

**HTL-LEONDING**  
**ABTEILUNG EDV UND ORGANISATION**

DIPLOMARBEIT

---

**James**

---

*Autoren:*

Florian LEIMGRUBER  
Michael MOSER

*Betreuer:*

Dipl. Ing. Prof. Gerald KÖCK

8. April 2016

## Eidesstattliche Erklärung

Wir erklären an Eides statt, dass wir die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht haben.

Florian Leimgruber

Michael Moser

Leonding, am 8. April 2016

## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>6</b>
<b>2 Ist-Situation</b>	<b>8</b>
<b>3 Lösungsansätze</b>	<b>11</b>
<b>4 Technische Beschreibung</b>	<b>29</b>
<b>5 Evaluation des Projektverlaufs</b>	<b>76</b>
<b>6 Quellen-, Literatur- und Abbildungsverzeichnis</b>	<b>78</b>

# Detailliertes Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>6</b>
1.1 Kurzbeschreibung [M] . . . . .	6
1.2 Abstract [L] . . . . .	6
1.3 Aufgabenstellung [M] . . . . .	7
1.4 Zielsetzung [M] . . . . .	7
1.5 Geplantes Ergebnis [M] . . . . .	7
<b>2 Ist-Situation</b>	<b>8</b>
2.1 Spotlight [L] . . . . .	8
2.2 Alfred [L] . . . . .	8
2.3 Wox [M] . . . . .	10
<b>3 Lösungsansätze</b>	<b>11</b>
3.1 Suche [L] . . . . .	11
3.1.1 Datenbank [L] . . . . .	11
3.1.2 Spotlight [L] . . . . .	12
3.1.3 Lucene [M] . . . . .	12
3.1.4 Eigene Implementierung [L] . . . . .	13
3.2 Auswahl der UI Technologie [M] . . . . .	13
3.2.1 OS X [L] . . . . .	14
3.2.2 Windows [M] . . . . .	14
3.3 Personalisierung der Suchergebnisse [L] . . . . .	15
3.4 Workflows [M] . . . . .	18
3.4.1 Interaktiver Workfloweditor [M] . . . . .	19
3.4.2 Workflow-Komponenten [M] . . . . .	22
3.4.2.1 Triggers [M] . . . . .	22
3.4.2.2 Actions [M] . . . . .	23
3.4.2.3 Outputs [M] . . . . .	24
3.5 James Downloadcenter [M] . . . . .	25
3.5.1 PHP 7 [M] . . . . .	26
3.5.2 ASP.NET 4.6 [M] . . . . .	26
3.5.3 ASP.NET CORE 1.0 [M] . . . . .	27
<b>4 Technische Beschreibung</b>	<b>29</b>
4.1 Suchalgorithmus [L] . . . . .	29
4.1.1 Radix Tree [L] . . . . .	30
4.1.1.1 Vergleich Trie [L] . . . . .	31
4.1.1.2 Transformation zu einem binären Baum [L] . . . . .	32
4.1.1.3 Suche [L] . . . . .	33
4.1.1.4 Einfügen [L] . . . . .	34
4.1.1.5 Löschen [L] . . . . .	35
4.1.2 Infix-Matching [L] . . . . .	35

4.1.3	Priorität [L] . . . . .	36
4.1.3.1	Add Priority [L] . . . . .	38
4.1.4	Rename [L] . . . . .	38
4.1.5	Save [L] . . . . .	39
4.1.6	Load [L] . . . . .	39
4.1.7	Implementierung [L] . . . . .	39
4.1.7.1	Umlaute [L] . . . . .	40
4.2	Suchengine-Wrapper [L] . . . . .	42
4.2.1	Implementierung unter Windows [M] . . . . .	42
4.2.2	Implementierung unter OS X [L] . . . . .	44
4.3	File Watcher [L] . . . . .	46
4.3.1	Cocoa [L] . . . . .	46
4.3.1.1	Rename [L] . . . . .	47
4.3.2	.Net Framework [M] . . . . .	48
4.4	Swift, Cocoa [L] . . . . .	49
4.4.1	Programmiersprache [L] . . . . .	49
4.4.2	Framework [L] . . . . .	53
4.4.3	Quick Look [L] . . . . .	55
4.5	WPF und .NET Framework [M] . . . . .	59
4.5.1	Neue Technologien [M] . . . . .	59
4.5.2	Trennung vom Code und Layout [M] . . . . .	60
4.5.3	MVC [M] . . . . .	61
4.5.4	MahApps [M] . . . . .	62
4.5.5	Aufgetauchte Problematiken [M] . . . . .	63
4.6	ASP.NET Core 1.0 [M] . . . . .	66
4.6.1	Namensgebung [M] . . . . .	69
4.7	Workflows [M] . . . . .	69
4.7.1	Persistierung [M] . . . . .	69
4.7.2	Importierung von Workflows [M] . . . . .	70
4.7.2.1	Implementierung Windows [M] . . . . .	70
4.7.2.2	Implementierung OS X [L] . . . . .	70
4.7.3	UI über Reflection [M] . . . . .	71
4.7.3.1	Windows [M] . . . . .	71
4.7.3.2	OS X [L] . . . . .	72
4.7.4	Custom-Url-Protocol-Hanlder bzw. Url Scheme [M] . . . . .	73
4.7.4.1	Windows - Custom-Url-Protocol-Handler [M] . . . . .	73
4.7.4.2	OS X - Url Scheme [L] . . . . .	74
4.7.5	Format String [L] . . . . .	74
4.7.6	Komponenten im Hintergrund ausführen [M] . . . . .	75
<b>5</b>	<b>Evaluation des Projektverlaufs</b>	<b>76</b>
5.1	Meilensteine [M] . . . . .	76
5.2	Gelerntes [L] . . . . .	76
5.3	Was würden wir anders machen? [L] . . . . .	76

<b>6 Quellen-, Literatur- und Abbildungsverzeichnis</b>	<b>78</b>
6.1 Quellen- und Literaturverzeichnis . . . . .	78
6.2 Abbildungsverzeichnis . . . . .	79

# 1 Einleitung

## 1.1 Kurzbeschreibung [M]

James ist ein plattformunabhängiges Suchsystem, welches es den BenutzerInnen erlaubt, Dateien auf ihrem System zu finden. Mithilfe einer neuen Datenstruktur zum Suchen ist James nicht nur schneller als seine Konkurrenten (siehe Abschnitt 2), sondern auch besser anpassbar.

Die Datenstruktur basiert auf Ideen des *Radix Trees* und *Implicit Treaps* um eine Lösung zu bieten, welche gleichzeitig zeit- und speichereffektiv ist. Das auf C++ basierende Suchsystem findet zugleich auf beiden unterstützten Plattformen, Windows und OS X, Einzug.

Unsere Applikation erlaubt es den BenutzerInnen, eigene Suchbereiche zu definieren - und Prioritäten zu diesen - um nur jene Dateien zu indizieren, welche die BenutzerInnen wollen. Durch die Nutzung dieser Prioritäten kann James die Suchresultate in der richtigen Reihenfolge sortieren, basierend auf der eigenen Konfiguration sowie der Anzahl an Aufrufen.

Weiters ist es möglich die Funktionalität von James durch sogenannte “Workflows” zu erweitern. Diese können einfach durch die Nutzung eines interaktiven Editors erstellt werden, wofür keine Programmierkenntnisse notwendig sind.

Durch die Nutzung dieser Workflows (siehe Abschnitt 3.4) kann man James einen Webservice durchsuchen lassen - oder sich erinnern lassen, wenn der Tee fertig ist. Man ist nur durch Ihre eigene Phantasie begrenzt!

Workflows werden in einem standardisierten Format gespeichert, was es ermöglicht diese zwischen den beiden Plattformen auszutauschen. Um diesen Austausch noch besser zu unterstützen, haben wir auch ein Downloadcenter entworfen, wo jede Person auf dem ganzen Globus seine oder ihre eigenen Workflows hochladen kann. Diese können mit der Leichtigkeit eines Mausklicks importiert werden.

## 1.2 Abstract [L]

James is a cross-platform search engine, enabling users to locate documents on their system. By introducing a new data structure for file search, James is not only faster than it's competitors, but also more customisable.

The data structure itself is based on ideas known from *radix trees* and *implicit treaps*, providing a solution that is both time and memory efficient. Written in C++, this search algorithm is used by the OS X as well as the Windows implementation.

Our application allows users to define search scopes, and priorities within those, to index only the files that matter. Utilising these priorities, James orders search resulted based on your configuration as well as the number of times you use them, dynamically adjusting to the users preferences.

Furthermore it is possible to extend James' core functionality by creating so-called "workflows". Those can be built using an interactive editor, which makes it possible to use even for non-programmers.

Using workflows you can let James search an online service for you - or make him send you a notification once your tea is done. You're only limited by your imagination!

Workflows are saved in a standardised way, making sharing between operating systems possible. To support this sharing even better, we have also created a download center where everyone around the world can upload his extensions. Those can then be imported at the ease of a button click.

### 1.3 Aufgabenstellung [M]

Bei dieser Diplomarbeit handelt es sich um die Implementierung eines Programmes zur Datei- / Programmsuche, welches an „Alfred“ (<http://www.alfredapp.com/> nur für OS X) angelehnt ist.

Die Suche wird für das Windows-Betriebssystem sowie Mac OS X implementiert und ist im Vergleich zu Alfred kostenlos.

James wird - genauso wie Alfred - mit einer intelligenten Suche und dessen einfacher Erweiterbarkeit mittels sogenannten Workflows bestechen. Diese ermöglichen es, selbst geschriebene Skripte bei der Eingabe eines Keywords auszuführen um sich z.B. mit dem VPN zu verbinden. Weiters ist die Erstellung eines Webstores geplant, um einen Austausch der Workflows auf eine einfache Art und Weise zu gewährleisten.

### 1.4 Zielsetzung [M]

Der User soll schnell Programme öffnen und Dateien suchen können. Die Suchergebnisse sollten von dem User beeinflusst / personalisiert werden können. Weiters sollte es für den User ein Leichtes sein, seine/ihre eigenen, persönlichen Funktionalitäten zu erstellen, sowie welche aus dem Downloadcenter hinzuzufügen.

