Operation muss nur ein entsprechender Konstruktor für den Zugriffstyp und ein weiterer `case`-Fall für die zugehörige Komponentenfunktion definiert werden.

▷ Autorisierungsanfragen

Um in der Steuerungs- und in der Präsentationsschicht nicht mit den Ergebnissen von Autorisierungsanfragen in Form von `AccessResults` hantieren zu müssen, definiert das Modul `Authorization` die Funktionen `checkAuthorization` und `byAuthorization`:

```
checkAuthorization
  :: (AuthZData -> AccessResult) -> (AuthZData -> Controller) -> Controller
byAuthorization :: AccessResult -> a -> (String -> a) -> a
```

`checkAuthorization` wird dabei zur Autorisierung von Controllern verwendet, während `byAuthorization` bspw. für die Definition von autorisierten View-Elementen geeignet ist. Beispiele für die Verwendung der beiden Funktionen sind in den Code-Beispielen der folgenden Abschnitte zu finden.

4.3.2. SmapIE

Die funktionalen Komponenten mit grafischer Benutzerschnittstelle (siehe Abbildung 3.3) werden im Allgemeinen durch jeweils einen Top-Level-Controller und ein View-Modul implementiert (vgl. Abbildung 4.1). Im Falle der SmapIE-Komponente sind dies die Module `controllers/SmapIEController` und `views/SmapIEView`. Der Zugriff auf eine Komponente erfolgt wiederum über die durch den Delegations-Controller spezifizierten Routen, die die mit dem Webbrowser aufrufbaren URLs repräsentieren. Für den interaktiven Editor sind dies die Routen `/new/`*`langName`* und */`*`progKey`*, auf welche entsprechend dem Entwurf in Abschnitt 3.3.1 über das Dropdown-Menü in der Navigationsleiste und über den Browser zugegriffen werden kann:

```
smapIEController :: Url -> Controller
smapIEController url@(path,_) =
  case path of
    ["new",langName] -> showBlankSmapIE langName
    [progKey]        -> validateKeyAndApply (readProgramKey progKey) url
                                            showProgramInSmapIE
    _                -> showInvalidUrlErrorPage url
```

Listing 4.15: `controllers/SmapIEController` (Funktion `smapIEController`)

Der Controller `showBlankSmapIE :: String -> Controller`, auf den die erste Route verweist, öffnet eine neue Instanz des Editors, die mit der Ausführungsumgebung der Sprache *`langName`* verknüpft wird. Über die zweite Route wird hingegen das Programm mit der ID *`progKey`* in den Editor geladen. Um das entsprechende Programm aus der Datenbank auslesen zu können, muss die textuelle Repräsentation der ID dazu zunächst mit der Funktion `Controllers.validateKeyAndApply` in einen Wert des Typs `ProgramKey` umgewandelt werden. Falls dieser Vorgang erfolgreich ist (falls `progKey` also eine valide ID repräsentiert), wird der `ProgramKey`-Wert anschließend dem Controller `showProgramInSmapIE` vom Typ `ProgramKey -> Controller` übergeben. Andernfalls wird eine Fehlerseite angezeigt.
`showBlankSmapIE` und `showProgramInSmapIE` lesen nun über die Model-Schicht die zur Sprache mit dem Namen *`langName`* gehörende Ausführungsumgebung bzw. das zur ID passende Programm und dessen zugehörige Ausführungsumgebung aus der Datenbank aus und übergeben diese dem internen Controller `showSmapIE`, welcher im Allgemeinen für die Darstellung des interaktiven Editors zuständig ist:

```
1 showSmapIE
2   :: Maybe Program -> ExecEnv -> Maybe ExecResult -> String -> String -> Controller
3 showSmapIE mProg execEnv mExecRes initCode initSystemKey =
4   checkAuthorization (smapIEOperation $ ShowSmapIE mProg) $ \authzData ->
5   return $ smapIE mProg execEnv mExecRes initCode initSystemKey
```

```
6                 doExecuteProgram tryShowProgramCreationForm
7                 tryShowVersionCreationForm authzData
```

Listing 4.16: `controllers/SmapIEController` (Funktion `showSmapIE`)

`showSmapIE` erhält neben dem (im Falle des leeren Editors nicht gegebenen) Programm und der mit der Editor-Instanz assoziierten Ausführungsumgebung auch einen optionalen Wert vom Typ `ExecResult`, welcher ggf. das Ergebnis einer durchgeführten Ausführung repräsentiert (siehe zweiter Unterabschnitt), den initialen Inhalt des Quelltexteingabefeldes und den Schlüssel des initial ausgewählten Ausführungssystems (siehe Zeile 2 und 3). In Zeile 4 wird nun zunächst mit der im letzten Abschnitt erwähnten Funktion `checkAuthorization` getestet, ob der ausführende Nutzer für die Durchführung der Operation (in diesem Fall das Anzeigen des Editors mit einem möglicherweise gegebenen Programm) autorisiert ist. Dies ist bspw. nicht der Fall, falls das gegebene Programm privat ist, der Nutzer aber nicht der Autor ist. Bei erfolgreicher Autorisierung wird das für die Bewertung der Anfrage von der Funktion `checkAuthorization` ausgelesene `AuthZData`-Objekt für die eventuelle Weiterverarbeitung in der View (in der Regel für die Definition autorisierter View-Elemente) über eine anonyme Funktion an den eigentlichen Rumpf des Controllers übergeben. Andernfalls wird eine entsprechende Fehlerseite angezeigt. Im Falle des Controllers `showSmapIE` besteht der Rumpf dabei aus der Berechnung der zentralen Weboberfläche des interaktiven Editors durch die View-Funktion `SmapIEView.smapIE` und deren Rückgabe als I/O-Aktion (Zeile 5 bis 7). Abbildung 4.3 zeigt beispielhaft die Weboberfläche des interaktiven Editors mit einem aus dem Browser (bzw. über die URL) geladenen Programm und die Umsetzung des Entwurfs aus Abbildung 3.4:

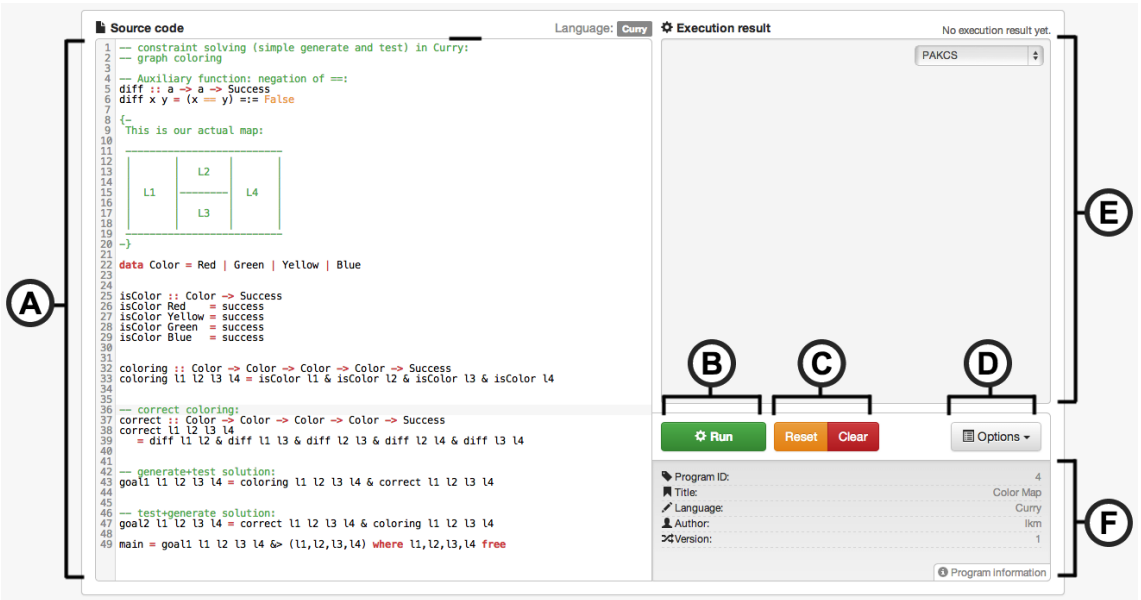


Abbildung 4.3.: Der interaktive Editor SmapIE (View-Bereich)

Analog zum Controller `showSmapIE` ist `smapIE` die zentrale View-Funktion für die Darstellung des interaktiven Editors und bildet mit den WUI-Formularen für die Erzeugung neuer Programme und Versionen (siehe letzter Unterabschnitt) die Schnittstelle des Moduls

`SmapIEView`. Wie Listing 4.16 zeigt, basiert die Berechnung der Weboberfläche des Editors im Wesentlichen auf den der Funktion `showSmapIE` übergebenen Argumenten (Zeile 5). Als zusätzliche Parameter erhält `smapIE` in den Zeilen 6 und 7 Controller für die Verarbeitung der möglichen Nutzeraktionen (Ausführung des Quelltextes, etc.) und das `AuthZData`-Objekt für das Ein- bzw. Ausblenden von Optionen (siehe nächste Unterabschnitte).

▷ Erstellung und Bearbeitung von Quelltext

Die Erstellung und Bearbeitung von Quelltext erfolgt – wie in der Entwurfsskizze vorgesehen – im Eingabefeld Ⓐ. Um die Quelltextverarbeitung möglichst benutzerfreundlich und effizient zu gestalten und um einen ähnlichen Arbeitsfluss wie beim Verwenden des Editors einer integrierten Entwicklungsumgebung (siehe Abschnitt 3.2) zu erreichen, ist das Eingabefeld als CodeMirror-Editor realisiert (siehe Abschnitt 2.4). Die CodeMirror-Instanz wird dabei wie im Beispiel in Abbildung 2.7 über die `fromTextArea`-Methode erzeugt, indem das zugehörige `textarea`-Element mit der ID `editor` verknüpft und durch nachfolgenden, eingebetteten JavaScript-Code zu einem CodeMirror-Editor ergänzt wird. Listing 4.17 zeigt die entsprechenden Ausschnitte aus der View-Funktion `smapIE`:

```
1  ...
2  ,textarea [] codeRef initCode ‘addId‘ "editor"
3  ,script [] [text codeMirrorInstance]
4  ...
5  codeMirrorInstance = -- locally defined
6    "var editor = document.getElementById(’editor’);"
7    "var code = CodeMirror.fromTextArea(editor, {\n"++
8    "  mode: ’"++map toLower langName++"’,\n"++
9    "  theme: ’smap’\n"++
10   "  lineNumbers: true,\n"++
11   ... -- shortened (further CodeMirror (addon) options, e.g. ‘tabSize‘)
12   "});\n"
13 ...
```

Listing 4.17: `views/SmapIEView` (Funktion `smapIE`, CodeMirror-Editor)

Das Argument `codeRef` (Zeile 2) ist hierbei die mit dem `textarea`-Element bzw. der CodeMirror-Instanz verknüpfte CGI-Referenz, über die der Quelltext mit einem HTML-Handler ausgelesen werden kann (siehe Abschnitt 2.1.2). Dies geschieht beispielsweise bei der Ausführung des Quelltextes über den Button Ⓑ (siehe nächster Abschnitt). Wie weiterhin in Zeile 8 zu erkennen ist, hat die Einbettung des JavaScript-Codes und dessen Darstellung als `String`-Wert in Curry den Vorteil, dass der von CodeMirror verwendete Sprachmodus dynamisch entsprechend der momentan über die Ausführungsumgebung geladenen Sprache gesetzt werden kann.[5]

[5] Das Setzen der `mode`-Option hat natürlich nur einen Effekt, wenn auch ein entsprechendes Sprachmodus-Skript existiert und in das HTML-Dokument eingebunden wird. Andernfalls wird kein Syntax-Higlighting aktiviert. Die korrekte Anbindung einer neuen Programmiersprache und des zugehörigen Sprachmodus-Skripts wird im Anhang B.2 beschrieben

Über die mit Ⓒ zusammengefassten Buttons „Reset“ und „Clear“ wird der CodeMirror-Editor via JavaScript[6] auf seinen initialen Wert (beim letzten Laden der Seite) zurückgesetzt oder komplett geleert. Bei der Bearbeitung von Quelltext eines geladenen Programms werden zudem ausgewählte Metainformationen des Programms in Informationsfeld Ⓕ dargestellt.