### 1.5 Geplantes Ergebnis [M]

Entwicklung eines effizienten (zeit- sowie speicheroptimierten) Such-Algorithmus, welcher es ermöglicht, eine große Menge an Dateien schnell (nach Priorität sortiert) zu suchen und zu ändern. Zusätzlich ist eines unserer Ziele, Workflows so zu designen, dass sie auf OS X sowie Windows erstellt und verwendet werden können, damit man diese einfach über einen Webstore austauschen kann.

## 2 Ist-Situation

### 2.1 Spotlight [L]

Die Standardsuche von OS X, Spotlight, ermöglicht es dem User, Informationen am Mac zu finden. Das inkludiert nicht nur Dokumente, sondern auch E-Mails, Kontakte, etc.

Der wohl größte Vorteil von Spotlight ist, dass es von Apple selbst entwickelt wurde/wird und somit perfekt ins Betriebssystem integriert ist. Doch dadurch, dass Spotlight eine so große Vielzahl an Dingen finden kann, lässt auch die Performance zum Teil zu wünschen übrig. So kann es schnell einmal eine Sekunde dauern bis die Ergebnisliste angezeigt wird.

Doch als “Poweruser” ist gerade diese Performance unabdingbar. Für den durchschnittlichen Mac-User ist Spotlight die optimale Such-Engine, da sie ermöglicht (fast) alles am Rechner zu finden. Doch für erfahrenere User ist es wichtiger, schnell an die gewünschte Datei zu kommen. Ist beispielsweise eine E-Mail gesucht, dann kann das auch mit der Suche im Mail Programm erledigt werden.

Zusätzlich stört bei Spotlight, dass man als BenutzerIn keinen direkten Einfluss auf die Sortierreihenfolge üben kann. Wie am Mac, üblich soll dies “einfach funktionieren”. Doch auch hier wäre es für einem Poweruser angenehmer eingreifen zu können und Regeln zu setzen: “Ordner A ist wichtiger als Ordner B”, “Dateien mit Endung .xy interessieren mich nicht”, usw.

Zudem kommt, dass man Spotlight nicht erweitern kann. Wünschenswert wäre es, wenn man sich eigene Erweiterungen programmieren könnte, welche spezielle Aktionen ausführen.

### 2.2 Alfred [L]

Alfred, <https://www.alfredapp.com/>, ist eine kostenpflichtige Alternative zu Spotlight und nur für den Mac verfügbar.

Neben der schnellen Suche ist der wichtigste Unterschied die sogenannten Workflows. Per Drag and Drop kann man Komponenten miteinander verbinden und so den Ablauf festlegen. Die einzelnen Komponenten können dann per Code angepasst werden.

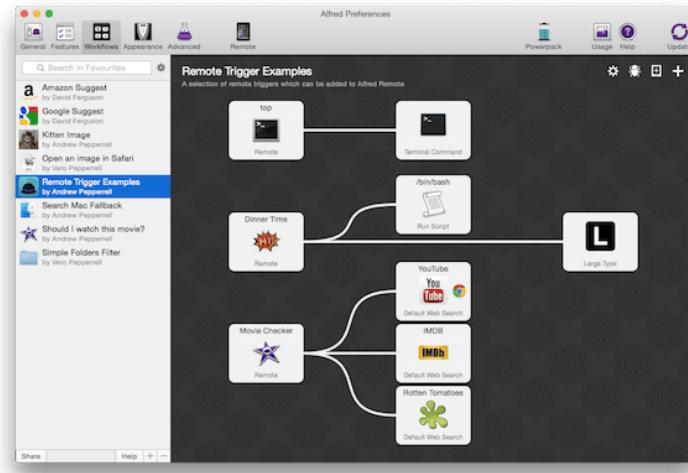


Abb. 1: Alfred Workflows

So ist es z.B. möglich einen Supplierplan Workflow zu schreiben, welcher anstatt Dateien den Supplierplan in den Suchergebnissen anzeigt:

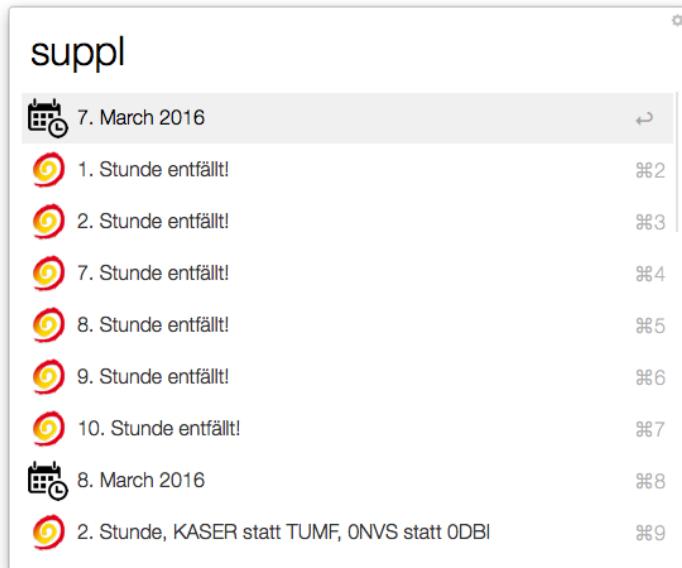


Abb. 2: Supplierplan Workflow

Was leider trotzdem noch fehlt, ist die Möglichkeit die Reihenfolge dieser Suchergebnisse zu beeinflussen. Ähnlich wie Spotlight versucht Alfred selber "mitzudenken" und die Ergebnisse richtig zu ordnen - nur nicht immer mit Erfolg.

## 2.3 Wox [M]

Wox ist eine erst seit kürzlich unter Windows bekannte Alternative. Bei diesem Programm handelt es sich um eine OpenSource Implementierung auf Github:

<https://github.com/Wox-launcher/Wox>

Einer der größten Vorteile von Wox ist die große Community hinter dem Projekt sowie die Konkurrenzlosigkeit unter Windows.

Weiters ist es mit Wox ein Leichtes, sich eigene Plugins / Erweiterungen für den Launcher zu entwerfen. Nicht zu vergessen ist dabei die große Anzahl an Plugins, welche unter <http://www.getwox.com/plugin> erreichbar sind.

Als Manko gilt, dass die Suche auf einer externen Software Everything ([voidtools.com](http://voidtools.com)) basiert. Dies hat den Nachteil, dass die Suchengine nicht spezifisch an den Launcher angepasst / abgestimmt ist. So ist es leider nicht möglich die Priorität, Dateitypen sowie Ordner nach eigenen Wünschen einzurichten.

Ein weiterer Mangel besteht in der Performance des Launchers. Durch die Verwendung des WPF-Controls ListBox erhält man während der Nutzung des Launchers teils deutliche Verzögerungen (siehe Abschnitt 4.5.5).

Durch die von Wox zur Verfügung gestellten Schnittstellen in verschiedenen Sprachen (u.a. C#, Python, NodeJS, Golang, ...) ist auch der Aufwand zur Einarbeitung in das Pluginsystem deutlich größer und stellt damit eine Hürde für neue EntwicklerInnen dar.

Weiters ist noch ein nicht ganz so ressourcenschonender Umgang von Wox zu erwähnen, da Wox in den Standardeinstellungen einen Arbeitsspeicherverbrauch von deutlich über 100 MB aufweist.

## 3 Lösungsansätze

### 3.1 Suche [L]

Die Suche ist eine der Kern-Funktionalitäten von *James*. Sie sollte folgende Eigenschaften haben:

- Schnell
  - Beste X Ergebnisse (nach individuellen Prioritäten)
- Geringer Speicherverbrauch
- Optimalerweise plattformunabhängig

#### 3.1.1 Datenbank [L]

Unsere erste Idee dazu war eine Datenbank. Diese sind ja bekannt dafür, Abfragen schnell behandeln zu können - also wieso nicht auch eine Dateisuche?

Hierfür haben wir zum Testen eine *MySQL* Datenbank aufgesetzt und eine Tabelle mit Pfaden und Prioritäten erstellt. In diese wurden dann alle relevanten Dateien unseres Rechners eingefügt.

Nach einigen *select* Statements stellte sich aber heraus, dass die Performance für unsere Zwecke nicht ausreichend ist - auch ein Index konnte dabei nicht helfen.

Erklären kann man das vielleicht damit, dass Datenbanken auch nicht auf String-Suchen optimiert sind. So können Systeme wie MySQL zwar schnell Tabellen joinen und ID's miteinander matchen, aber prüfen welche Strings einen gewissen Suchstring enthalten und davon jene  $k$  mit höchster Priorität zu finden, ist einfach nicht deren Fokus.

Trotzdem muss mehr aus einer Datenbank zu holen sein, dachten wir uns. So kamen wir auf die Idee, es mit einer in-memory Datenbanken zu versuchen.

Auch MySQL bietet dazu eine Option. Wir wiederholten also unseren Test. Diesmal passte die Performance, aber der RAM Verbrauch war deutlich zu hoch.

Neben den Performance-/Speicherproblem mit diesem Lösungsansatz, ist es natürlich nicht so einfach ein Datenbank System mit der Cocoa/.NET Applikation auszuliefern. Wir konnten zumindest keine packaged-version finden.

Zusätzlich ist es aus der Sicht der BenutzerInnen auch nicht wünschenswert, die ganze Zeit eine Datenbank im Hintergrund laufen zu lassen.

Somit entschlossen wir uns diesen Ansatz zu verwerfen.

### 3.1.2 Spotlight [L]

Eine andere Alternative ist natürlich auch die Schnittstelle der Mac-Suche *Spotlight* zu verwenden. Hierfür gibt es im Cocoa Framework die *NSMetadataQuery* Klasse. Eine einfache Abfrage zu erstellen und beispielsweise alle Dateien beginnend mit “A” zu finden ist mit Hilfe des Apple Tutorials, <https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/SpotlightQuery/Concepts/QueryingMetadata.html>, kein Problem.

Bei genauerem Betrachten stößt man aber schnell auf eine Vielzahl von Problemen:

- Geschwindigkeit

Eine *NSMetadataQuery* gibt immer alle zutreffenden Suchergebnisse zurück. Gerade bei einfachen Abfragen können dies sehr (sehr) viele werden. Und dadurch ist die Performance nicht die, die man sich erwartet.

- Schwierig für unsere Use-Cases anzupassen

Ein Grundziel von James ist es, Prioritäten für Files konfigurieren zu können, sodass auch jene Files gefunden werden um die es wirklich geht.

Standardmäßig bietet die Spotlight-API keine Möglichkeit nach “eigenen Werten” zu filtern. Dies liegt vermutlich auch daran, dass ihr Suchindex auf die Spotlight Requirements ausgelegt ist. Es ist jedoch möglich nach jedem der File-Attribute abzufragen/sortieren. Deshalb versuchten wir mit Hilfe von Extended-Attributes, *xattr*, eine Priorität zu den Files hinzuzufügen. Problem hierbei ist nur, dass man natürlich Schreibrechte auf der Datei braucht. Dies ist jedoch bei System-Applikationen wie dem Finder, Calender, Mail, Terminal, ... nicht der Fall und nur als Superuser möglich.

Als Alternative könnte man natürlich im Nachhinein auch selber filtern/sortieren - doch die Performance einer solchen Lösung ließ mehr als nur zu Wünschen übrig.

### 3.1.3 Lucene [M]

Apache Lucene ist ein freie, populäre Software von der Apache Software Foundation. Das Ziel des Frameworks ist in erster Linie eine hoch performante, individuell anpassbare Suchengine fürs Durchsuchen großer Textmengen bereitzustellen. Lucene wird unter anderem von Wikipedia und Twitter verwendet. Grundsätzlich wird Lucene in Java entwickelt, es existieren jedoch viele (in)offizielle Ports für diverse Programmiersprachen, unter anderem für C#: Lucene.NET.

Lucene besitzt zwar eine sehr aktive Community, jedoch nur die originale Fassung in Java. Da Lucene.Net ein inoffizieller Port zum .NET Framework ist, verfügt es über eine deutlich schwächere Community und schwacher Dokumentation. Weiters hinkt die Entwicklung des .NET - Clones der originalen Version deutlich hinterher und unterscheidet sich in manchen Punkten. Dadurch ist es auch nicht möglich die originale Java Dokumentation zu verwenden.

Zur Testung entwickelten wir einen funktionalen Prototypen. Dadurch machten wir uns mit Lucene vertraut und dessen umfangreichen Konfigurationsmöglichkeiten, kamen jedoch im

Laufe der Zeit darauf, das Lucene für unseren Usecase nicht das Richtige ist. Einerseits ist der Funktionsumfang viel zu groß für unsere Belange, da Lucene für deutlich mächtigere Use-Cases als das Suchen von Dateipfaden entwickelt wurde. Ein weiterer Grund zur Abwandlung von Lucene ist die Performance, da Lucene seinen Index auf die Festplatte schreibt und dadurch nicht im Hauptspeicher arbeiten kann und die Index Dateien von Lucene schnell einen Gigabyte überschreiten. Auf einer herkömmlichen Festplatte war es dadurch auch überhaupt nicht zu gebrauchen und auf SSD's ist es für unsere Belange noch immer zu langsam.

Abschließend kann man sagen, dass Lucene ein mächtiges Framework ist, welches ihre Stärken jedoch in der Indizierung von großen Dokumenten und nicht von lediglich Dateipfaden zeigt. Durch diesen Umstand und da kein Port für Objective C oder Swift existierte, mussten wir jedoch diesen Prototypen verabschieden.

### 3.1.4 Eigene Implementierung [L]

Da Spotlight unter OS X, sowie Lucene unter Windows nicht die gewünschten Resultate brachten und jeweils nur unzureichend unsere Usecases erfüllen, entschieden wir uns für eine Eigenentwicklung.

Ähnlich wie bei Standard vs. Individualsoftware sieht es mit der eigenen Implementierung aus: (viel) mehr Aufwand - aber genau unseren Anforderungen gerecht.

## 3.2 Auswahl der UI Technologie [M]

Da wir James von Anfang an für OS X sowie Windows entwickeln wollten, stellte sich bereits zu Beginn die Frage, ob wir auf eine plattformunabhängige Programmiersprache wie Java setzen. Die Entscheidung für Java würde folgende Vor- und Nachteile mit sich bringen:

### Vorteile [M]

- Durch die Plattformunabhängigkeit von Java muss nur eine Version für beide Betriebssystem geschrieben werden
- Weite Verbreitung von JavaFX
- Bereits in der Schule gelernt

### Nachteile [M]

- Eine Installation von Java wäre für den Benutzer eine Voraussetzung
- Launcher benötigt eine nähere Anbindung ans Betriebssystem

- Look and Feel wäre auf beiden BS gleich und würde sich nicht an die Designrichtlinien der beiden Plattformen halten
- Eigene plattformabhängige Entwicklung von Modulen zur Kapselung systemnaher Funktionen wären notwendig

Auf Grund der in dieser Auflistung überwiegenden Nachteile kamen wir zum Schluss, dass Java für uns nicht die Lösung sein kann.

Daraufhin probierten wir kurz noch andere plattformunabhängige Programmiersprachen wie Python und C++ für die UI Entwicklung aus, mussten diese jedoch auf Grund von fehlender Funktionalität bzw. Verbreitung ziemlich bald verwerfen.

So blieb uns nichts anderes übrig, als für das jeweilige Betriebssystem eine eigene Lösung zu entwerfen.

### 3.2.1 OS X [L]

Auf OS X liegt die Entscheidung hierfür auf der Hand: Das Cocoa Framework von Apple ist die Lösung, in der selbst die Standardapplikationen des Betriebssystems geschrieben werden. D.h. wenn man ein Mac look-and-feel erreichen will, dann ist das Cocoa Framework die (einzig) richtige Wahl.

Als Programmiersprache ist man dann auf das hauseigene Objective C angewiesen. Doch seit der Worldwide Developer Conference (WWDC) 2014 gibt es auch eine Alternative: Swift. Es überzeugt mit einer modernen Syntax (speziell im Vergleich zu Objective C) und - zumindest laut Apple - schnellen Performance.

Swift ist kompatibel mit Objective C, wodurch Libraries, welche vor 2014 entwickelt wurden, auch verwendet werden können. Und dank Objective C++ können so Swift, Objective C und C(++) in einem einzelnen Programm verwendet werden, was gerade im Hinblick auf die gemeinsame Such-Engine von James wichtig ist.

Zum Erstellen von Benutzeroberflächen bietet XCode, Apple's IDE, mit Storyboards sowie xib Files einen (mehr oder weniger) einfachen Weg ohne großen Aufwand eine GUI zu erstellen.

### 3.2.2 Windows [M]

Unter Windows haben sich zwei Frameworks für die UI Entwicklung etabliert. Einerseits das neue WPF (Windows Presentation Frameworks) sowie das etwas ältere Windows Forms. Im folgenden Abschnitt werden wir eine Gegenüberstellung dieser beiden Frameworks vornehmen und daraus den Entschluss auf eines der Beiden ziehen, jedoch zählen wir zuerst die Gemeinsamkeiten kurz auf:

- Bestandteil des .NET Frameworks
- Tiefe Integration ins Windows Betriebssystems

- große Community, weite Verbreitung

**Windows Forms** Mit dem Betriebssystem Windows XP hat Windows Forms eine große Beliebtheit sowie Verbreitung erlangt. Microsoft bietet dadurch eine einfache Schnittstelle zur Erstellung grafischer Benutzeroberflächen. Es ist ein Wrapper für die bereits länger existierende GDI (Graphics Device Interface, ist ein Bestandteil der Windows API), welche zuvor nur mittels C++ gesteuert werden konnte. Dadurch war es ein Leichtes, ohne viel Aufwand eine Benutzeroberfläche in den .NET Sprachen (u.a. C#, VB, C++ CLR) zu erstellen.

**WPF** Mit der Windows Presentation Foundation entwickelte Microsoft eine neuere Möglichkeit Oberflächenprogrammierung unter Windows zu betreiben und setzt nicht wie Windows Forms auf die Windows API sowie der GDI+ Schnittstelle auf, sondern direkt auf DirectX. Dadurch verwendet WPF wenn möglich automatisch eine Hardwarebeschleunigung, was zu einer steigenden Performance führt. Außerdem wurde eine Vielzahl an neuen Technologien integriert. Im Abschnitt "Technische Beschreibung"(4) wird genauer auf die jeweiligen Technologien Bezug genommen:

- XAML (Extensible Application Markup Language)
- DataBinding
- Styles
- Routed Event
- "lookless"Controls

**Auswahl** Wir wählten für unser Projekt WPF, da es gegenüber WinForms deutlich mehr Flexibilität sowie Features bietet, jedoch kaum Nachteile gegenüber WinForms aufweist.

### 3.3 Personalisierung der Suchergebnisse [L]

Da jeder Mensch unterschiedliche Ansprüche an eine Dateisuche stellt, wollten wir es einfach machen die Suche nach seinen eigenen Vorlieben zu gestalten und uns so teils deutlich von der Konkurrenz abzuheben. Daher suchten wir eine einfache Möglichkeit, in der der/die BenutzerIn leicht Einstellungen festlegen können, welche Dateien (Dateiendung) überhaupt in den Index aufgenommen werden soll und mit welcher Priorität dies geschehen sollte.

Für ProgrammiererInnen kann es z.B.: wichtig sein .json - Dateien priorisiert anzuzeigen, der Durchschnittsverbraucher kennt meistens diese Dateiendung nicht und will diese auch nicht angezeigt bekommen. Für den einen ist der "*Desktop*" Ordner wichtig, für den anderen wiederum der "*Dokumente*".

Da wir jedoch diese Vielzahl an Personen nicht in einen Topf werfen und diese dadurch in ein enges Korsett zwingen wollen, entschlossen wir uns für das Prinzip "Konvention vor Kon-

figuration” zu setzen. Zwar gibt es von Beginn an festgelegte Einstellungen, jedoch können diese vom User komplett überarbeitet / angepasst werden.

Die Konfigurationsmöglichkeit des Search Scopes sowie Prioritäten sind unser Ansatz, um dieses Problem zu lösen.

Jeder Datei im Suchindex ist eine Priorität  $p \in \mathbb{N}$  zugeordnet. Beim Suchen werden dann die  $k$  besten, im Sinne der Priorität, Suchergebnisse zurückgegeben.