▷ Ausführung von Programmen

Wie im letzten Abschnitt erwähnt, wird die Ausführung des Quelltextes über den Button Ⓑ getriggert. Zu diesem Zweck verknüpft die View-Funktion `smapIE` den entsprechenden HTML-Ausdruck über den Handler `execHdlr` mit dem Controller `doExecuteProgram` aus Listing 4.16:

```
execHdlr env = next $ doExecuteProgram (execEnv,env systemRef,env codeRef,mProg)
```

Die Funktion `Controllers.next` ergänzt einen Controller zu dem Ergebnis eines HTML-Handlers (`IO HtmlForm`). Über die CGI-Umgebung `env` wird der Schlüssel des ausgewählten Ausführungssystems mithilfe der CGI-Referenz `systemRef` und der auszuführende Quelltext mithilfe der Referenz `codeRef` ausgelesen. Die Ausführungsumgebung `execEnv` und das (möglicherweise) geladene Programm `mProg` werden für die anschließende „Rückkehr“ zum Editor benötigt. Listing 4.18 zeigt die entsprechende Implementierung des Controllers `doExecuteProgram`:

```
1 doExecuteProgram :: (ExecEnv,String,String,Maybe Program) -> Controller
2 doExecuteProgram (execEnv@(_,systems),execSystemKey,code,mProg) =
3   checkAuthorization (smapIEOperation $ ExecuteProgram) $ \authzData ->
4   do execRes <- execute code $ getExecSystem execSystemKey systems
5      return $ smapIE mProg execEnv (Just execRes) code execSystemKey
6                      doExecuteProgram tryShowProgramCreationForm
7                      tryShowVersionCreationForm authzData
```

Listing 4.18: `controllers/SmapIEController` (Funktion `doExecuteProgram`)

Die eigentliche Ausführung wird in Zeile 4 über die Funktion `execute` angestoßen, die den auszuführenden Quelltext `code` und ein Ausführungssystem der entsprechenden Sprache erhält und ein Ergebnis vom Typ `ExecResult` erzeugt. Die konkrete `System`-Entität wird dazu von `getExecSystem` über den Schlüssel `execSystemKey` aus der Liste `systems` ausgelesen.

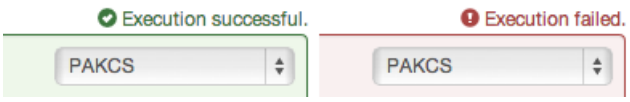


Abbildung 4.4.: Darstellung des Typs eines Ausführungsergebnisses

Liegt das Ausführungsergebnis `execRes` vor, wird es der View-Funktion `smapIE` übergeben (Zeile 5), welche es gemäß dem Entwurf in Textfeld Ⓔ darstellt. Dabei wird entsprechend

[6] Alle anwendungsspezfischen Skripte sind in `public/js/smap.js` zu finden.

Abbildung 4.4 sichtbar gemacht, ob die Ausführung erfolgreich war oder fehlgeschlagen ist.

Der Datentyp `ExecResult` und die Funktion `execute` bilden die Schnittstelle des Moduls `system/Execution`, welches die konkrete Implementierung der Ausführung kapselt. `ExecResult` repräsentiert dabei – wie bereits erwähnt – das Ergebnis einer Ausführung, welches stets aus einem Typ (Erfolg oder Fehlschlag) und der textuellen Rückgabe besteht:

```
data ExecResult = ExecSuccess String | ExecError String
```

Für gegebenen Quelltext und ein Ausführungssystem berechnet die Funktion `execute` ein solches Ergebnis als I/O-Aktion. Die Umsetzung der Ausführung basiert dabei auf dem im Entwurf skizzierten Verfahren (siehe 3.3.1):

```
1 execute :: String -> Maybe System -> IO ExecResult
2 execute code mSystem =
3   maybe (return $ ExecError noSystemFoundErr)
4         (\s -> do (exitCode,result) <- connectToCGI (systemExecUrl s) code
5                   execResult        <- getExecResult exitCode (header s++result)
6                   return execResult)
7         mSystem
8   where ... -- shortened (noSystemFoundErr, header string)
```

Listing 4.19: `system/Execution` (Funktion `execute`)

Die Funktion `connectToCGI`[7] sendet dazu den Quelltext `code` im Body eines HTTP-Post-Requests an die durch das Ausführungssystem bereitgestellte URL, unter der ein externes CGI-Skript (ein sogenannter Ausführungsdienst) liegt, welches die eigentliche Ausführung des Programms übernimmt und dessen Antwort das Ergebnis als Klartext enthält. Diese Antwort muss dabei folgendermaßen zusammengesetzt sein:

- Die erste Zeile des Rumpfs enthält *ausschließlich* die textuelle Repräsentation des vom Ausführungsprozess erzeugten Fehlercodes (`0` im Falle einer erfolgreichen Ausführung).
- Der Rest der Nachricht (ab der zweiten Zeile) enthält die textuelle Ausgabe der Ausführung (Berechnungsergebnisse, Fehlernachrichten, etc.).

Die Anwort wird anschließend von `connectToCGI` geparst und in ein entsprechendes Paar aus Fehlercode (`exitCode :: Int`) und Ergebnis (`result`) umgewandelt (Zeile 4). Der Fehlercode und die um einen Header ergänzte Nachricht werden daraufhin von `getExecResult` auf einen konkreten `ExecResult`-Wert gemappt (Zeile 5), welcher letztendlich als I/O-Aktion zurückgegeben wird (Zeile 6).

Der Vorteil der vorgenommenen Implementierung ist die flexible Handhabung der tatsächlichen Ausführung durch die Ausführungsdienste und deren leichte Austauschbarkeit. Unter Berücksichtigung der beschriebenen Schnittstelle können die zugrunde liegenden

[7] Die Funktion `connectToCGI` und ihre Implementierung enstammt dem Quellcode des PAKCS-WWW-Interface (`https://www-ps.informatik.uni-kiel.de/~pakcs/webpakcs/main.cgi`), welches von Michael Hanus entwickelt wurde und ebenfalls auf dem oben beschriebenen Konzept basiert. Für die Webanwendung Smap wurde die Implementierung für die Klassifizierung des Ausführungsergebnisses leicht angepasst.

CGI-Skripte beliebig umgesetzt werden. Insbesondere können zu deren Implementierung theoretisch beliebige Programmmiersprachen verwendet werden.[8]

▷ Speichern von Programmen und Versionen

Wie im Entwurf beschrieben ist der Button Ⓓ (mittels Bootstrap) als Dropdown-Menü realisiert und ermöglicht autorisierten Nutzern das Speichern neuer Programme oder das Anlegen neuer Versionen zu einem Programm. Das Anzeigen entsprechender Optionen wird dabei innerhalb der Funktion `smapIE` von der in Abschnitt 4.3.1 erwähnten Funktion `byAuthorization` übernommen, die auf Basis des jeweiligen Zugriffstyps und des `AuthZData`-Objekts Menüpunkte ein- oder ausblendet:

```
byAuthorization (smapIEOperation CreateProgram authzData)
  [li [class "divider"] [] -- access granted case
  ,li [] [linkSubmitBtn pcFormHdlr [saveIcon,text " Save program to Smap"]]]
  (\_ -> [empty])          -- access denied case
```

Im Falle einer erfolgreichen Autorisierung wird dem Dropdown-Menü in diesem Fall der Menüpunkt „`Save program to Smap`“ in Form eines Submit-Buttons hinzugefügt, welcher über den HTML-Handler `pcFormHdlr` mit dem Controller `tryShowProgramCreationForm` verknüpft ist.

`tryShowProgramCreationForm` ist – wie der Name vermuten lasst – für die Weiterleitung auf ein WUI-Formular zuständig, mit welchem der Quelltext nach Angabe der im Entwurf spezifizierten Metadaten als neues Programm gespeichert werden kann. Das Präfix `try` deutet hierbei an, dass diese Weiterleitung nicht immer erfolgreich ist. Zunächst testet der Controller nämlich nach dem im letzten Abschnitt beschriebenen Verfahren, ob der Quelltext mit dem gewählten Ausführungssystem erfolgreich ausführbar ist. Im Falle eines `ExecError`-Ergebnisses wird ein entsprechender `Alert` gesetzt und der Controller kehrt zur Weboberfläche des interaktiven Editors zurück. Bei erfolgreicher Ausführung erfolgt hingegen die Weiterleitung auf das durch die View-Funktion `SmapIEView.programCreationForm` generierte WUI-Formular. Dieses wird über die in Abschnitt 4.2.3 erwähnte Funktion `renderWuiForm` erzeugt und verwendet dazu das ebenfalls im Modul `SmapIEView` definierte zusammengesetzte Widget `wProgram`, welches entsprechende Formularelemente für die Eingabe des Titels, der Beschreibung, der Sichtbarkeit und einer Liste von Tags definiert. Die eigentliche Persistierung der Programm-Entität wird von dem Second-Level-Controller `doCreateProgram` aus dem Modul `ProgramsController` übernommen, welcher der View-Funktion `programCreationForm` vom Controller `tryShowProgramCreationForm` übergeben wird.

Analog wird über die Funktion `byAuthorization` im Falle eines geladenen Programms ggf. die „`Save as new version`“-Option angezeigt, die entsprechend mit dem Controller `tryShowVersionCreationForm` verknüpft ist und bei erfolgreicher Ausführung auf ein

[8] Zwei Beispiele für in Curry implementierte Dienste zur Ausführung von Curry-Programmen mit den Implementierungen PAKCS und KiCS2 sind im Verzeichnis `services/curry/` zu finden. Die korrekte Anbindung von Ausführungsdiensten und die Installation der beiden Beispiel-Skripte werden in den Anhängen B.3 und B.4 beschrieben.

WUI-Formular weiterleitet, mit dem der Quelltext nach Angabe einer Versionsnachricht als neue Version persistiert werden kann.

4.3.3. Browser

Analog zum Delegations-Controller der SmapIE-Komponente (siehe letzter Abschnitt) definiert der Delegations-Controller `BrowserController.browserController` folgende Routen für den Zugriff auf die Browser-Komponente:

(1) **`/browser`** Verweist auf das einleitend in Abschnitt 3.3.2 beschriebene Dashboard und lädt dessen Weboberfläche über den Controller `showDashboard`.

(2) **`/browser/tags`** Verweist auf die Tag-Übersicht und lädt deren Weboberfläche über den Controller `showTagList`.

(3) **`/browser/programs[?`*`query`*`]`** Zeigt im Allgemeinen die Listenansicht aller öffentlichen Programme, wobei über den optionalen Query String *`query`* gemäß dem Entwurf Suchparameter spezifiziert werden können, die eine Anpassung des Inhalts und der Sortierung der Ergebnisliste ermöglichen.

(4) **`/browser/myprograms[?`*`query`*`]`** Wie oben, wobei ausschließlich Programme des momentan authentifizierten Nutzers betrachtet werden (privat und öffentlich).

(5) **`/browser/myfavorites[?`*`query`*`]`** Wie oben, wobei ausschließlich die favorisierten Programme des momentan authentifizierten Nutzers betrachtet werden.

(6) **`/browser/`*`progKey`*`[/`*`versNum`*`]`** Zeigt die Einzelansicht des Programms mit der ID *`progKey`*. Bei expliziter Angabe einer Versionsnummer *`versNum`* über das optionale Pfad-Suffix wird dabei der Quelltext der entsprechenden Version dargestellt. Andernfalls wird die aktuellste Version geladen.

Wie schon im Entwurf der Browser-Komponente in Abschnitt 3.3.2 werden das Dashboard und die Tag-Übersicht auch im Rahmen der Implementierung nicht weiter behandelt, da sie für die Umsetzung der zentralen Anforderungen keine Relevanz haben. In den nachfolgenden Unterabschnitten werden deshalb ausschließlich die Funktionalitäten des Browsers erläutert, auf die über die letzten vier Routen zugegriffen werden kann. Dies sind die allgemeine Programm-Suche bzw. die Umsetzung mittels Query String spezifizierter komplexer Programm-Anfragen, die Darstellung der Ergebnisse in der Listenansicht und die Programm-Einzelansichten.

▷ Programm-Suche und Listenansicht

Die Routen 3 bis 5 repräsentieren Programm-Anfragen, deren Parameter über den Inhalt des Query Strings *`query`* bestimmt werden. Ursprünglich sah der Entwurf dabei nur im Falle der allgemeinen Programm-Suche (Route 3) die Spezifikation von Filter- und Sortierungseinstellungen über den Query String vor (siehe Abschnitt 3.3.2). Aufgrund der zweckmäßigen Darstellung der Ergebnisse aller drei Anfragen von einer zentralen Browser-Weboberfläche – der Listenansicht für Programme – und der simplen Adaptierbarkeit auf das Auslesen

aller Programme bzw. Favoriten eines Nutzers, wurde das Konzept jedoch auch für die letzten beiden Fälle übernommen. Für die Umsetzung komplexer Programm-Anfragen im Allgemeinen wurde in Abschnitt 4.2.1 das `ProgramQuery`-Konzept eingeführt, auf welchem auch die Implementierung der durch die Routen repräsentierten Anfragen basiert.
Der `browserController` delegiert die Verarbeitung der Routen 3 bis 5 dazu an die Controller `showProgramList`, `showUserProgramList` und `showUserFavoritesList`, welche als einziges Argument das `Url`-Objekt des Delegations-Controllers erhalten. Die drei Controller überprüfen nun zunächst über `checkAuthorization` die Zugriffsrechte auf die jeweilige Operation, wobei diese in den letzten beiden Fällen sinnigerweise nur dann erteilt werden, wenn der Nutzer authentifiziert ist. Aus dem dabei erzeugten `AuthZData`-Objekt extrahieren `showUserProgramList` und `showUserFavoritesList` den Namen des Nutzers (`userName`), welcher zur Konstruktion folgender Funktionen vom Typ `ProgramQuery -> IO [Program]` verwendet wird:

```
getAllProgramsWith . withIsVisibleOnly False
                   . withExactAuthorName userName       -- showUserProgramList
getAllProgramsWith . withAddExactFavoriterName userName -- showUserFavoritesList
```

Diese Funktionen repräsentieren nun exakt die Basis-URL-Pfade (bzw. die „Basis-Anfragen“) der Routen 4 und 5, da sie die durch das erwartete `ProgramQuery`-Objekt spezifizierte Suche derart einschränken, dass im ersten Fall ausschließlich private und öffentliche Programme des Nutzers und im zweiten Fall nur vom Nutzer favorisierte Programme zurückgegeben werden. Die *`query`*-Variable wird folglich von dem `ProgramQuery`-Argument repräsentiert, dessen Konstruktion aus den Parametern des Query Strings nun unabhängig von dem zugrunde liegenden Anfragetyp implementiert werden kann. Dazu werden die weiteren Steuerungsmaßnahmen von allen drei Controllern an den Controller `applySearchAndListPrograms` delegiert, welcher im Allgemeinen für die Durchführung von Programm-Anfragen und die Darstellung der zentralen Listenansicht zuständig ist:

```
1  applySearchAndListPrograms
2    :: Url
3    -> (ProgramQuery -> IO [Program])
4    -> ([Program] -> [Tag] -> Int -> SearchPanelData -> PagerData -> View)
5    -> Controller
6  applySearchAndListPrograms url@(_,qStr) getProgs programListView =
7    do (query,sets) <- getQueryAndSettingsFromQueryString qStr
8       results      <- getProgs query
9       ... -- shortened (computing searchPanelData, pagerData, ...)
10      return $ programListView pageResults popularTags totalResults
11                               searchPanelData pagerData
12   where ... -- shortened (local helper functions)
```

Listing 4.20: `controllers/BrowserController` (Fkt. `applySearchAndListPrograms`)

Als Argumente erhält `applySearchAndListPrograms` das `Url`-Objekt, dessen zweite Komponente die Parameter des Query Strings enthält (siehe Abschnitt 4.2.2), die Basis-Anfrage in Form einer der oben beschriebenen Funktionen (im Falle von `showProgramList` wird hier die Funktion `getAllProgramsWith` übergeben) und eine View-Funktion (siehe unten). Zunächst werden nun die im Query String `qStr` enthaltenen Suchparameter von der internen Funktion `getQueryAndSettingsFromQueryString` in das `ProgramQuery`-Objekt

query umgewandelt (Zeile 7). Dazu liest getQueryAndSettingsFromQueryString über das Modul Url die Werte der Query-String-Parameter über deren Bezeichner aus:

```
mQ       <- getStrValueFromQueryString "q"       qStr -- keyword
mTargets <- getStrValueFromQueryString "targets" qStr -- targets, e.g. "title,tags"
mLang    <- getStrValueFromQueryString "lang"    qStr -- lang. name, e.g. "curry"
mSort    <- getStrValueFromQueryString "sort"    qStr -- sorting, e.g. "created"
mOrder   <- getStrValueFromQueryString "order"   qStr -- ordering, "asc" or "desc"
```

Die für die komplexe Suche einstellbaren Parameter (q, targets, etc.) entsprechen dabei den im Entwurf vorgesehenen Einstellungsmöglichkeiten durch das Search Panel (siehe Abschnitt 3.3.2). Die ausgelesenen Werte werden (nach geeigneter Formatierung) über die vom Modul ProgramModel bereitgestellten Akzessoren aufbauend auf der defaultProgramQuery sukzessive in ein ProgramQuery-Objekt übersetzt:

```
  (maybe id withKeyword mQ)                                  -- set keyword
$ (withInfixIgnoreCaseKeywordInTitle titleInTs)              -- set title in targets
$ (withInfixIgnoreCaseKeywordInDescr descrInTs)              -- set descr. in targets
$ (withExactIgnoreCaseKeywordInTags  tagsInTs)               -- set tags int targets
$ (maybe id withExactIgnoreCaseImplLangName mLang)           -- set language name
$ (withSorting $ maybe defaultSort id $ getSorting mSort) -- set sorting
$ (withOrdering (getOrdering mOrder))                       -- set ordering
$ defaultProgramQuery                                       -- base query
```

titleInTs, descrInTs und tagsInTs sind hierbei vom Wert True, falls der entsprechende Bezeichner in der targets-Liste enthalten ist. getSorting bildet anhand einer internen Konstante programSortingOptions den Wert des sort-Parameters auf eine konkrete Sortierung vom Typ Sorting Program ab (siehe Abschnitt 4.2.1). getOrdering erzeugt analog einen Wert vom Typ OrderingType. Für nicht im Query String enthaltene Parameter (gekennzeichnet durch den Wert Nothing) werden über die Identitätsfunktion id und Konstanten wie defaultSort die Standardwerte der defaultProgramQuery beibehalten. Neben dem ProgramQuery-Objekt gibt getQueryAndSettingsFromQueryString auch ein Tupel vom Typ SearchSettings (sets, Zeile 7 in Listing 4.20) zurück, welches die aus dem Query String ausgelesenen Parameter enthält und später von der View (insbesondere vom Search Panel) zur Darstellung der Sucheinstellungen verwendet wird.
Nach der Erzeugung des ProgramQuery-Objekts wird nun durch die Anwendung der Basis-Anfrage getProgs die eigentliche Anfrage durchgeführt (Zeile 8). Anschließend werden weitere, für die Generierung der Weboberfläche durch die View-Funktion programListView benötigte Daten berechnet (Zeile 9). Dies sind:

- Die Programm-Liste pageResults, welche den von der momentanen Seite (ebenfalls ein Query-String-Parameter), der internen Konstante resultsPerPage und der Gesamtzahl an Ergebnissen abhängigen, darzustellenden Ausschnitt der Ergebnisliste results enthält.
- Die Tag-Liste popularTags, welche als zusätzliche Navigationshilfe unter dem Search Panel angezeigt wird (in Abbildung 4.5 nicht dargestellt).
- Die Gesamtanzahl an Ergebnissen (totalResults).
- Das Tupel searchPanelData :: SearchPanelData, welches alle zur Darstellung des Search Panels benötigten Informationen enthält (wie das SearchSettings-

Objekt `sets`).

- Und das `PagerData`-Objekt `pagerData`, welches wiederum alle zur Darstellung der Paginierungs-Fußleiste benötigten Daten enthält (vgl. Entwurf, Abbildung 3.5).

Auf all diese Daten wird im letzten Schritt die übergebene View-Funktion `programListView` angewendet (Zeile 10 und 11). Abhängig vom ursprünglich aufgerufenen Controller (`showProgramList`, `showUserProgramList` oder `showUserFavoritesList`) ist dies eine der View-Funktionen `programList`, `userProgramList` oder `userFavoritesList` aus dem Modul `BrowserView`, deren Zweck einzig darin besteht, der internen View-Funktion `renderProgramListView`, welche letztendlich in allen drei Fällen die zentrale Listenansicht rendert, zusätzlich routenspezifische View-Elemente zu übergeben (z. B. den Text des Headers; siehen unten). Abbildung 4.5 zeigt den wesentlichen Ausschnitt der für eine allgemeine Programm-Suche (Route 3) von `renderProgramListView` generierten Weboberfläche und die Umsetzung des Entwurfs aus Abbildung 3.5:

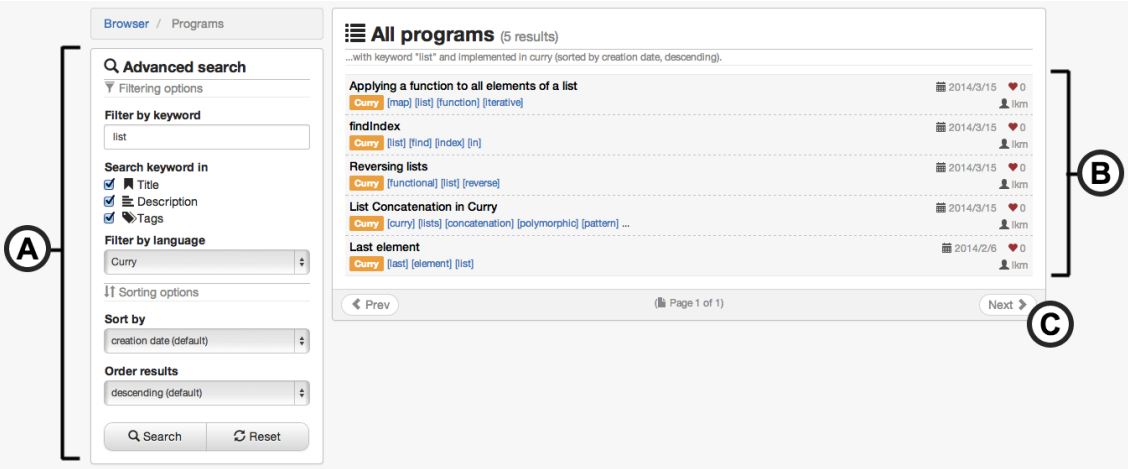


Abbildung 4.5.: Listenansicht im Browser (View-Bereich)

Das Search Panel (A) wird von der internen Funktion `renderSearchPanel` des Moduls `BrowserView` berechnet und visualisiert mithilfe des im Browser-Controller erzeugten `SearchSettings`-Tupel die Suchparameter des Query Strings. Das Klicken des „Search“-Buttons bewirkt wiederum das Starten einer neuen Suche mit den durch die Eingabeelemente spezifizierten Einstellungen. Dazu ist das HTML-Element des Buttons über eine ID mit einem jQuery-Handler verknüpft, der aus den Formulareingaben (Keyword, Targets, etc.) einen entsprechenden, für die Funktion `getQueryAndSettingsFromQueryString` verständlichen Query String konstruiert (siehe `/public/js/smap.js`). Das selbe gilt auch für das bereits im Entwurf beschriebene Quick-Search-Feld in der Navigationsleiste (siehe Abbildung 4.2, (A)). Für ein gegebenes Schlüsselwort *`keyword`* wird hier beispielsweise die URL `/browser/programs?q=`*`keyword`* generiert (bei keiner expliziten Angabe des `target`-Parameters wird das Schlüsselwort standardmäßig im Titel, in der Beschreibung und in den Tags gesucht).
Der Bereich (B) zeigt die visuelle Umsetzung der Anfrageergebnisse als Liste. Diese stellt für jedes enthaltene Programm den Titel (oben links), die Sprache (orangenes Label), die Tags (blaue Hyperlinks) und in der rechten Hälfte das Erstellungsdatum, den Autor und die

Anzahl verknüpfter `Favoriting`-Beziehungen dar. Über den vom Programm-Titel repräsentierten Hyperlink (Route 6, siehe nächster Unterabschnitt) kann zur Einzelansicht navigiert werden. Wie im Entwurf vorgesehen, wird zudem für das momentan mit dem Mauszeiger ausgewählte Listenelement ein Hyperlink angezeigt, über den das Programm direkt in den interaktiven Editor geladen wird (vgl. Abbildung 3.5). Weiterhin wird beim Klicken eines Tagnamen *tag* über die URL `/programs/browser?q=`*tag*`&targets=tags` eine Suche nach Programmen mit dem selben Tag initiiert. Die Berechnung der gesamten Programmliste erfolgt über die interne Funktion `renderProgramList`. Die Taglisten werden dabei analog von der Funktion `renderTagList` erzeugt.
Die Fußleiste © wird auf Basis des `PagerData`-Objekts gerendert und enthält die (ggf. über JavaScript deaktivierten) Paginierungselemente und Angaben zur momentan angezeigten Seite, sowie zur absoluten Seitenanzahl. Die momentane Seite wird dabei – wie bereits erwähnt – ebenfalls über einen Query-String-Parameter ermittelt.