Bei der Installation wird ein Default Search Scope konfiguriert, welcher folgende Ordner umfasst:

- User Ordner
  - Desktop
  - Downloads
  - Documents
  - Pictures
  - Movies
  - Music
- Programme
  - Applications (Mac)
  - ProgramFiles (Windows)
  - ProgramFilesX86 (Windows)
- Startmenü (Windows)
  - CommonStartMenu
  - StartMenu
- Systemeinstellungen
  - /System/Library/PreferencePanes (Mac)

Selbstverständlich können Ordner nach Belieben hinzugefügt bzw. entfernt werden.

Für jeden dieser Ordner kann konfiguriert werden, welche Dateiendungen hier von Bedeutung sind. So sind am Mac z.B. im Programme Ordner nur .app Dateien von Interesse, wohingegen am Desktop z.B. auch .pdf Dateien dabei sind. Für jede dieser Dateiendungen kann eine Priorität konfiguriert werden, sprich wie wichtig dieser Dateityp ist. Weiteres kann konfiguriert werden ob Ordner indiziert werden sollen oder nicht. So will man im Programme Ordner beispielsweise wirklich nur die eigentlichen Programme finden, jedoch im User Ordner auch die Ordner selber. Gleichzeitig hat jeder Ordner auch eine eigene Priorität, d.h. alle Dateien in diesem Ordner sind schon von Grund auf so (un)wichtig.

Zusätzlich gibt es die Möglichkeit eine Liste an Default-Extensions festzulegen. Diese werden, wenn gewünscht, auf verschiedene Search Scopes übernommen, sodass gewisse Standardprioritäten nicht für jeden Ordner extra konfiguriert werden müssen. Durch Setzen einer Extension zu einem SearchScope wird dieser Wert der Default-Extensions Liste vorgezogen.

Zudem ist es möglich Ordnernamen festzulegen, die nie indiziert werden sollen. Beispiele dafür sind z.B. “bin” oder “gen” Ordner, die beim Kompilieren von Programmen oft entstehen.

Zusätzlich wird noch die Länge des Namens in die Priorität eingespielt. So werden kürzere Dateinamen mit der gleichen Endung und Search Scope längeren vorgezogen. Beispiele:

- *chrome.exe* soll der *old\_chrome.exe* vorgezogen werden
- *filezilla.exe* der Setupdatei *FileZilla\_3.16.0\_win64-setup.exe* vorgezogen werden

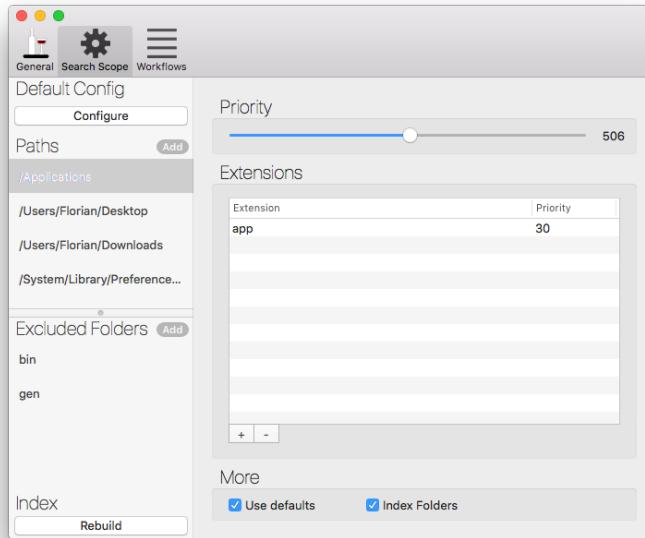


Abb. 3: Search Scope Konfiguration OS X

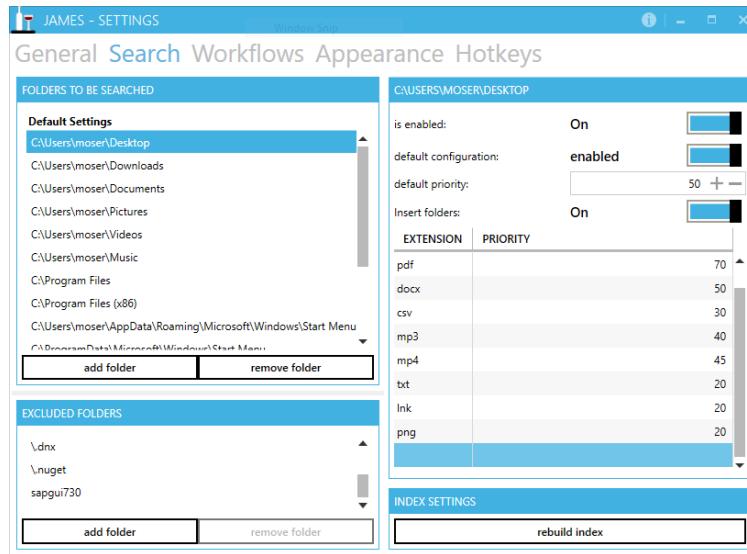


Abb. 4: Search Scope Konfiguration Windows

Zusätzlich werden Dateien oder Ordner, welche von James geöffnet werden, mit einer Priorität von 5 erhöht. Damit erreicht man einen wichtigen Punkt für die Personalisierung der Suchergebnisse, da Ordner und Dateien, welche öfters aufgerufen werden, eine höhere Priorität erhalten.

### 3.4 Workflows [M]

Ein großer Bestandteil des Projektes ist die eigens entwickelte Workflow-Engine. Diese soll es den BenutzerInnen leicht machen eigene Funktionalitäten bzw. Erweiterungen für James zu erstellen. Ein Workflow ist somit ein Hilfsmittel zur Unterstützung des Endanwenders um oft verwendete Aktionen zu automatisieren.

Ein Workflow kann beispielsweise dafür erstellt werden, dass bei Eingabe des Keywords "suppl" der Supplierplan vom Internet abgefragt wird und als Suchergebnis dargestellt wird (siehe Abschnitt 2). Workflows bestehen somit aus mehreren Komponenten (KeywordInput, ScriptAction, etc. - Abschnitt 3.4.2), welche den Ablauf beschreiben.

Durch folgende Punkte hebt sich James deutlich von seiner Konkurrenz ab:

#### Programmiersprachenunabhängig [M]

Wir möchten es dem/der WorkflowstellerInnen so einfach wie möglich machen eigene Workflows zu erstellen. Damit er oder sie nicht extra eine Programmiersprache erlernen muss, haben wir uns dazu entschlossen eine Vielzahl an Programmiersprachen zu unterstützen (siehe Actions, Abschnitt 3.4.2.2).

### 3.4.1 Interaktiver Workfloweditor [M]

Dieser Workfloweditor stellte einen großen Arbeitsaufwand im gesamten Projektverlauf dar. Vorbild des Editors war das Blueprintssystem der Unrealengine. Durch dieses System haben wir es erreicht, eine Möglichkeit zu bieten, auch komplizierte Workflows zu erstellen. Da wir nach kurzer Recherche für beide Plattformen feststellen mussten, dass eine geeignete Programmbibliothek zur Darstellung unserer Workflows nicht existieren, entschieden wir uns wiederum, wie schon bei der Suchengine, für ein eigenes System (Details siehe Abschnitt 4.7).

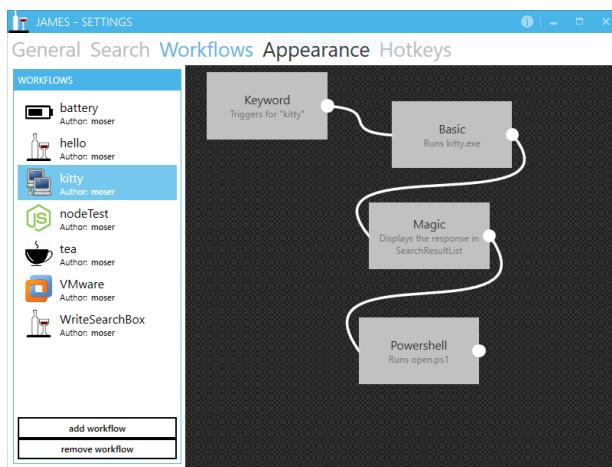


Abb. 5: Win: Interaktiver Workfloweditor

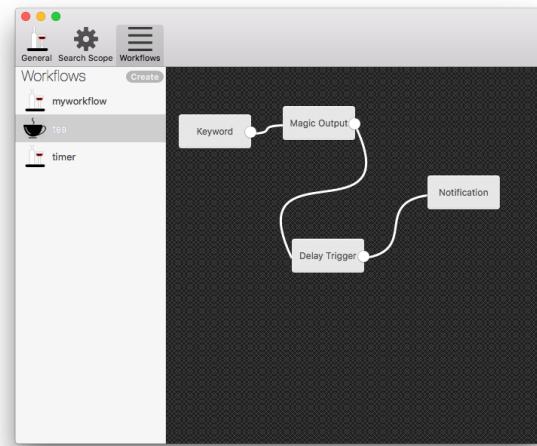


Abb. 6: OSX: Interactive Workflow Editor

### Funktionsweise [M]

Grundsätzlich stellt jede Workflowkomponente eine eigene Box dar. Diese können je nach Typ mit anderen verbunden werden oder nicht. Auf der linken Seite der Box kommt immer die Verbindung von den vorherigen Komponenten herein und rechts verlässt die Verbindung diesen Komponenten um einen anderen Komponenten aufzurufen. Je nach Typ können von diesen Komponenten Verbindungen eingehen sowie ausgehen. Zum Beispiel besitzt ein Trigger keine eingehenden Verbindungen, da diese direkt von James aufgerufen werden. Wird nun ein Keyword-Trigger von James ausgeführt, werden alle ausgehenden Verbindungen durchgegangen und die jeweils nächste Komponente aufgerufen. Dabei wird diesem Aufruf ein String Array als Parameter immer übergeben, bei einem Keyword-Trigger stellt dies die eingegebenen Wörter in der SearchBox dar und bei einer Action die zeilenweise gesplitteten Zeilen der erzeugten Ausgabe des Programms.

Mehrere Verbindungen zwischen zwei Komponenten sind nicht möglich, da dies auch keinen Sinn ergeben würde, es ist jedoch möglich Zyklen bei entsprechenden Anwendungsfall zu erstellen.