▷ Einzelansicht von Programmen

Die Route 6 wird vom Delegations-Controller nach Validierung und Umwandlung der Programm-ID und der Versionsnummer durch die Funktion `validateKeyAndApply` (vgl. Listing 4.15 in Abschnitt 4.3.2) auf den Controller `showProgramPage` abgebildet. Falls die URL dabei keine explizite Versionsnummer vorgibt, wird dem Controller zunächst die Versionsnummer `0` übergeben, die immer auf die aktuellste Version eines Programms zeigt.

```
1  showProgramPage :: (ProgramKey,Int) -> Controller
2  showProgramPage (progKey,versNum) =
3    do mProg <- getProgramByKey progKey
4       maybe (showStdErrorPage programNotFoundErr) (\prog ->
5             checkAuthorization (browserOperation $ ShowProgram prog) $ \azData ->
6             do mValidVersNum <- getValidVersionNumber
7                                 (length $ programVersions prog)
8                maybe (showStdErrorPage $ versionNotFoundErr prog)
9                      (\validVersNum -> return $ programPage (prog,validVersNum)
10                                               doMakeVisibleCtrl doAddFavCtrl
11                                               doRemFavCtrl (doDeleteProgCtrl prog)
12                                               doCreateComCtrl azData)
13                     mValidVersNum)
14             mProg
15   where ... -- shortened (controller definitions, getValidVersionNumber, alerts, ...)
```

Listing 4.21: `controllers/BrowserController` (Funktion `showProgramPage`)

`showProgramPage` überprüft zunächst ob ein Programm für den gegebenen Schlüssel existiert (Zeile 4), ob der Nutzer ggf. für das Aufrufen der Einzelansicht dieses Programms autorisiert ist (Zeile 5) und ob die gegebene Versionsnummer valide ist (Zeile 6 bis 8). Wurde dem Controller die Versionsnummer `0` übergeben, wird diese zudem von der Funktion `getValidVersionNumber` auf die Nummer der aktuellsten Version abgebildet.
Wenn alle Überprüfungen erfolgreich sind, ruft `showProgramPage` in einem letzten Schritt die View-Funktion `programPage` auf und übergibt ihr das Programm, die valide Versionsnummer, eine Reihe von Controllern für die Verarbeitung der unterstützten Nutzeraktio-

nen und das `AuthZData`-Objekt (Zeile 9 bis 12). Unterstützte Nutzeraktionen sind dabei im Wesentlichen die durch den Entwurf beschriebenen Operationen, auf die über das Optionsmenü zugegriffen werden kann (Hinzufügen und Entfernen von den Favoriten, Löschen des Programms; siehe Abschnitt 3.3.2 und Abbildung 3.6, Ⓐ). Über den Controller `doMakeVisibleCtrl` können Nutzer zusätzlich ihre privaten Programme öffentlich machen. Alle der View-Funktion übergebenen Controller werden dabei über entsprechende Second-Level-Controller realisiert.

```
doAddFavCtrl = doAddFavoritingForCurrentUser
  (const $ showAccessDeniedErrorPage programNotFoundErr,Nothing) -- error case
  (showProgramPage (progKey,versNum)                   ,Nothing) -- success case
```

Der Controller `doAddFavCtrl :: Program -> Controller`, der ein gegebenes Programm zu den Favoriten des momentan authentifizierten Nutzers hinzufügt, wird bspw. lokal über den Second-Level-Controllers `doAddFavoritingForCurrentUser` aus dem Modul `UsersController` definiert.

Die View-Funktion `programPage` generiert nun für das gegebene Programm und die spezifizierte Version die Weboberfläche der Einzelansicht. Abbildung 4.6 zeigt beispielhaft die Einzelansicht des Programms aus Abbildung 4.3 und demonstriert die Umsetzung des Entwurfs aus Abbildung 3.6:

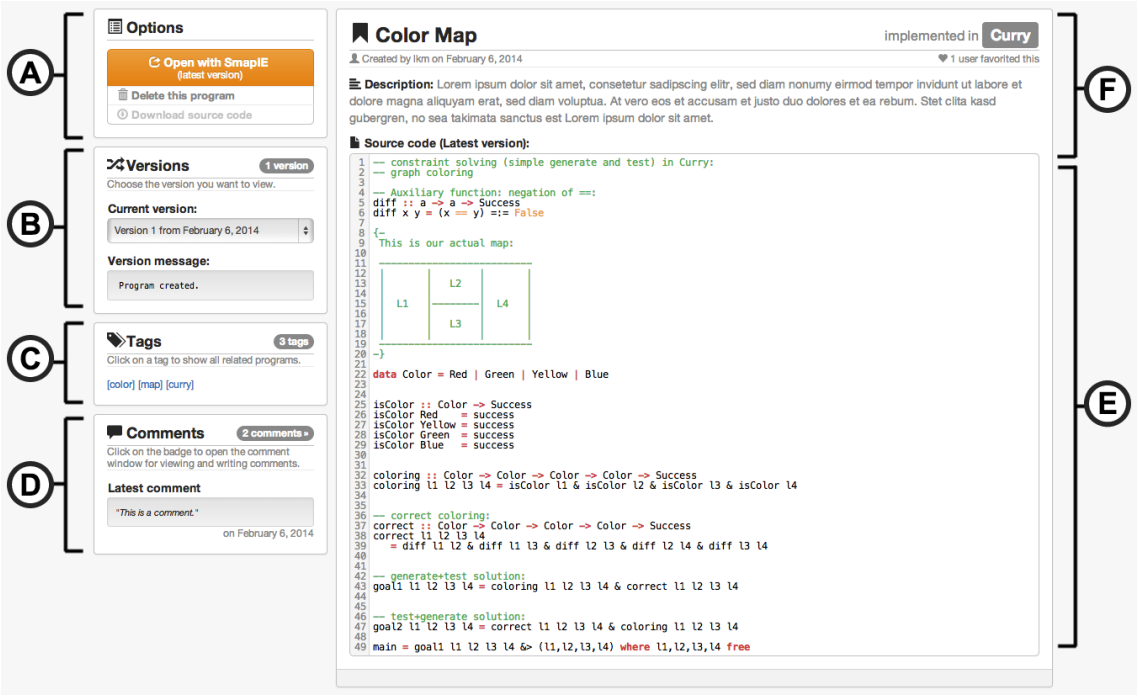


Abbildung 4.6.: Einzelansicht im Browser (View-Bereich)

Wie im Entwurf vorgesehen, enthält Panel Ⓐ alle zu dem betrachteten Programm verfügbaren Optionen. Dies umfasst stets mindestens den orangenen Button, mit dem das Programm über die entsprechende URL in den interaktiven Editor geladen wird. Alle weiteren Optionen unterhalb des Buttons werden (wie in Abschnitt 4.3.2 demonstriert) mithilfe

des `AuthZData`-Objekts und der Funktion `byAuthorization` ein- bzw. ausgeblendet. Im Beispiel aus Abbildung 3.6 ist der Nutzer bspw. zum Löschen des Programms autorisiert. Die Delete-Option stellt eine Besonderheit dar, da sie gemäß der Anforderungsspezifikation (siehe Abschnitt 3.1.1) vom Nutzer bestätigt werden muss. Zu diesem Zweck ist das entsprechende HTML-Element über die Funktion `Views.withConfirmation` mit einem Bestätigungsdialog verknüpft, der über Bootstraps Modal-Komponente[9] realisiert wurde:

```
(deleteOption,confirmDeleteDialog) = withConfirmation 2
  (linkLinkBtn "#" [deleteIcon,text " Delete this program"]) -- trigger element
  confirmDeleteMsg deleteHdlr
```

`withConfirmation` ergänzt ein gegebenes HTML-Element um die notwendigen Trigger-Eigenschaften (der Ausdruck aus Zeile 2 wird hier bspw. zu `deleteOption` ergänzt) und erzeugt aus einer ID (für den Fall mehrerer Modals im HTML-Dokument), der anzuzeigenden Nachricht (`confirmDeleteMsg`) und dem HTML-Handler, der die zu bestätigende Aktion durchführt (`deleteHdlr`) den HTML-Ausdruck für den Bestätigungsdialog. Trigger-Element und Bestätigungsdialog müssen anschließend nur noch an den gewünschten Stellen in die View eingefügt werden.
Über Panel Ⓑ erfolgt der Zugriff auf die Versionen des betrachteten Programms. Bei Auswahl einer Version über das Select-Menü wird die Seite dabei mit der entsprechenden URL über JavaScript/jQuery direkt neugeladen.
Auf die Liste der Tags des Programms kann – entsprechend dem Entwurf – über das Panel Ⓒ zugegriffen werden. Die Tagliste wird dabei wie in der Listenansicht über die Funktion `renderTagList` erzeugt, welche jedes Tag auf einen Hyperlink abbildet (siehe letzter Unterabschnitt).
Das Kommentar-Panel Ⓓ zeigt zunächst den letzten zu dem Programm abgegebenen Kommentar an. Über das Badge in der oberen rechten Ecke des Panels wird (ähnlich wie bei der Delete-Option) das Anzeigen eines Modals getriggert, welches alle bisher abgegebenen Kommentare auflistet und die Abgabe neuer Kommentare ermöglicht. Die Abgabe neuer Kommentare ist dabei nur authentifizierten Nutzern erlaubt. Für die Auflistung und Erstellung von Kommentaren muss die Einzelansicht des Programms also nicht verlassen werden.
Das Textfeld Ⓔ zeigt den Quelltext der momentan ausgewählten Version an. Wie zu erkennen ist, wurde die entsprechende `textarea` aus Lesbarkeitsgründen ebenfalls als CodeMirror-Editor realisiert (siehe Abschnitt 2.4 und 4.3.2). Analog zu Listing 4.17 wird die CodeMirror-Instanz dabei über die `fromTextArea`-Methode erzeugt. Im Gegensatz zur CodeMirror-Instanz des interaktiven Editors wird die Editierbarkeit des Quelltextes in der Einzelansicht allerdings durch die zusätzliche Option `readOnly: 'nocursor'` im Konfigurationsobjekt unterbunden.
Die restlichen relevanten Metainformationen werden schließlich im Kopfbereich Ⓕ dargestellt. Neben dem Titel, der Sprache, dem Erstellungsdatum, dem Datum der letzten Änderung und der Beliebtheit (in Form der verknüpften `Favoriting`-Beziehungen) wird hier auch die bei der Erstellung des Programms verfasste Beschreibung angezeigt.

[9] `http://getbootstrap.com/javascript/#modals`

4.3.4. Authentifizierung

Gemäß dem Entwurf aus Abschnitt 3.3.3 besteht die Schnittstelle der Authentifizierungs-Komponente im Wesentlichen aus den Weboberflächen für die Registrierung neuer Nutzer (Route: `/signup`), die Authentifizierung mittels Nutzername und Passwort (Route: `/signin`) und das Versenden eines neuen Passworts per E-Mail (Route: `/forgot`).

▷ Registrierung

Für das Anzeigen der Weboberfläche für die Registrierung ist im Allgemeinen der Controller `showSignUpPage` aus dem Modul `AuthNController` zuständig, welcher als Argument die initialen Eingaben des entsprechenden WUI-Formulars erwartet (Nutzername, E-Mail-Adresse, Passwort und bestätigtes Passwort). Beim Aufrufen über die Route `/signup` werden dabei sinnvollerweise zunächst leere Strings übergeben. Das Registrierungsformular selbst wird (wie die meisten WUI-Formulare der Anwendung) von der Funktion `Views.renderWuiForm` (siehe Abschnitt 4.2.3) generiert. Die zugehörige View-Funktion `AuthNView.signUpPage` erhält dabei vom Controller `showSignUpPage` den Second-Level-Controller `doCreateUser` zur Validierung der Eingaben und zur Persistierung der entsprechenden `User`-Entität. `doCreateUser` wird dazu in folgender Form übergeben:

```
doCreateUser
  (showSignUpPage       ,Just nameNotUniqueErrAlert ) -- name not unique case
  (showSignUpPage       ,Just emailNotUniqueErrAlert) -- email not unique case
  (showSignInPage . Just,Just signUpSucceededAlert  ) -- success case
```

Wie zu erkennen ist, wird der Nutzer im Falle eines bereits vergebenen Namens oder einer bereits vergebenen E-Mail-Adresse nach dem Setzen entsprechender `Alerts` auf das Registrierungsformular zurückgeleitet. Die eingegebenen Daten werden dabei an den Controller `showSignUpPage` übergeben (vgl. Signatur in Listing 4.11). Bei erfolgreicher Persistierung der `User`-Entität erfolgt hingegen eine Weiterleitung auf das Authentifizierungsformular (über den Controller `showSignInPage`; siehe nächster Unterabschnitt), auf dem sich der neu registrierte Nutzer einloggen kann.

▷ Einloggen und Ausloggen

Die Weboberfläche für die Authentifizierung eines Nutzers (repräsentiert von der View-Funktion `signInPage` und ebenfalls ein WUI-Formular) wird durch den im letzten Abschnitt bereits erwähnten Controller `showSignInPage` zurückgegeben, dem als Argument eine `User`-Entität übergeben werden kann, deren `Name`-Wert ggf. initial in das Formularfeld für den Nutzernamen eingesetzt wird. Für den eigentlichen Authentifizierungsprozess ist der Controller `doSignIn` zuständig:

```
1 doSignIn :: (String,String) -> Controller
2 doSignIn (userName,password) =
3   checkAuthorization (authNOperation SignIn) $ \_ ->
4   do userHash <- getUserHash userName password
5      mUser    <- getUserByNameWith userName (Nothing,Just userHash)
```

```
6       maybe (do setAlert signInFailedAlert
7                 return $ signInPage (Just userName) doSignIn)
8             (\user -> do signInToSession (userName,userIsAdmin user)
9                          return $ landingPage)
10            mUser
11    where ... -- shortened (signInFailedAlert)
```

Listing 4.22: `controllers/AuthNController` (Funktion `doSignIn`)

In Zeile 4 wird das gegebene Passwort dazu zunächst mit der Funktion `getUserHash` aus dem Modul `system/Authentication` verschlüsselt.[10] Mithilfe der `UserModel`-Funktion `getUserByNameWith` wird anschließend getestet, ob zu den gegebenen Authentifizierungsdaten ein passender Nutzer in der Datenbank existiert (Zeile 5 und 6). Ist dies der Fall, werden über die Funktion `signInToSession` entsprechende Sitzungsdaten angelegt und es erfolgt eine Weiterleitung zur Startseite der Anwendung (Zeile 8 und 9). Andernfalls wird eine entsprechende Warnmeldung angezeigt (Zeile 6 und 7).
Für die Verwaltung von Sitzungsdaten ist im Allgemeinen das bereits erwähnte Modul `Authentication` zuständig. Wie in Zeile 8 zu erkennen ist, bestehen Sitzungsdaten dabei aus dem Namen und dem `IsAdmin`-Attribut des Nutzers:

```
type AuthNData = (String,Bool)
```

Mithilfe von `signInToSession`, `signOutFromSession` und `getSessionAuthNData` können Sitzungsdaten nun angelegt, gelöscht und ausgelesen werden. Die Implementierung dieser Operationen basiert dabei auf dem von Spicey standardmäßig unterstützten Authentifizierungskonzept (siehe [HK12]).

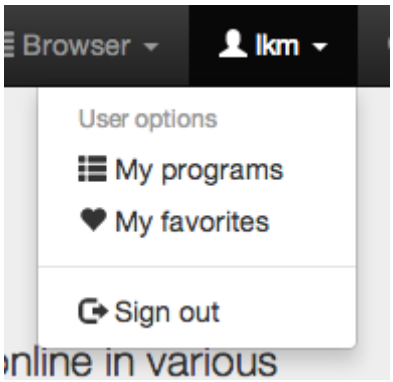


Abbildung 4.7.: Benutzermenü

Nach der Authentifizierung wird die „Sign in“-Option in der Navigationsleiste von der zuständigen Funktion `renderNavbar` aus dem Modul `SmapHtml` (siehe Abschnitt 4.2.3) durch das im Entwurf in Abschnitt 3.3.3 beschriebene Benutzermenü ersetzt, über welches zu den Listenansichten der eigenen Programme und der Favoriten navigiert werden kann (siehe Abbildung 4.7). Zusätzlich können sich Nutzer über das Menü auch wieder ausloggen. Analog zur Authentfizierung erfolgt das Ausloggen eines Nutzers dabei über den Controller `doSignOut`, welcher im Gegensatz zu `doSignIn` direkt über eine URL aufgerufen wird (Route `/signout`) und die beim Authentifizierungsprozess angelegten Sitzungsdaten über die Operation `signOutFromSession` wieder löscht.

▷ Senden eines neuen Passworts

Für das Senden eines neuen Passworts kann über die Route `/forgot` ein WUI-Formular aufgerufen werden, welches als Eingabe eine E-Mail-Adresse erwartet und beim Absenden den Controller `doSendNewPassword` aufruft. `doSendNewPassword` testet zunächst, ob für die gegebene E-Mail-Adresse ein Nutzer existiert. Ist dies der Fall, wird mit der

[10] Auf identische Weise wird das Passwort auch bei der Registrierung durch den Controller `doCreateUser` verschlüsselt. Der dabei entstehende Hash-Wert wird dann mit der `User`-Entität in der Datenbank abgelegt (vgl. Spezifikation der `User`-Entität in Abbildung 3.8).

Funktion `Authentication.randomPassword` ein neues, zufälliges Passwort erzeugt und mit `getUserHash` verschlüsselt. Danach wird das `Hash`-Attribut der zuvor ausgelesenen `User`-Entität in der Datenbank mit dem neuen Wert überschrieben. Abschließend sendet `doSendNewPassword` über das Modul `lib/Mail` (eine lokale Kopie des `Mail`-Moduls aus den Standardbibliotheken von PAKCS; siehe Anhang A.2 und C.3) eine Nachricht mit dem neuen Passwort an die anfangs eingegebene E-Mail-Adresse.

4.3.5. Administration

Wie in Abschnitt 3.3.4 beschrieben, stellt die Administrations-Komponente zunächst nur WUI-Formulare für das Hinzufügen von `Language`- und `System`-Entitäten zur Verfügung. Auf diese Formulare können Administratoren über das Benutzermenü oder über die Routen `/languages/new` bzw. `/systems/new` zugreifen.

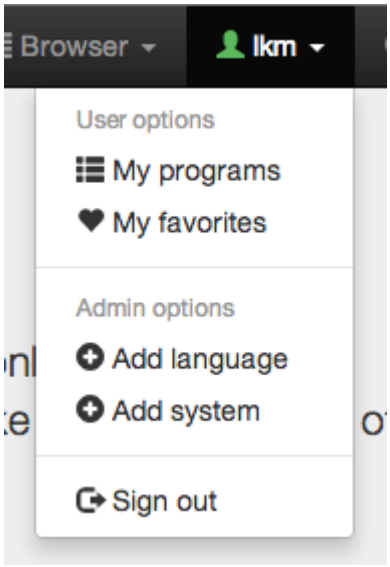


Abbildung 4.8.: Erweitertes Benutzermenü

Abbildung 4.8 zeigt das erweiterte Benutzermenü für Nutzer mit Administratorrechten (gekennzeichnet durch das grüne Symbol neben dem Nutzernamen). Die entsprechenden WUI-Formulare für die Erstellung neuer Programmiersprachen und Ausführungssysteme werden von den Controllern `showLanguageCreationForm` und `showSystemCreationForm` aus dem Controller-Modul `AdminController` geladen und von den beiden Views `languageCreationForm` und `systemCreationForm` aus dem Modul `AdminView` gerendert. `AdminView` definiert dazu ebenfalls die Widgets `wLanguage` und `wSystem`.

Für die vollständige Installation einer neuen Programmiersprache (inklusive der Unterstützung von Syntax-Highlighting durch die CodeMirror-Editoren) ist das bloße Erzeugen der entsprechenden `Language`-Entität nicht ausreichend. Zusätzlich muss vom Administrator noch das zugehörige Sprachmodus-Skript installiert werden (siehe Abschnitt 2.4 und 4.3.2). Die vollständige Anbindung einer neuen Programmiersprache, sowie das Anlegen eines Administratorkontos und die Installation von Ausführungssystemen werden im Rahmen einer Beschreibung der Anwendungsverwaltung durch Administratoren im Anhang B erläutert.

5. Fazit

Dieses Kapitel widmet sich abschließend der Zusammenfassung der erarbeiteten Ergebnisse und deren Bewertung unter Berücksichtigung der ursprünglichen Zielsetzung. Im letzten Abschnitt des Kapitels werden zudem Ideen für zukünftige Erweiterungen und Optimierungen des Software-Systems und von dessen Funktionsumfang diskutiert.

5.1. Zusammenfassung und Bewertung

Primäres Ziel war die Entwicklung einer webbasierten Anwendung, welche als zentrale Plattform für die Erstellung, Bearbeitung, Verwaltung, Bereitstellung und Ausführung von Programmen bzw. Quelltexten beliebiger Programmiersprachen im Browser dienen soll. Für die konkrete Entwicklung und Implementierung sollte dabei auf die experimentelle, funktional-logische Programmiersprache Curry bzw. das in Curry implementierte Webframework Spicey zurückgegriffen werden, um deren Praxistauglichkeit bei der Umsetzung nichttrivialer, komplexer Webanwendungen zu erproben.
Unter diesen Voraussetzungen wurde im Rahmen der vorliegenden Ausarbeitung die Webanwendung *Smap* vorgestellt und schrittweise entwickelt. Smap stellt mit dem interaktiven Web-Editor *SmapIE* und dem *Browser* als integrierte Verwaltungseinheit zwei unabhängige und erweiterbare Komponenten für die allgemeine Verarbeitung und Verwaltung von Programmen im Webbrowser zur Verfügung. Der interaktive Editor SmapIE ermöglicht dabei die effiziente und benutzerfreundliche Erstellung und Bearbeitung von Quelltext, das Abspeichern von Quelltext in Form von mit Metainformationen versehenen Programmen, sowie das Anlegen neuer Versionen zu bereits abgespeicherten Programmen. Dazu unterstützt SmapIE viele typische Funktionalitäten gängiger Quelltext-Editoren wie Syntax-Highlighting, automatische Code-Einrückung oder Brace Matching. Insbesondere erlaubt der interaktive Editor aber auch jederzeit die Ausführung von Quelltext einer bestimmten Programmiersprache unter Auswahl einer passenden Sprachimplementierung. Der Umfang an unterstützten Programmiersprachen und Sprachimplementierungen zur Erstellung und Ausführung von Programmen ist dabei beliebig erweiterbar. Über den Browser kann auf die von Benutzern der Anwendung erstellten Programme (inklusive aller mit ihnen verknüpften Informationen) mittels Suchfunktion, sowie Listen- und Einzelansichten zugegriffen werden. Der Browser ermöglicht das Laden von Programmen in den interaktiven Editor, das Löschen, Kommentieren und Favorisieren von Programmen, das Durchblättern ihrer Versionen, die Auflistung der selbst erstellten Programme und das Anzeigen der Favoritenliste. Für den Zugriff auf die von der Anwendung zur Verfügung gestellten Funktionen definiert Smap zudem ein auf der Authentifizierung basierendes Rollensystem, welches die Verwaltung durch Nutzer mit Administratorrechten vorsieht.

Zunächst ist festzustellen, dass sich die für die Implementierung der Anwendung verwendete Software durchweg als gute Wahl herausgestellt hat. So offenbarte sich bspw. der JavaScript-Editor CodeMirror mit seiner simplen Anbindbarkeit und der nahezu beliebigen Erweiterbarkeit durch Addons als ein leicht handhabbares, aber dennoch mächtiges Werkzeug bei der Implementierung des interaktiven Editors, welches Spielraum für zukünftige Anpassungen der Quelltexteingabe und -darstellung bietet und zudem durch das Konzept der Sprachmodus-Skripte gut mit dem erweiterbaren Umfang an unterstützten Programmiersprachen korreliert. Für die Konstruktion einheitlicher, schlichter, aber dennoch funktional umfangreicher Weboberflächen und die Wahrung einer schmalen Präsentationsschicht hat sich insbesondere auch der Einsatz des Front-End-Frameworks Bootstrap bezahlt gemacht, welches mit seinem großen Angebot an Gestaltungsvorlagen und wiederverwendbaren stilistischen Komponenten großen Einfluss auf den Entwurf der grafischen Benutzerschnittstellen hatte.
Die Verwendung der funktional-logischen Programmiersprache Curry war im Kontext der Webprogrammierung angesichts des deklarativen Programmierstils zunächst ungewohnt, stellte sich aufgrund des daraus folgenden hohen Abstraktionsgrads und der leichten Verständlichkeit und Prägnanz des Codes allerdings schnell als gut für die Umsetzung der Problemstellung geeignet heraus. Dies lag nicht zuletzt an dem Webframework Spicey, welches (abgesehen von anfänglichen technischen Schwierigkeiten) durch seinen simplen und strukturierten Aufbau und die Eigenschaft, aus einem ER-Diagramm eine initiale Anwendung mit grundlegenden Funktionalitäten generieren zu können, eine fokussierte und prägnante Umsetzung der wesentlichen funktionalen Anforderungen ermöglichte. Insbesondere der ER-basierte Charakter und die automatische Erzeugung der elementaren Datenbankoperationen erlauben in der Regel eine schnelle und flexible Adaption der Datenbankschnittstelle an Änderungen des Datenmodells, was der Skalierbarkeit der Anwendung zuträglich ist. Der durch Spicey erzeugte Code (einschließlich der standardmäßig unterstützten Implementierungen für Session Management, Authentifizierung und Autorisierung) erwies sich zudem insgesamt als gut erweiterbar und war ohne größere Schwierigkeiten an die Anforderungen einer „realen“ Anwendung anpassbar. Insbesondere das MVC-Architekturmuster bewährte sich hierbei als geeignetes Strukturierungsprinzip. Als eher negative Eigenschaft des Frameworks lässt sich die Art der generierten Datenbankschnittstelle nennen, die es bisher nicht erlaubt, komplexe Anfragen zu formulieren, die anschließend auf Datenbankebene (z. B. in Form einer äquivalenten SQL-Anfrage) ausgeführt werden. Stattdessen ist nur das Auslesen aller Einträge einer Tabelle und die anschließende Bearbeitung auf Anwendungsebene möglich, was in der Regel mit einem höheren Programmierungsaufwand und einer geringeren Performance verbunden ist. Aus diesem Grund wurde in Hinblick auf eine mögliche zukünftige Überarbeitung der Datenbankschnittstelle auch das `ProgramQuery`-Konzept eingeführt.