Durch einen Doppelklick oder durch Auswählen des entsprechenden Eintrags im Kontextmenü der Komponente, gelangt man zu einem Formular zur genaueren Konfiguration dessen. Dieses wird je nach Typ anders aussehen und wurde mittels Reflection gelöst (Details siehe Abschnitt 4.7.3). So gibt es beispielsweise bei einem Keyword-Trigger mehrere Argumente zur Konfiguration:

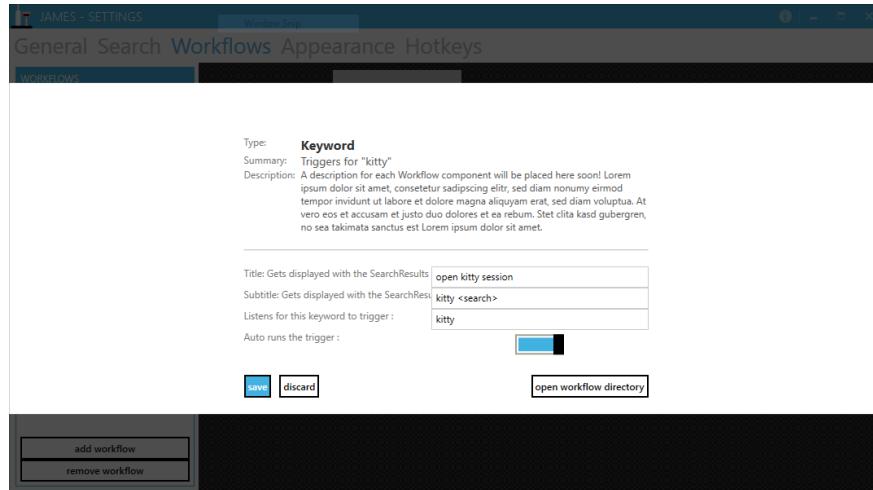


Abb. 7: Win: Komponente-Konfigruation

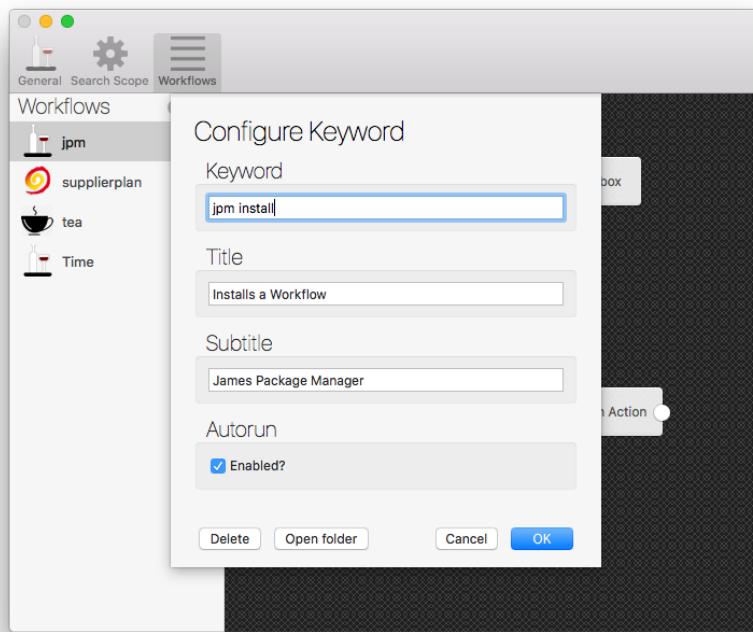


Abb. 8: OSX: Komponente-Konfigruation

- **Titel:** Dies stellt den Titel des Eintrages unter den Suchresultaten dar, da jedes Keyword in einem Eintrag in die Ergebnisliste resultiert.

- Subtitel: Dieses Feld stellt den Subtitel des Eintrages unter den Suchresultaten dar.
- Keyword: Auf eine String “*a*” der gehorcht werden soll. Ist die aktuelle Eingabe in der Suchbox einen Prefix von “*a*” dar, wird dieser Trigger in die Resultate aufgenommen und verdrängt dadurch gegebenenfalls andere Resultate.
- AutoRun: Dieses Flag erlaubt es das Keyword auf den AutoRun Status zu setzen. Sobald das Keyword ein Suffix der Eingabe entspricht, wird dieser automatisch ausgeführt und kein Eintrag mehr für das Keyword angezeigt. Diese Konfiguration ermöglicht es leicht interaktive Suchen zu erstellen. Im Folgenden möchten wir das anhand des Kitty-Workflows erklären. Der Kitty-Workflow bietet die Möglichkeit alle gespeicherten SSH-Sessions zu durchsuchen und die ausgewählte Session sofort zu starten. Dafür ist der Keyword-Trigger auf AutoRun und mit dem Keyword “*kitty*” konfiguriert. Ist die Eingabe nun ein Prefix des Keywords (z.B.: “*kitt*”) wird nur ein Eintrag des Keywords in den Resultaten angezeigt:

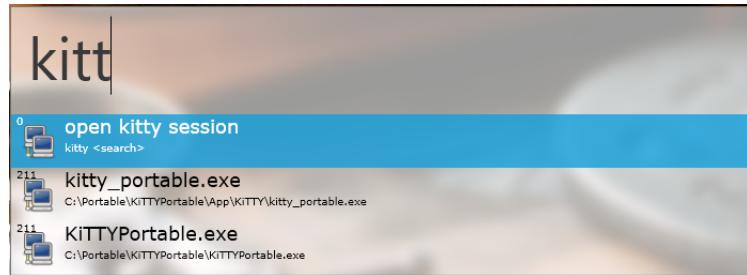


Abb. 9: Beispiel Keyword mit AutoRun

Sobald das Keyword ein Prefix von der Eingabe ist, wird dieser automatisch ausgeführt und startet in diesem Beispiel eine Action, welche alle gespeicherten SSH-Sessions zum Suchfilter (2.Argument) auflistet:



Abb. 10: Beispiel Keyword mit AutoRun 2

## Plattformunabhängig [M]

Durch die Vielzahl von uns unterstützten Programmiersprachen zur Erstellung von Workflows gibt es auch einige darunter, welche von beiden Plattformen unterstützt werden: u.a. NodeJs und Python. Durch diese ist es möglich einen Workflow nur einmal zu schreiben und

auf beiden Plattformen zu verwenden. Diese Möglichkeiten zum Austausch von Workflows zwischen den Plattformen ist wiederum ein Alleinstellungsmerkmal für unseres James.

### **Workflow Export/Import - Funktion [M]**

Die Export / Import - Funktion stellen einen wichtigen Aspekt der Workflow-Verwaltung dar. Damit ist es möglich ein Backup für einzelne Workflows zu erstellen, sowie diese ins Downloadcenter hochzuladen und einer Vielzahl an BenutzerInnen zur Verfügung zu stellen.

### **Workflow-Downloadcenter [M]**

Das Downloadcenter bietet die Möglichkeit Workflows für alle User zugänglich zu machen. Dazu muss der / die BenutzerIn sich nur auf der Website registrieren, auch OAuth-Anmeldung via Facebook und Google sind möglich, und seinen Workflow mitsamt Beschreibung hochzuladen. Diese müssen dann nur noch von einem Admin genehmigt werden, damit diese öffentlich zugänglich werden.

#### **3.4.2 Workflow-Komponenten [M]**

Zur Erstellung von Workflows werden mehrere Workflowkomponenten verbunden. Ein Workflow ist plattformunabhängig, wenn jede Komponente für sich selbst plattformübergreifend ist. In der folgenden Auflistung sind plattformunabhängige Komponenten speziell mit einem [CP] gekennzeichnet. Für eine bessere Übersicht haben wir die einzelnen Komponenten in 3 grundsätzliche Kategorien unterteilt:

**3.4.2.1 Triggers [M]** Verschiedene Triggers geben dem User die Möglichkeit das Starterereignis, sprich wenn ein Workflow ausgeführt werden soll, zu definieren. Dafür steht folgende Auswahl an Triggern zur Verfügung:

**Keyword-Trigger [CP]** Dieser Trigger definiert ein Schlüsselwort (Keyword) auf welches gehört wird. Wird dieses Schlüsselwort zumindest teilweise in die SearchBox eingegeben, wird der Trigger in den Ergebnissen angezeigt. Beim Öffnen des Resultates (Eingabe oder Klick) wird dieser ausgelöst. Es besteht auch die Möglichkeit das “autorun” Attribut auf true zu setzen. Dann wird das Keyword automatisch ab Vollständigkeit des Schlüsselwortes ausgeführt.

**Api-Trigger [CP]** Dieser Trigger bietet eine einfache API an. Damit ist es möglich einen Workflow von einem anderen Programm aus zu starten. Dafür wurde im Betriebssystem ein sogenanntes Custom-Url-Protocol eingetragen. Horcht jetzt zum Beispiel ein Api-Trigger auf

“insert”, wird dieser über einen HTML Link im folgenden Format gestartet (Implementierung siehe Abschnitt 4.7.4):

```
<a href="james:insert/parameter">Hier Workflow starten</a>
```

**3.4.2.2 Actions [M]** Die Actions bieten die Möglichkeit eigene Scripte sowie Programme zu starten. Weiters besitzen alle Actions eine Einstellung, ob diese im Hintergrund ”background” weiterlaufen (Abschnitt 4.7.6) oder abgebrochen werden sollten, falls James während der Laufzeit des Workflows geschlossen/ausgeblendet wird oder sich die aktuelle Eingabe verändert. Dafür steht folgende Auswahl an Actions zur Verfügung:

**Basic-Action** Beginnen wir mit der wohl generischsten Action von allen. Mit dieser kann man jede Executable mit eigens definierbaren Parameter starten.

**NodeJs-Action [CP]** Da NodeJs auf beiden Plattformen verfügbar ist, ist diese Action auch plattformunabhängig. Dadurch können NodeJs-Actions einfach zwischen OS X und Windows ausgetauscht werden. Zur Konfiguration besitzt dieser Workflow ein Attribut für den relativen Pfad zum Scriptfile.

**Python-Action [CP]** Wie NodeJs-Action ist auch Python plattformunabhängig und besitzt auch das gleiche Attribut für das Scriptfile.