Bezüglich der entwickelten Webanwendung lässt sich zusammenfassend feststellen, dass die zentralen Anforderungen aus Abschnitt 3.1 durch das System und dessen Implementierung erfüllt werden. Die in der ursprünglichen Zielsetzung spezifizierten Funktionalitäten werden durch die funktionalen Komponenten der Webanwendung (SmapIE, Browser, Authentifizierung, Administration und Autorisierung) implementiert. Die Verwendung des Bootstrap-Frameworks gewährleistet ein einheitliches Layout und unterstützt zudem die intuitive Bedienbarkeit. Durch die vorgenommene Umsetzung des MVC-Architekturmusters ist zudem eine grundsätzliche Skalierbarkeit gegeben. Bezüglich der Unterstützung von

Programmiersprachen und Sprachimplementierungen ist der Funktionsumfang durch die Verwendung des flexiblen Webdienst-Konzepts zur Ausführung von Programmen beliebig erweiterbar. Selbstverständlich ist die entwickelte Anwendung aber noch nicht als fertiges Produkt anzusehen. Insbesondere befand sich das System noch nicht im intensiven realen Einsatz und birgt diesbezüglich noch viel Potential für notwendige Fehlerkorrekturen und Verbesserungen. Aber auch sonst gibt es viele sinnvolle Möglichkeiten zur Erweiterung des Funktionsumfangs der Anwendung oder zur Optimierung des bestehenden Systems. Im nächsten Abschnitt wird deshalb abschließend eine Auswahl an Ideen für zukünftige Erweiterungen und Optimierungen gegeben.

5.2. Ausblick

Die durch das entwickelte Software-System bereitgestellten Funktionalitäten entsprechen im Wesentlichen den spezifizierten Anforderungen, wobei diese im Laufe der Entwicklung bereits teilweise erweitert wurden (wie bspw. durch die Kommentarfunktion).
Eine weitere sinnvolle Erweiterung wäre z. B. die Ergänzung von Programmen (bzw. `Metadata`-Entitäten) um ein `IsEditable`-Flag, welches – wenn gesetzt – *beliebige* reguläre Nutzer autorisiert, neue Versionen zu einem Programm zu verfassen. Auf diese Weise könnten Autoren Nutzern erlauben, etwaige Verbesserungsvorschläge und Fehlerkorrekturen an ihren Programmen selbst umzusetzen. Bisher besteht für Nutzer in einem solchen Fall nur die Möglichkeit einen entsprechenden Kommentar zu verfassen, um den Autor oder einen Administrator zur Erstellung einer neuen Version anzuregen. Im Zuge dieser Erweiterung müsste sinnvollerweise auch das Datenmodell bzw. die `Authoring`-Beziehung derart modifiziert werden, dass zu jeder Version eines Programms ein Autor existiert, statt nur zu dem Programm (bzw. der `Metadata`-Entität) selbst.
Eine zweite Idee umfasst die Implementierung sogenannter *Sammlungen* (collections). Sammlungen sind im Wesentlichen definierte Teilmengen der selbst verfassten Programme, die für eine bestimmte Gruppe an Benutzern verfügbar (sichtbar) gemacht werden können. Dies würde beispielsweise die Bereitstellung von Programmen an Studenten durch Dozenten erleichtern (siehe Abschnitt 1.1) und eine grundsätzliche Organisation der eigenen Programme ermöglichen. Ein erster Schritt zur Umsetzung von Sammlungen wäre die Erweiterung des Datenmodells um einen `Collection`-Entitätstyp (z. B. mit `Title`- und `Description`-Attributen), welcher mit einer beliebigen Menge an Programmen (`Metadata`-Entitäten) und einer beliebigen Menge an Nutzern assoziiert wird. Derartige Sammlungen könnten dann bspw. durch eine Auswahl bereits gespeicherter Programme und Nutzer erstellt werden.
Sonstige Ideen für zukünftige Entwicklungen an dem bestehenden System umfassen die Implementierung einer Download-Funktion für Quelltexte aus der Einzelansicht bzw. aus dem interaktiven Editor (für die teilweise bereits eine entsprechende ausgegraute Option in der Webanwendung eingeblendet wird), die Umsetzung einer zusätzlichen funktionalen Komponente für die weiterführende Benutzerverwaltung (Definition von Benutzergruppen, öffentliche Benutzerprofile, direkter Zugang zu den Programmen eines beliebigen Nutzers, etc.), den Ausbau der Administrationskomponente (z. B. um die Möglichkeit, Nutzern über die Anwendung Administratorrechte zuweisen zu können), sowie den Ausbau der Hilfsseite, deren Grundgerüst bereits implementiert, aber noch nicht mit Inhalt gefüllt wurde.

Die hier genannten Ideen sind als kleine Teilmenge sinnvoller Erweiterungen und Optimierungen der ürsprunglichen Zielsetzung und der in Abschnitt 3.1 spezifizierten Anforderungen zu betrachten und wurden im Wesentlichen aufgrund der zeitlichen Beschränkung bei der Umsetzung des Projekts nicht mehr implementiert. Ihre Aufführung soll deshalb an dieser Stelle als Motivation und potentieller Einstiegspunkt für die zukünftige Weiterentwicklung der in dieser Ausarbeitung vorgestellten Webanwendung dienen.

A. Installation und Betrieb der Webanwendung

A.1. Voraussetzungen für Installation und Betrieb

Für die Installation und den Betrieb der Webanwendung Smap wird eine lauffähige Curry-Implementierung benötigt. Aus Kompatibilitätsgründen mit dem Spicey Framework wird dazu die PAKCS[1]-Distribution empfohlen. PAKCS ist momentan für Solaris, Linux und Mac OS X erhältlich und übersetzt Curry-Programme in äquivalente Prolog-Programme, weshalb auch das Vorhandensein einer Prolog-Installation vorausgesetzt wird (siehe `http://www.informatik.uni-kiel.de/~pakcs/INSTALL.html` für weitere Informationen zur Installation von PAKCS). Außerdem wird folgende Software benötigt:

- Die Versionsverwaltungssoftware Git[2] für das Kopieren des Quelltextes der Anwendung aus dem Git-Repository.
- Ein lokaler, CGI-fähiger Webserver für die Installation der Webanwendung als CGI-Skript (z. B. Apache[3]).
- Die SQLite3[4]-Bibliothek (und insbesondere das `sqlite3`-Kommando) für die korrekte Anbindung der standardmäßig verwendeten SQLite3-Datenbank.
- Ein Kommandozeilen-Tool zur Berechnung von Hashwerten für die Verschlüsselung von Passwörtern (z. B. `md5`[5] oder `sha1sum`[6]).
- Das `mailx`[7]-Kommando für das Versenden neuer Passwörter über die Authentifizierungskomponente.

A.2. Installationsanleitung

Die folgende Anleitung ist vor allem für die Installation von Testversionen der Webanwendung Smap gedacht. Für den realen Einsatz müssen ggf. noch Anpassungen vorgenommen werden (z. B. bezüglich des Pfades der SQLite3-Datenbank). Zudem zeigte sich, dass PAKCS

[1] `http://www.informatik.uni-kiel.de/~pakcs/`
[2] `http://git-scm.com/`
[3] `http://httpd.apache.org/`
[4] `https://sqlite.org/`
[5] `http://linux.die.net/man/1/md5`
[6] `http://linux.die.net/man/1/sha1sum`
[7] `http://linux.die.net/man/1/mailx`

im Vergleich zu KiCS2 eine signifikant geringere Performance beim Verwenden der Webanwendung aufweist. Nach Behebung der oben erwähnten Kompatibilitätsprobleme wird deshalb eine Neuinstallation unter KiCS2 empfohlen.

(1) Kopieren des Quelltextes aus dem Git-Repository

Im ersten Schritt wird zunächst das Git-Repository, welches den Quelltext der Anwendung enthält, in ein geeignetes Verzeichnis kopiert. Dies geschieht durch folgenden Befehl, wobei die entsprechenden Zugriffsrechte auf das Repository vorausgesetzt werden:

```
git clone git@git-ps.informatik.uni-kiel.de:theses/2013-lkm-ba.git
```

Hierbei wird insbesondere das Wurzelverzeichnis `Smap/` nebst Inhalt kopiert, in welches anschließend gewechselt wird:

```
cd 2013-lkm-ba/Smap/
```

Um alle Funktionalitäten nutzen zu können, müssen vor der eigentlichen Installation möglicherweise noch die in den nächsten zwei Schritten beschriebenen Anpassungen vorgenommen werden.

(2) Anpassung der Passwort-Verschlüsselung

Zur Aktivierung der Passwort-Verschlüsselung ist ggf. die Modifikation des verwendeten Kommandozeilen-Tools zu Berechnung der Hashwerte notwendig. Dieses wird durch die Funktion `getHash` im Modul `lib/Crypto` ausgewählt:

```
getHash = getHashWith "md5"
```

Standardmäßig wird der Message-Digest Algorithm 5 (über das Kommandozeilen-Tool `md5`) verwendet. Falls nötig, kann dieser Standardwert jederzeit über den der Funktion `getHashWith` übergebenen `String`-Wert geeignet modifiziert werden.

(3) Anpassung des `mailx`-Kommandos

Zur Aktivierung des E-Mail-Dienstes beim Anfordern eines neuen Passworts über die Authentifizierungskomponente ist unter Umständen die Modifikation des konkreten `mailx`-Kommandos zum Versenden von Nachrichten erforderlich. Dieses wird durch die Funktion `sendMailWithOptions` im Modul `lib/Mail` definiert:

```
execMailCmd ("mailx -r \""++from++"\" -s \""++subject++"\" "++
             (if null bccs then "" else "-b \""++bccs++"\" ")++
             (if null ccs  then "" else "-c \""++ccs++"\" ")++"\""++tos++"\"")
             contents
```

Abhängig von dem zugrunde liegenden System muss hier möglicherweise die Option `-r` durch die Option `-a` ersetzt werden.

(4) Installation der Webanwendung als CGI-Skript

Um die Webanwendung nun als CGI-Skript zu installieren, ist zunächst noch die Modifikation der im Shell-Skript `scripts/deploy.sh` definierten Variablen `WEBSERVERDIR` und `MAKECURRYPATH` notwendig. Erstere erhält dabei als Wert den Pfad zum Zielverzeichnis auf dem Webserver, in dem das CGI-Skript abgelegt werden soll. Letztere zeigt hingegen auf den Pfad zum PAKCS-Tool `makecurrycgi`, mit welchem das CGI-Skript kompiliert wird. Nach Anpassung der Variablen wird die Anwendung über den Befehl

```
make deploy
```

kompiliert und mitsamt aller notwendigen Dateien (z. B. dem Inhalt des `public/`-Ordners) in das Installationsverzeichnis auf dem Webserver kopiert. Die SQLite3-Datenbank `Smap.db` wird automatisch beim ersten Datenbankzugriff erzeugt und ebenfalls im Installationsverzeichnis abgelegt.

B. Verwaltung der Webanwendung

B.1. Anlegen eines Administratorkontos

Das Anlegen eines Administratorkontos bzw. das Zuweisen von Administratorrechten ist momentan nur Nutzern möglich, die direkten Zugriff auf die Datenbank haben. Im Falle einer Standardinstallation nach der in Abschnitt A.2 beschriebenen Vorgehensweise werden also Lese- und Schreibrechte für die SQLite3-Datenbankdatei `<installpath>/Smap.db` vorausgesetzt, wobei `<installpath>` der absolute Pfad zu dem in Installationsschritt **(4)** gewählten Installationsverzeichnis ist.

(1) Anlegen eines regulären Benutzerkontos

Für die Erstellung eines Administratorkontos muss zunächst ein reguläres Benutzerkonto angelegt werden, welches anschließend in Schritt **(2)** um Administratorrechte erweitert wird. Falls bereits ein entsprechendes Benutzerkonto existiert, kann dieser Schritt übersprungen werden.

Die Erstellung regulärer Benutzerkonten ist äquivalent zur Nutzerregistrierung. Die Navigation zum Registrierungsformular erfolgt über die Option „Sign up“ unterhalb des Authentifizierungsformulars auf der Anmeldeseite. Diese ist wiederum über den Menüpunkt „Sign in“ in der Navigationsleiste erreichbar. Die Registrierung erfordert die Angabe eines eindeutigen Nutzernamens, einer eindeutigen E-Mail-Adresse und eines Passworts. Nach erfolgreicher Registrierung wird die Anmeldeseite mit einer entsprechenden Hinweismeldung („Success! You can now sign in to Smap with your username and password!“) angezeigt.

(2) Zuweisung von Administratorrechten

Die Zuweisung von Administratorrechten erfolgt manuell über das Kommandozeilen-Tool `sqlite3`, welches Bestandteil der SQLite3-Bibliothek ist. Dazu wird zunächst die oben erwähnte Datenbankdatei `<installpath>/Smap.db` in die Shell geladen:

```
sqlite3 <installpath>/Smap.db
```

Durch das Setzen des `IsAdmin`-Flags der `User`-Entität wird der gewünschte Nutzer anschließend zum Administrator ernannt:

```
update User set IsAdmin='Prelude.True' where Name='"<userName>"';
```

Hierbei ist <userName> durch den in Schritt **(1)** gewählten Nutzernamen zu ersetzen.

Die Änderung wird auf Anwendungsebene nach der nächsten erfolgreichen Authentifizierung des entsprechenden Nutzers und der damit verbundenen Aktualisierung der Sitzungsdaten wirksam. Die erfolgreiche Zuweisung von Administratorrechten ist anhand des grünen Symbols neben dem Nutzernamen in der Navigationsleiste erkennbar (siehe Abbildung 4.8).

B.2. Installation einer neuen Programmiersprache

(1) Erzeugung einer neuen Sprache in der Datenbank

Für das Hinzufügen einer neuen Sprache in der Datenbank sind Administratorrechte notwendig. Authentifizierte Administratoren können über den Menüpunkt „Add language“ des erweiterten Benutzermenüs auf das entsprechende Formular navigieren (siehe Abbildung 4.8). Zur Erzeugung der neuen Sprache ist die Angabe eines Namens (z. B. „Curry“), der zugehörigen Dateiendung (für eventuelle Erweiterungen; z. B. „curry“) und einer Quelltext-Schablone (Source code template) erforderlich. Letztere wird beim Öffnen des leeren interaktiven Editors mit der entsprechenden Sprache als initialer Inhalt in das CodeMirror-Textfeld geladen und sollte verdeutlichen, welche grundsätzlichen Bausteine zur generellen Ausführung von Programmen der Sprache vorhanden sein müssen. Dabei handelt es sich in der Regel um das Gerüst einer `main`-Funktion. Nach erfolgreicher Erzeugung der Sprache wird die Startseite der Anwendung mit einer entsprechenden Hinweismeldung angezeigt.

(2) Installation des Sprachmodus-Skripts

Für die Aktivierung von Syntax-Highlighting durch CodeMirror muss zu jeder Sprache ein Sprachmodus-Skript installiert werden. Dazu wird das Skript <languagename>.js, welches die Syntax-Highlighting-Regeln der Sprache <LanguageName> spezifiziert, im Verzeichnis

```
<srcpath>/public/js/codemirror/modes/
```

abgelegt. Hierbei ist <scrpath> der absolute Pfad zum Wurzelverzeichnis `Smap/`, in dem sich der Quelltext der Anwendung befindet. Nach der Neuinstallation der Anwendung durch die Befehlsfolge

```
cd <srcpath>
make deploy
```

wird das Sprachmodus-Skript auch ins Installationsverzeichnis übernommen. Die Einbindung des Skripts in die Weboberflächen erfolgt bei korrekter Benennung automatisch durch die Anwendung.

Um Programme neu hinzugefügter Sprachen ausführen und speichern zu können, muss mindestens ein zugehöriges Ausführungssystem installiert werden (siehe nächster Abschnitt).

B.3. Installation eines neuen Ausführungssystems

(1) Konstruktion und Installation von Ausführungsdiensten

Ausführungsdienste werden in der Regel in Form von CGI-Skripten auf einem CGI-fähigen Webserver (z. B. Apache) installiert und können daher prinzipiell in einer beliebigen Programmiersprache verfasst werden. Für die Konstruktion von Ausführungsdiensten und für die Kommunikation mit der Webanwendung sind folgende Punkte zu beachten:

- Der Ausführungsdienst erhält den auszuführenden Code im Rumpf eines HTTP-Post-Requests und antwortet in Form einer Klartextnachricht, wobei der Inhalt der Antwort folgendermaßen zusammengesetzt sein muss:
  - Die erste Zeile des Rumpfs enthält *ausschließlich* die textuelle Repräsentation des vom Ausführungsprozess erzeugten Fehlercodes (`0` im Falle einer erfolgreichen Ausführung)
  - Der Rest der Nachricht (ab der zweiten Zeile) enthält die textuelle Ausgabe der Ausführung (Berechnungsergebnisse, Fehlernachrichten, etc.)
- Sinnigerweise sollte die Berechnungszeit des vom Ausführungsdienst gestarteten Prozesses, der die eigentliche Ausführung durchführt, durch ein Timeout begrenzt werden (z. B. durch das `timeout`-Kommando).
- Die Art der Ausführung sollte für alle Ausführungssysteme einer Programmiersprache identisch sein und zu der entsprechenden Quelltext-Schablone der Sprache passen (siehe letzter Abschnitt, Schritt **(1)**). Enthält die Quelltext-Schablone bspw. das Gerüst einer `main`-Funktion, sollten alle Ausführungssysteme diese als Einstiegspunkt für die Ausführung wählen.

Die Installation von Ausführungsdiensten hängt von ihrer Implementierung ab. Wie einleitend erwähnt, entspricht die Installation eines Ausführungsdienstes in der Regel der Installation des zugehörigen CGI-Skripts auf einem CGI-fähigen Webserver. Abschnitt B.4 demonstriert die Installation zweier in Curry implementierter Ausführungsdienste für die Ausführung von Curry-Programmen mit den Sprachimplementierungen PAKCS und KiCS2.

(2) Erzeugung eines neuen Ausführungssystems in der Datenbank

Für das Hinzufügen eines neuen Ausführungssystems in der Datenbank sind Administratorrechte notwendig. Authentifizierte Administratoren können über den Menüpunkt „Add system“ des erweiterten Benutzermenüs auf das entsprechende Formular navigieren (siehe Abbildung 4.8). Zur Erzeugung des neuen Ausführungssystems ist die Angabe eines Namens (z. B. „PAKCS“) und der URL des zugehörigen Ausführungsdienstes erforderlich. Die assoziierte Sprache wird über das Select-Menü ausgewählt. Nach erfolgreicher Erzeugung der Sprache wird die Startseite der Anwendung mit einer entsprechenden Hinweismeldung angezeigt.

B.4. Installation der Webdienste für die Ausführung mit PAKCS und KiCS2

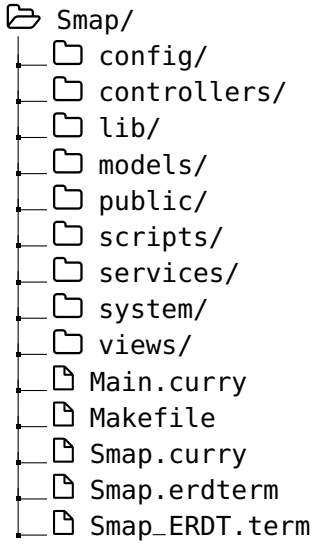
Das Verzeichnis `services/curry/` enthält mit `PAKCS.curry` und `KiCS2.curry` zwei Beispiele in Curry implementierter Webdienste für die Ausführung von Curry-Programmen. Für die Installation dieser Dienste wird eine lauffähige Curry-Implementierung (z. B. PAKCS) und ein CGI-fähiger Webserver (z. B. Apache) benötigt. Nach der Navigation in das entsprechende Verzeichnis können die Dienste durch folgende Befehlsfolge in lauffähige CGI-Skripte umgewandelt werden (hier am Beispiel des Dienstes `PAKCS.curry` mit der Curry-Implementierung PAKCS; die Installation des Dienstes `KiCS2.curry` verläuft analog):

```
pakcs :load PAKCS.curry :save :quit
mv PAKCS PAKCS.cgi
```

Das dabei erzeugte CGI-Skript `PAKCS.cgi` muss anschließend nur noch auf dem CGI-fähigen Webserver installiert werden und auf Anwendungsebene durch die Erzeugung eines neuen Ausführungssystems in der Datenbank unter Angabe der entsprechenden URL aktiviert werden (siehe letzter Abschnitt, Schritt **(2)**).

C. Beschreibung der Verzeichnisstruktur

In diesem Abschnitt wird die hierarchische Struktur und der Inhalt des nach Installationsschritt **(1)** aus Anhang A angelegten Verzeichnisses `Smap/` erläutert, welches den gesamten Quelltext der Webanwendung, sowie Beispiele für Ausführungsdienste zum Ausführen von Curry-Programmen und die Installations- und Kompilierungsskripte enthält. Die folgende Abbildung zeigt die oberste Ebene der Verzeichnisstruktur:



Es folgen Kurzbeschreibungen der einzelnen Verzeichnisse dieser Ebene und der in ihnen enthaltenen Module und Dateien.

C.1. Das `config/`-Verzeichnis

Das `config/`-Verzeichnis enthält die von Spicey während des Scaffolding-Prozesses erzeugten Module `RoutesData` und `ControllerMapping`, welche für die Verwaltung der unterstützten Routen und deren Mapping auf Controller-Funktionen zuständig sind.

`RoutesData.curry` Definiert Referenzen für alle Controller der Top-Level-Controller-Schicht (siehe Abbildung 4.1) und verknüpft die unterstützten Routen mit diesen Referenzen.

ControllerMapping.curry Bildet die im Modul `RoutesData` definierten Referenzen auf die Delegations-Controller der einzelnen Komponenten ab und wendet diese auf die momentane URL an (siehe Abschnitt 4.2.2).

C.2. Das `controllers/`-Verzeichnis

Das `controllers/`-Verzeichnis enthält alle in Abbildung 4.1 abgebildeten Module der Controller-Schicht (Second-Level-Controller und Top-Level-Controller; siehe Abschnitt 4.2.2).

CommentsController.curry Second-Level-Controller; Definiert beliebig einsetzbare, von der Präsentations- und der Top-Level-Controller-Schicht unabhängige Controller für die Erzeugung von `Comment`-Entitäten.

ExecEnvsController.curry Second-Level-Controller; Definiert beliebig einsetzbare, von der Präsentations- und der Top-Level-Controller-Schicht unabhängige Controller für die Erzeugung von `Language`- und `System`-Entitäten.

ProgramsController.curry Second-Level-Controller; Definiert beliebig einsetzbare, von der Präsentations- und der Top-Level-Controller-Schicht unabhängige Controller für die Erzeugung, Modifikation und Entfernung von `Program`-Entitäten und das Anlegen neuer Versionen.

UsersController.curry Second-Level-Controller; Definiert beliebig einsetzbare, von der Präsentations- und der Top-Level-Controller-Schicht unabhängige Controller für die Erzeugung und Modifikation von `User`-Entitäten und das Hinzufügen und Entfernen von `Favoriting`-Beziehungen.

SmapIEController.curry Top-Level-Controller; Implementiert die Steuerungseinheit der SmapIE-Komponente (siehe Abschnitte 3.3.1 und 4.3.2).

BrowserController.curry Top-Level-Controller; Implementiert die Steuerungseinheit der Browser-Komponente (siehe Abschnitte 3.3.2 und 4.3.3).

AuthNController.curry Top-Level-Controller; Implementiert die Steuerungseinheit der Authentifizierungs-Komponente (siehe Abschnitte 3.3.3 und 4.3.4).

AdminController.curry Top-Level-Controller; Implementiert die Steuerungseinheit der Administrations-Komponente (siehe Abschnitte 3.3.4 und 4.3.5).

StaticController.curry Top-Level-Controller; Implementiert Controller, die die statischen Weboberflächen der Anwendung generieren (Landing Page, Hilfsseite und About Page).

C.3. Das `lib/`-Verzeichnis

Das `lib/`-Verzeichnis enthält von der Anwendung unabhängige Module bzw. Bibliotheken. Dies umfasst auch Bibliotheken aus dem Standardumfang der Curry-Implementierungen

PAKCS und KiCS2, die für die Umsetzung der Anwendung modifiziert wurden.

- **`Bootstrap3.curry`** Definiert eine Schnittstelle für die Konstruktion von HTML-Dokumenten unter Verwendung des Front-End-Frameworks Bootstrap (siehe Abschnitt 2.3).
- **`Crypto.curry`** Von Spicey erzeugtes Modul, welches Hashing-Funktionen für die Verschlüsselung von Passwörtern bereitstellt.
- **`ERDGeneric.curry`** Von Spicey erzeugtes Modul, welches die Integritätstest der vom `erd2curry`-Tool generierten, sicheren Datenbankoperationen implementiert (siehe Abschnitt 4.1.2).
- **`HTML.curry`** Kopie der PAKCS-Bibliothek `HTML`, welches die Grundlage für die Webprogrammierung in Curry bildet (siehe Abschnitt 2.1.2). Exportiert zusätzlich den Konstruktor für CGI-Referenzen (`CgiRef`), um die Definition eigener Formularelemente zu ermöglichen (insbesondere im Modul `Html5`).
- **`Html5.curry`** Definiert alternative Konstruktoren für HTML5-Elemente, HTML-Attribute, Formularelemente und Operationen für die Verknüpfung von HTML-Elementen mit CSS-Klassen und IDs.
- **`KeyDatabase.curry`** Von Spicey erzeugtes Modul, welches die direkte Schnittstelle zur Datenbank implementiert (siehe Abschnitt 4.1.2). Diese wurde im Rahmen der Umsetzung der Webanwendung derart modifiziert, dass der interne Schlüssel von Entitäten als explizites „`integer primary key autoincrement`“-Attribut realisiert wird (und nicht über die `rowid`; siehe dazu `http://sqlite.org/autoinc.html`).
- **`Mail.curry`** Kopie der PAKCS-Bibliothek `Mail`, welche Funktionen für das Senden von Mails über das `mailx`-Kommando bereitstellt. Muss bei der Installation ggf. an das zugrunde liegende System angepasst werden (siehe ...).
- **`WUI.curry`** Kopie der PAKCS-Bibliothek `WUI`, welches die Konstruktion von typsicheren Web User Interfaces (WUIs) ermöglicht (siehe Abschnitt 2.1.2). Das im Verzeichnis `lib/` enthaltene Modul erweitert Widgets um `ErrorRendering`s, welche im Falle einer fehlerhaften Eingabe auf das Formularelement angewendet werden (standardmäßig werden nur Fehlernachrichten unterstützt). Die Standard-Schnittstelle ist davon nicht betroffen.