**Powershell-Action** Wie der Name schon sagt, handelt es sich hier um eine Action zum Starten eines Powershell-Scriptes. Da Powershell nur unter Windows verfügbar ist, ist diese Action nicht plattformunabhängig. Wie die meisten anderen Actions besitzt diese Action ein Attribut für den relativen Filepath zum Scriptfile.

**Delay-Action [CP]** Diese Action erzeugt lediglich eine gewisse zeitliche Verzögerung (Delay). Es besitzt ein Attribut ”multiplier”, das den einkommenden Wert multipliziert. Der einkommende Wert kann mithilfe des Formatstrings (Abschnitt 4.7.5) auch statisch gesetzt werden. Diese Action ist für beiden Plattformen implementiert.

**AppleScript** Dieser Komponente erlaubt es ein AppleScript (osascript) auszuführen. Da dies nur am Mac verfügbar ist, gibt es diese Action auch nur für OS X.

**PerlAction** Ähnlich den anderen Actions, aber für Perl Skripte.

Zusätzlich gibt es noch Actions für andere Programmier-/Skriptsprachen:

- Perl

- Ruby [CP]
- PHP [CP]
- Swift
- Bash
- Zsh

**3.4.2.3 Outputs [M]** Verschiedene Outputs stellen für die Ausgabe bestimmter Workflows bereit. Dafür stehen folgende Outputs zur Verfügung:

**Clipboard-Output [CP]** Dieser Output kopiert den eingehenden Text in die Zwischenablage.

**LargeType-Output [CP]** Dieser Output bietet die Möglichkeit den Text groß darzustellen. LargeType ist auch über die Tastenkombination (ALT + L(Windows), oder CMD + L (OS X)) erreichbar. Dabei wird, falls ein MagicOutput selektiert ist, dieser Text, ansonsten der Text der SearchBox angezeigt.



Abb. 11: Win: LargeType-Output

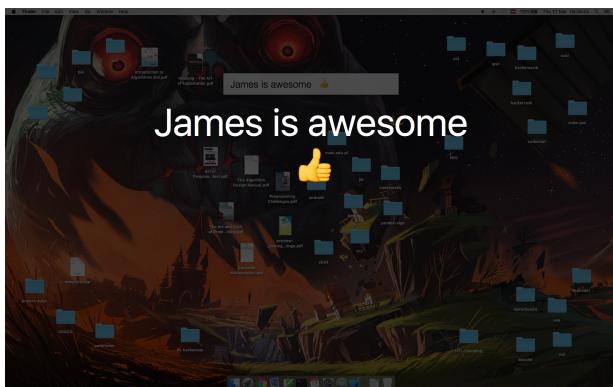


Abb. 12: OSX: LargeType-Output

**Notification-Output [CP]** Dieser Output zeigt den eingehenden Text als Nachricht (Notification) an.

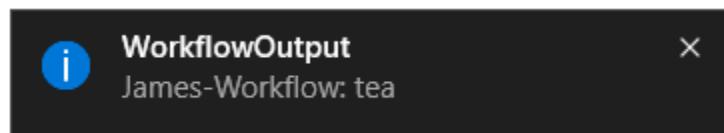


Abb. 13: Win: Notification-Output

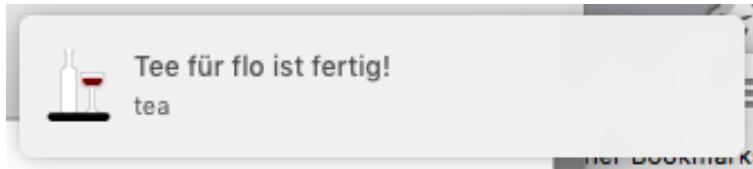


Abb. 14: OS X: Notification-Output

**Magic-Output [CP]** Der Magic-Output bietet die Möglichkeit Resultate in den Ergebnissen anzuzeigen. Dabei können Title, Subtitle sowie das Icon angegeben werden. Magic-Outputs können vom User ausgewählt und gestartet (Enter oder Klick) werden.

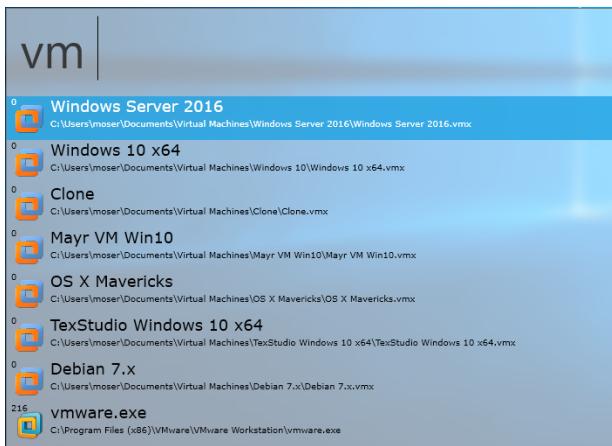


Abb. 15: Win: Magic-Output

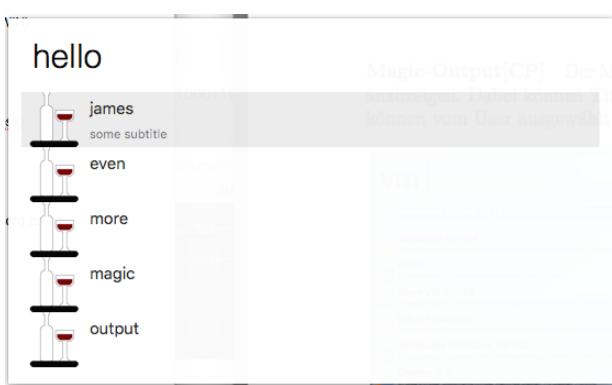


Abb. 16: OSX: Magic-Output

**Open-Output [CP]** Der Open-Output bietet die Funktionalität um Links, Dateipfade sowie Programme zu öffnen.

**SearchBox-Output [CP]** Dieser Output ersetzt den aktuellen Text der Searchbox mit dem eingehenden Text.

### 3.5 James Downloadcenter [M]

Ein Ziel war ein eigenes Downloadcenter, welches den Austausch der Workflows sowie die Verbreitung des Programmes übernehmen sollte. Folgende Anforderungen wurden an das Downloadcenter gestellt:

- Ansprechende Startseite um James dem Besucher näher zu bringen
- Downloadbereich für die beiden James Versionen (Windows, OSX)
- Workflows

- OAuth Anbindung für diverse Anbieter
- Upload von Workflows nur mittels Benutzeraccounts möglich
- Verifizierung der Workflows vor dem Veröffentlichen durch einen Admin

Durch die gegebenen Anforderungen starteten wir zunächst ein Auswahlverfahren folgender Webtechnologien:

- PHP 7
- ASP.NET 4.6
- ASP.NET Core 1.0

Im Folgenden gehen wir näher auf die jeweilige Technologie ein und erläutern deren Vor- und Nachteile.

### 3.5.1 PHP 7 [M]

PHP 7 ist eine serverseitige Skriptsprache, welche hauptsächlich zur Erstellung von dynamischen Webseiten angewendet wird. Durch dessen hohen Marktanteil jenseits der 80% ist es natürlich in aller Munde.

#### Vorteile

- große Verbreitung
- Vielzahl von Funktionsbibliotheken
- breite Datenbankunterstützung
- Vielzahl von gratis Webhostinglösungen

#### Nachteile

- gewöhnungsbedürftige Syntax
- keine einheitlichen Designrichtlinien
- niedrigere Performance im Vergleich zu kompilierten Webanwendungen.

### 3.5.2 ASP.NET 4.6 [M]

ASP.NET ist die Lösung von Microsoft für dynamische Webseiten. Durch das Konzept von ASP.NET MVC besticht dieses Framework mit einer klaren Aufteilung und Designrichtlinien für den Aufbau des Codes. Mit dem zweithöchsten Marktanteil nach PHP mit 16% ist diese Variante nicht zu unterschätzen.

## Vorteile

- Klare Designvorgaben durch MVC oder MVVM sowie objektorientierter Programmierung
- Kompilierte Sprache (Performance Vorteile gegenüber PHP)
- Entity Framework
- Bereits im Unterricht gelernt.
- Schnellere / Einfachere Entwicklung
- Integriertes User Management

## Nachteile

- Kaum externe Hostingmöglichkeiten
- Hosting nur mit einem Windows Server und IIS (Internet Information Services) möglich
- Bestandteil vom .NET Framework. Dadurch können Erweiterungen sowie Patches nur mit einer neuen Version des .NET Frameworkes ausgeliefert werden.

### 3.5.3 ASP.NET CORE 1.0 [M]

ASP.NET CORE 1.0 ist eine Neuentwicklung von ASP.NET vom Grund auf, welches sich aktuell noch in der Beta-Phase befindet.

Technische Details finden Sie in Abschnitt 4.6.

Durch die Ähnlichkeit mit dem ASP.NET 4.6 Framework gehe ich nicht mehr auf die Vorteile von ASP.NET CORE 1.0 ein sondern vergleiche es lediglich mit ASP.NET 4.6.

## zusätzliche Vorteile gegenüber ASP.NET 4.6

- Open Source
- Plattformübergreifend
- State of the Art
- Modularisiert
- Leichtgewichtiger mit Hilfe der Modularisierung und dadurch schneller

## Nachteile gegenüber ASP.NET 4.6

- Beta Phase
- Kaum öffentliche Dokumentation

## Finale Entscheidung

Anhand dieser Punkte evaluierten wir die einzelnen Technologien und kamen zum Schluss, dass es eine ASP.NET Lösung werden soll.

Da das neue ASP.NET CORE 1.0 einen interessanten Ansatz mit der plattformübergreifenden Entwicklung bot und alle Vorteile von ASP.NET 4.6 aufgriff, entschieden wir uns unser Downloadcenter mittels ASP.NET CORE 1.0 umzusetzen.

## 4 Technische Beschreibung

In diesem Kapitel folgen nun die technischen Beschreibungen der Lösungsansätze aus Kapitel 3.

### 4.1 Suchalgorithmus [L]

Wie in Abschnitt 3.1 erklärt, haben wir uns für eine eigene Implementierung der Suche entschieden. Dafür wird ein Suchalgorithmus benötigt.

Dieser sollte folgende Operationen effizient unterstützen:

- $insert(s, p)$   
Hinzufügen eines Strings  $s$  mit Priorität  $p$ .
- $remove(s)$   
Entfernen eines Strings  $s$ . Dies soll auch rekursiv funktionieren.
- $rename(old, new)$   
Umbenennen von  $old$  auf  $new$
- $find(s)$   
Jene  $k$  (z.B. “Top 10”) Strings in der Datenstruktur finden, welche maximale Priorität haben und  $s$  matchen.
- $addPriority(s, \Delta)$   
Die Priorität eines Strings  $s$  um  $\Delta$  ändern, sowohl positiv als auch negativ.
- $save(f)$   
Datenstruktur in Datei  $f$  persistieren.
- $load(f)$   
Datenstruktur von Datei  $f$  laden.

Hier ist anzumerken, dass es sich bei den Strings immer um einen Dateinamen, inklusive Pfad, handelt. Damit macht auch das rekursive Entfernen Sinn. Und deshalb ist die *rename* Operation auch nicht so trivial, wie sie zu Beginn vielleicht wirken mag. Beim Umbenennen eines Ordners sollten nicht alle Files darin geändert werden müssen.

Die Konstante  $k$  wird beim Anlegen der Datenstruktur angegeben und kann in den Einstellungen konfiguriert werden. Da sowieso nur die Strings mit höchster Priorität gefunden werden sollen, haben wir uns auf  $k \leq 20$  festgelegt. Denn ähnlich wie bei Google, wo selten Suchergebnisse auf der 2. Seite aufruft, werden auch bei uns nur die besten Ergebnisse, im Sinne der Priorität, relevant sein. Somit ist in den meisten Anwendungsfälle sogar  $k \leq 10$  anzunehmen.

Mehr als 20 Suchergebnisse sind also nicht möglich. Dies hat einerseits den Grund, dass das Ziel unseres Programms nicht ist jede Datei zu finden - dazu gibt es schon Suchen im Betriebssystem. James hilft dabei, die *richtige* Datei *schnell* zu finden. Wenn man dazu zuerst 20

“falsche” Ergebnisse durchklicken muss, dann verwendet man die Software falsch. Zusätzlich haben die Konkurrenten (Abschnitt 2) ähnliche Beschränkungen.

Andererseits würde der Speicherbedarf unserer Datenstruktur bei mehr Suchergebnissen einfach nicht in Relation zum Nutzen stehen, wodurch wir uns dann dagegen entschieden haben.

Weiters ist noch die Definition von *matchen* interessant. Für eine bessere Benutzererfahrung soll der Suchalgorithmus nicht nur Prefix-Matching unterstützen. In folgenden Fällen soll ein Suchtext  $a$  mit einem String in der Datenstruktur  $b$  *matchen*:

- $a$  ist ein Prefix von  $b$
- $a$  beginnt mit einem alphanumerischen Zeichen, ist ein Infix von  $b$  und für den Character vor dem Match,  $c$ , gilt:  $c \in \{‘ ‘, ‘:’, ‘_’, ‘-’, ‘;’, ‘\} \vee isUppercase(c)$
- $a$  ist ein Infix von  $b$  und beginnt mit ‘‘

Das eigentliche Matching (Prefix bzw. Infix) ist case insensitive. So matched beispielsweise “world” mit den Strings “hello world”, “helloWorld”, “hello-World”, “hello\_world“, “hello,world”, nicht aber “helloworld”. Es erlaubt auch “Google Chrome” mit der Query “chro” zu finden, nicht aber mit “oogle”. “test.txt” mit “.txt” würde hingegen wieder funktionieren.

Bei String-Matching-Problemen wie diesen sind Suffix Datenstrukturen wie z.B. Suffix Trees, Suffix Arrays oder der Suffix Automaton die ersten Gedanken. Da diese aber natürlich keine remove Operationen unterstützen, wurden diese Ideen bald verworfen.

Als sehr bekannte Alternative bietet sich der Trie an. Dabei handelt es sich um einen gewurzelten Baum, in dem jede Kante ein Zeichen repräsentiert - und jeder Knoten den String der sich als Konkatenation der Zeichen von diesem Knoten bis zur Wurzel ergibt. (Wenn man alle Prefixe eines Strings einfügt, erhält man also einen “unkomprimierten Suffix Tree”).

Da jeder Pfad einen String repräsentiert, können diese einfach aus einem Trie wieder entfernt werden - man muss nur wissen ob es sich um den letzten String in diesem Subtree handelt. Problematisch bei Tries ist nur der Speicherverbrauch. Um diesen zu minimieren, haben wir uns für eine Radix Tree entschieden, da dieser redundante Knoten zu einem merged und somit Speicher spart.

Die “standard” Variante unterstützt zwar weder Prioritäten noch das Umbenennen, doch dazu später mehr. Beginnen wir mit einer Übersicht des Radix Trees:

#### 4.1.1 Radix Tree [L]

Im Folgenden, wie auch in unserer Implementierung, befindet sich am Ende jedes Strings ein spezielles Zeichen \$ um das Ende eines Strings zu markieren. Dieses Zeichen darf sonst nirgends in den Strings vorkommen. So ein Zeichen ist zwar nicht zwingend notwendig, vereinfacht den Code aber erheblich. Für unsere Implementierung haben wir hierfür ‘\0’ verwendet.

Dadurch, dass jeder String mit  $\$$  endet, kann man sich sicher sein, dass ein Knoten mit dem Label  $\$$  auch sicher ein Leaf ist.

**4.1.1.1 Vergleich Trie [L]** In einem Trie wird jeder Buchstabe als Kante repräsentiert. Zusätzlich gibt es einen “dummy” Knoten, die *root*, von welchem alles ausgeht. Jeder Knoten hat somit maximal einen Ausgang pro unterschiedlichem Zeichen.

Betrachten wir die Strings  $S = \{“aba”, “abab”, “b”, “bc”, “bac”, “baca”\}$

Der dazugehörige Trie sieht folgendermaßen aus:

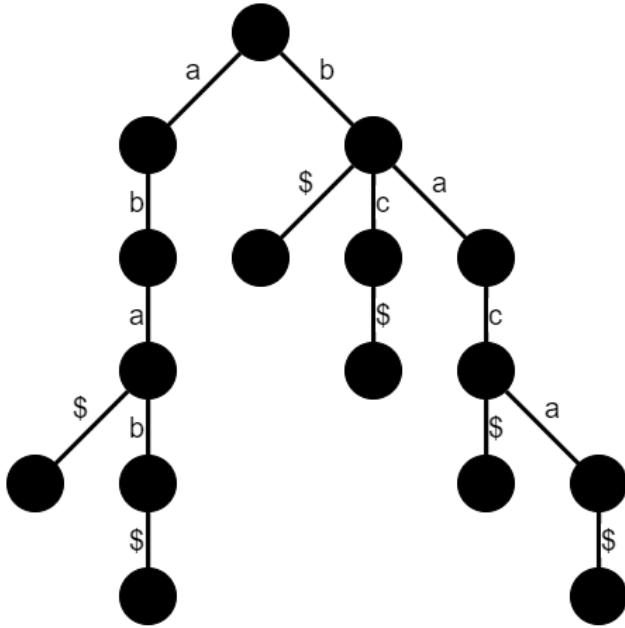


Abb. 17: Trie

Wie man erkennen kann, haben einige Knoten sowohl *in* als auch *out-degree* = 1. Diese sind eigentlich nicht nötig und können daher in einen Knoten komprimiert werden. Die Bedingung, dass verschiedene Kanten keinen gemeinsamen Prefix haben, ist natürlich trotzdem gegeben.

Es entsteht der sogenannte *Radix Tree*, oft auch kompakter Trie genannt:

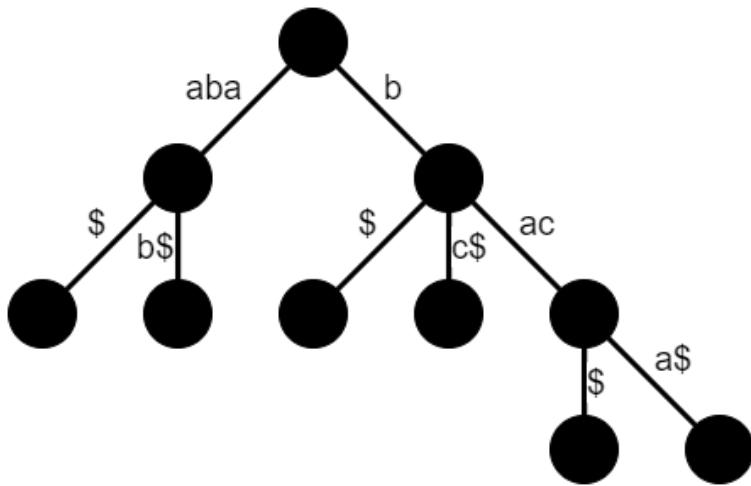


Abb. 18: Radix Tree

Diese Komprimierung macht speziell bei Filenamen Sinn, da diese oft einen gemeinsamen Prefix besitzen - beispielsweise `ic_menu_` bei Android Projekten.