C.4. Das `models/`-Verzeichnis

Das `models/`-Verzeichnis enthält alle in Abbildung 4.1 abgebildeten Module der Model-Schicht (siehe Abschnitt 4.2.1).

- **`CommentModel.curry`** Implementiert die Datenbankschnittstelle zur Erzeugung und Verarbeitung von `Comment`-Entitäten.
- **`ExecEnvModel.curry`** Implementiert die Datenbankschnittstelle zur Erzeugung und Verarbeitung von `ExecEnv`-, `Language` und `System`-Entitäten.

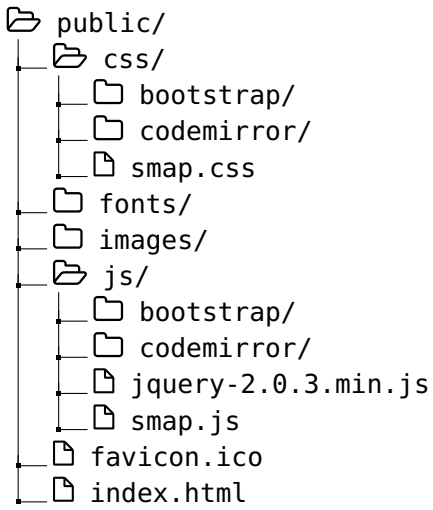
`ProgramModel.curry` Implementiert die Datenbankschnittstelle zur Erzeugung und Verarbeitung von `Program`-Objekten. Ermöglicht zudem die Konstruktion von komplexen Programm-Anfragen in Form von `ProgramQuery`-Objekten (siehe Abschnitt 4.2.1).

`TagModel.curry` Implementiert die Datenbankschnittstelle zur Erzeugung und Verarbeitung von `Tag`-Entitäten.

`UserModel.curry` Implementiert die Datenbankschnittstelle zur Erzeugung und Verarbeitung von `User`-Entitäten.

C.5. Das `public/`-Verzeichnis

Das `public/`-Verzeichnis enthält alle Dateien, die von einer Installation der Webanwendung verwendet werden. Dies umfasst beispielsweise CSS-Stylesheets, Schriftarten, Bilddateien und JavaScript-Skripte. Bei der Installation der Webanwendung wird auch der gesamte Inhalt des `public/`-Verzeichnisses zum Zielpfad kopiert. Die folgende Abbildung zeigt die wesentlichen Bestandteile der obersten zwei Ebenen der Verzeichnisstruktur innerhalb des `public/`-Ordners:



`css/` Enthält die von der Webanwendung verwendeten CSS-Stylesheets:

- **`bootstrap/`** Enthält die vom Front-End-Framework Bootstrap zur Verfügung gestellten Stylesheets (siehe Abschnitt 2.3).
- **`codemirror/`** Enthält die Stylesheets `codemirror.css` und `smap.css` (siehe Abschnitt 2.4.2), wobei `smap.css` das standardmäßig verwendete Theme für alle CodeMirror-Instanzen ist.
- **`smap.css/`** Das allgemeine Stylesheet der Webanwendung Smap.

`fonts/` Enthält die von Bootstrap für die Darstellung von Icons verwendeten Glyphicons[1]-Schriftarten.

[1] `http://glyphicons.com/`

images/ Enthält die in den Weboberflächen verwendeten Bilddateien.

js/ Enthält die von der Webanwendung verwendeten JavaScript-Skripte:

- **bootstrap/** Enthält die von Bootstrap bereitgestellten Skripte zur Nutzung der JavaScript-Features (siehe Abschnitt 2.3.2).
- **codemirror/** Enthält alle für die Verwendung des CodeMirror-Editors benötigten JavaScript-Skripte. Dies umfasst das in Abschnitt 2.4.2 erwähnte Skript `codemirror.js`, welches die Grundfunktionalität implementiert, sowie alle verwendeten Addon- und Sprach-Modus-Skripte.
- **jquery-2.0.3.min.js** Das von Bootstrap verwendete jQuery-Framework (siehe Abschnitt 2.3.2).
- **smap.js** Das allgemeine `js`-Skript der Webanwendung Smap, welches beispielsweise die Konstruktion von Query Strings für Programm-Suchen und die Funktionalität der SmapIE-Buttons implementiert.

favicon.ico Das Favicon der Webanwendung Smap.

index.html Von Spicey erzeugtes HTML-Dokument, welches Nutzer zur Webanwendung (also zum CGI-Skript `smap.cgi`) weiterleitet.

C.6. Das `scripts/`-Verzeichnis

Das `scripts/`-Verzeichnis enthält die von Spicey generierten Skripte zur Installation und Kompilierung der Webanwendung.

C.7. Das `services/`-Verzeichnis

Das `services/`-Verzeichnis enthält Beispiel-Implementierungen für Web Services zur Ausführung von Programmen im interaktiven Editor SmapIE (siehe Abschnitt 4.3.2). Aktuell werden dabei folgende Sprache unterstützt:

```
services/
└── curry/
    ├── KiCS2.curry
    └── PAKCS.curry
```

curry/ Enthält Web Services für die Programmiersprache Curry (siehe Abschnitt 2.1):

- **KiCS2.curry** In Curry implementierter Web Service, der Curry-Programme mit der Curry-Implementierung PAKCS ausführt (für die Installation des Web Services siehe Abschnitt B.3)
- **PAKCS.curry** In Curry implementierter Web Service, der Curry-Programme mit der Curry-Implementierung KiCS2 ausführt (für die Installation des Web Services

siehe Abschnitt B.3)

C.8. Das `system/`-Verzeichnis

Das `system/`-Verzeichnis enthält die globalen Module der System-Schicht (siehe Abbildung 4.1). Dies sind Module, die im Allgemeinen von mehreren Schichten des MVC-Architekturmusters importiert werden.

- **`Alerts.curry`** Implementiert die auf Weboberflächen einblendbaren Warn- und Hinweismeldungen (siehe Abschnitt 4.2.2).
- **`Authentication.curry`** Implementiert alle für die Authentifizierung benötigten Datentypen und Funktionen. Dies sind insbesondere die Operationen zur Verwaltung der Sitzungsdaten (siehe Abschnitt 4.3.4).
- **`Authorization.curry`** Definiert die in Abschnitt 4.3.1 beschriebenen Datentypen und Funktionen für die grundlegende Implementierung der Autorisierung.
- **`AuthorizedOperations.curry`** Definiert Zugriffstypen und Nutzeroperationen für alle funktionalen Komponenten und spezifiziert die Zugriffsrechte auf diese Operationen unter Berücksichtigung des Nutzer-Typs (siehe Abschnitt 4.3.1).
- **`Controllers.curry`** Exportiert den Typ `Controller`, Controller für die Darstellung von Fehlerseiten, allgemeine Operationen für das Ausführen von Controllern und die Funktion `getForm` (siehe Abschnitt 4.2.2).
- **`Execution.curry`** Implementiert die Schnittstelle für die Anbindung von Ausführungsdiensten und die Ausführung von Programmen (siehe Abschnitt 4.3.2).
- **`Models.curry`** Definiert allgemeine Typen und Operationen für die Implementierung von komplexen Programm-Anfragen (siehe Abschnitt 4.2.1).
- **`Routes.curry`** Exportiert die Funktion `getControllerReference`, welche den aktuellen URL-Pfad auf eine Controller-Referenz aus dem Modul `RoutesData` abbildet, falls eine entsprechende Regel existiert.
- **`Session.curry`** Von Spicey generiertes Modul, welches die allgemeine Verwaltung von Sessions implementiert (Session-IDs, Session-Cookies, etc.).
- **`SmapHtml.curry`** Enthält anwendungsspezifische HTML-Komponenten, Bestandteile des Rahmenlayouts, Buttons und Icons (siehe Abschnitt 4.2.3).
- **`SmapWui.curry`** Enthält anwendungsspezifische WUI-Komponenten für die Zusammensetzung komplexer Widgets auf Basis des Bootstrap-Frameworks (siehe Abschnitt 4.2.3).
- **`Url.curry`** Definiert allgemeine Operationen für die Verarbeitung von URLs und die Extraktion von Werten aus dem Query String (siehe Abschnitt 4.2.2).
- **`Views.curry`** Exportiert den Typ `View` und implementiert allgemeine, von den Views verwendete Operationen und Funktionen für die Konstruktion von WUI-Formularen

(siehe Abschnitt 4.2.3).

C.9. Das `views/`-Verzeichnis

Das `views/`-Verzechnis enthält alle in Abbildung 4.1 abgebildeten Module der View-Schicht (siehe Abschnitt 4.2.3).

- **`SmapIEView.curry`** Exportiert alle Views der SmapIE-Komponente (siehe Abschnitte 3.3.1 und 4.3.2).
- **`BrowserView.curry`** Exportiert alle Views der Browser-Komponente (siehe Abschnitte 3.3.2 und 4.3.3).
- **`AuthNView.curry`** Exportiert alle Views der Authentifizierungs-Komponente (siehe Abschnitte 3.3.3 und 4.3.4).
- **`AdminView.curry`** Exportiert alle Views der Administrations-Komponente (siehe Abschnitte 3.3.4 und 4.3.5).
- **`StaticView.curry`** Exportiert alle Views für die statischen Weboberflächen der Anwendung (Landing Page, Hilfsseite und About Page).

C.10. Sonstige Module und Dateien

- **`Main.curry`** Das in Abschnitt 2.2.3 erwähnte Modul `Main`, welches die `main`-Funktion und somit den Einstiegspunkt der Webanwendung definiert.
- **`Makefile`** Enthält die Regeln für die Installation und Kompilierung der Anwendung.
- **`Smap.curry`** Enthält die von Spicey aus der ER-Spezifikation erzeugten Datentypen und Datenbankoperationen (siehe Abschnitt 4.1.2).
- **`Smap.erdterm`** Enthält den ER-Term aus dem die Datenbankschnittstelle des Moduls `Smap.curry` generiert wurde (siehe Abschnitt 4.1.1).
- **`Smap_ERDT.term`** Enthält den erweiterten, durch das `erd2curry`-Tool erzeugten ER-Term (siehe Abschnitt 4.1.2).

Literaturverzeichnis

[Abr76] ABRIAL, Jean-Raymond: Data Semantics. In: *IFIP Working Conference Data Base Management* (1976), S. 1–60

[BHM08] BRASSEL, Bernd ; HANUS, Michael ; MÜLLER, Marion: High-Level Database Programming in Curry. In: *Proc. of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, 2008, S. 316–332

[Che76] CHEN, Peter Pin-Shan: The Entity-Relationship Model Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), S. 9–36

[Han01] HANUS, Michael: High-Level Server Side Web Scripting in Curry. In: *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, 2001, S. 76–92

[Han06] HANUS, Michael: Type-Oriented Construction of Web User Interfaces. In: *Proc. of the 8th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'06)*, 2006, S. 27–38

[Han12] HANUS (ED.), MICHAEL: *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*. `http://www.curry-language.org`, 2012. – Letzter Abruf: 07.03.2014

[Han14] HANUS (ED.), MICHAEL: *PAKCS 1.11.3 — The Portland Aachen Kiel Curry System*. `http://www.informatik.uni-kiel.de/~pakcs/Manual.pdf`, 2014. – Letzter Abruf: 07.03.2014

[Hav07] HAVERBEKE, Marijn: *Implementing a Syntax-Highlighting JavaScript Editor — In JavaScript*. `http://codemirror.net/1/story.html`, 2007. – Letzter Abruf: 26.02.2014

[HK12] HANUS, Michael ; KOSCHNICKE, Sven: An ER-based Framework for Declarative Web Programming. In: *To appear in Theory and Practice of Logic Programming (TPLP)* (2012)

[Kos08] KOSCHNICKE, Sven: Entwicklung von Web-Applikationen aus deklarativen Beschreibungen (Master Thesis), 2008

[KP88] KRASNER, Glenn E. ; POPE, Stephen T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: *Journal of Object-Oriented Program.* 1 (1988), August, Nr. 3, S. 26–49

[Mar83] MARTIN, James: Managing the Data-base Environment. (1983), S. 381

[Mar11] MARCOTTE, Ethan: *Responsive Webdesign*. New York, USA : A Book Apart, 2011

[Net04] NETWORK WORKING GROUP: *The Common Gateway Interface (CGI) Version 1.1*.

http://tools.ietf.org/html/rfc3875, 2004. – Letzter Abruf: 06.03.2014

[Ott11] OTTO, Mark: *Bootstrap from Twitter*. https://blog.twitter.com/2011/bootstrap-twitter, 2011. – Letzter Abruf: 25.02.2014

[W3C11] W3C — WORLD WIDE WEB CONSORTIUM: *Cascading Style Sheets (CSS) Snapshot 2010*. http://www.w3.org/TR/CSS/, 2011. – Letzter Abruf: 26.02.2014

[W3C14] W3C — WORLD WIDE WEB CONSORTIUM: *HTML5 — A vocabulary and associated APIs for HTML and XHTML*. http://www.w3.org/TR/html5/, 2014. – Letzter Abruf: 26.02.2014