**4.1.1.2 Transformation zu einem binären Baum [L]** In der “klassischen” Version, würde jeder Knoten des Radix Trees eine Liste (oder ähnliches) mit Referenzen zu den Child-Knoten speichern. Um diesen Overhead zu vermeiden, entschieden wir uns dafür, den Baum in einen binären Baum zu transformieren. Dazu speichert jeder Knoten einerseits eine Referenz zu dem Knoten darunter - und einen Verweis auf den nächsten Knoten “der selben Ebene”. D.h. die Liste wird implizit dargestellt, indem jeder Knoten nur einen next Pointer speichert. Es wird die Information “was” die Kante repräsentiert dabei auch in den Knoten verschoben - somit kann man sich zusätzlich den “künstlichen” Root Knoten sparen:

Am Beispiel von vorher:

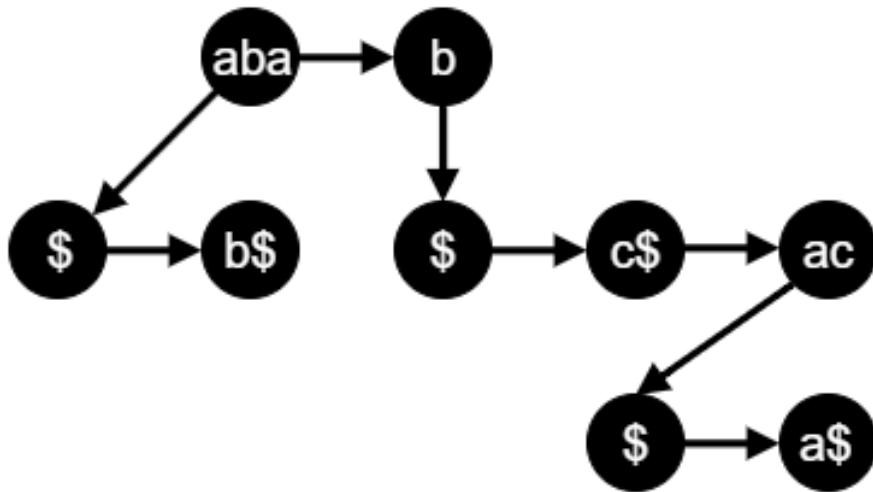


Abb. 19: Binärbaum Darstellung

Natürlich stellt sich immer noch die Frage wie man von einem Knoten den richtigen Child Knoten findet. Wie auch im “klassischen” Fall werden alle Kinder durchiteriert, bis der Richtige gefunden ist.

An dieser Stelle wären Datenstrukturen wie beispielsweise eine Skip-List anwendbar. Doch da die Anzahl an verschiedenen Kindern durch die Größe des Alphabets  $\Sigma$  begrenzt ist, macht es praktisch nicht viel Unterschied. Daher haben wir uns hier für Speicher über Performance entschieden.

#### 4.1.1.3 Suche [L]

Angenommen wir haben den Radix Tree bereits aufgebaut, wie kann ein String darin wieder gefunden werden?

Angenommen man befindet sich bei einem gegebenen Knoten  $n$ . Dann gibt es nur 3 Möglichkeiten: Gefunden, eine Ebene tiefer (child) - oder zum nächsten Knoten derselben Ebene (next). In diesem Sinn vereinfacht die Transformation zu einem binären Baum auch die Implementierung.

Sei  $k$  die Länge des Prefixes, den der Such-String  $s$  mit Label  $l$  der Kante zu *child* gemeinsam hat.

- $k = 0$   
Präfixe stimmen nicht überein - fortfahren beim Knoten *next*
- $k = |s|$   
Gefunden, Suche erfolgreich beenden
- $k = |l|$   
Richtiges Kind gefunden - bei *child* fortfahren
- Sonst  
Da  $k > 0$ , aber keiner der vorigen Fälle zutrifft, kann  $s$  nicht in der Datenstruktur sein. Das folgt daraus, dass die Ausgänge paarweise einen longest common prefix von 0 haben.

Da es sich bei dieser Rekursion offensichtlich um eine Tail-Recursion handelt, kann diese iterativ implementiert werden.

Ein Ausschnitt unserer Implementierung:

```
int l=strlen(s);
node *n=root;
while (n)
{
    int k=compare(n->val ,n->len ,s ,l );
    if (k==l)
    {
        // string gefunden
        ...
        return ..;
    }
}
```

```

if  (k==0)
    n=n->next ;
else if  (k==n->len )
{
    n=n->child ;
    s+=k ;
    l-=k ;
} else
    break ;
}
// nicht gefunden

```

Wie man erkennt, sind C-Strings für diese Zwecke sehr praktisch.

Als Zeitkomplexität ergibt sich somit  $O(|s| \cdot |\Sigma|)$ , wobei es vermutlich in den meisten Fällen nur ein Bruchteil von  $|\Sigma|$  sein wird. Auf jeden Fall ist die Komplexität nicht von den Anzahl an Einträgen in der Datenstruktur abhängig, wodurch eine schnelle Such-Performance auch bei einer hohen Anzahl an Dateien gegeben ist. (Wobei natürlich bei mehr Dateien der Konstante Faktor immer mehr an  $|\Sigma|$  "wandert".)

**4.1.1.4 Einfügen [L]** Das Einfügen ist sehr ähnlich dem Suchen, mit dem Unterschied, dass beim "nicht-finden" ein neuer Knoten erstellt werden muss. Aber auch, dass eventuell eine Kante in zwei Teil "gesplittet" werden muss, sodass die Ausgänge nach wie vor keinen Prefix gemeinsam haben.

D.h. falls wir an einen nicht vorhandenen Knoten gelangen, wird ein neuer Knoten angelegt, dessen Kante den restlichen String der eingefügt werden soll als Label hat.

Der Split Fall tritt ein, sobald wir nur einen Teil des Kanten-Labels matchen (der Fall, wo der Find-Algorithmus schon abbrechen konnte). Statt das Einfügen abzubrechen muss ein neuer Knoten erstellt werden und zwischen den Aktuellen eingefügt werden.

Führt man ein Insert von "abcd" auf den Baum von vorher aus:

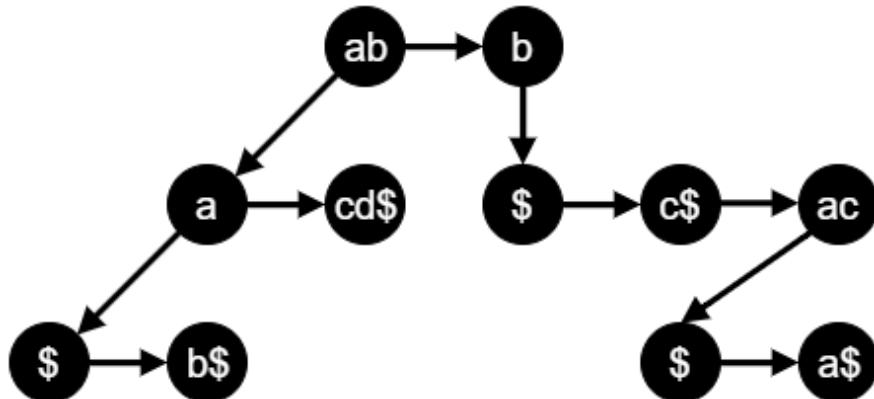


Abb. 20: Split

Wie man erkennt wurde der Knoten “*aba*” zu “*ab*” und “*a*” gesplitted. Dieses Splitten ist offensichtlich nur das “Umhängen” von ein paar Pointern und somit  $O(1)$ .

Daher ist die Komplexität somit die gleiche wie beim Suchen  $O(|s| \cdot |\Sigma|)$ .

**4.1.1.5 Löschen [L]** Das Löschen ist auch sehr ähnlich dem Such- sowie Insert-Algorithmus. Anstatt einen Knoten zu splitten muss hier aber gejoined werden.

D.h. nachdem ein Knoten gelöscht wird, hat der Parent eventuell einen *in* und *out-degree* von 1 - und unsere Struktur entspricht nicht mehr der eines Radix Trees. Somit werden diese beiden Knoten gejoined, was genau die Gegenoperation des Splittens darstellt:

Wieder aufbauend auf dem vorigen Baum, hier das Ergebnis nach dem Entfernen von “*bac*”:

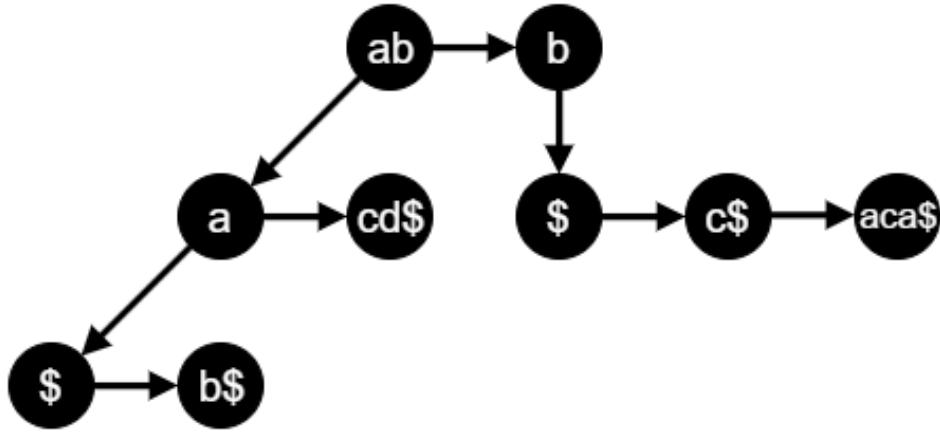


Abb. 21: Join

Das Leaf für “*bac*”, also der Knoten mit Label “\$” wurde gelöscht. Doch dadurch hätte der Knoten “*ac*” nur mehr 1 Kind und somit werden “*ac*” und “*a\$*” zu einem Knoten gemerged. Auch das ist nicht mehr als ein paar Pointer neu zu setzen,  $O(1)$ .

Also auch hier ist die Komplexität dieselbe wie beim Suchen und Einfügen  $O(|s| \cdot |\Sigma|)$ .

#### 4.1.2 Infix-Matching [L]

Nun haben wir eine Datenstruktur, welche eine Menge an Strings verwalten kann - und effizient jene findet die mit dem Prefix übereinstimmen.

Um auch das Infix-Maching zu Unterstützen (*matchen* siehe 4.1), muss der Such String einfach mehrere Male eingefügt werden. Z.B.: reicht es den String “Google Chrome“ einmal ganz und einmal als “Chrome“ einzufügen, um unserer Definition von *matchen* zu genügen.

Das wirkt zwar auf den ersten Blick vielleicht wie eine Zeit und/oder Speicherverschwendung, hat bei näherer Betrachtung aber auch viele Vorteile:

- Die meisten Dateinamen werden nur - wenn überhaupt - 2 bis 3 mal eingefügt
- Die Datenstruktur bleibt so simpel wie sie ist.  
Komplexere Datenstrukturen die auch Infix matching supporten sind zusätzlich schwierig zu ändern. So ist ein Suffix Tree zwar zeit und speichereffizient - es kann aber nichts mehr davon entfernt werden
- Gemeinsame Präfixe werden sowieso komprimiert  
Wir versuchen die Strings so zu splitten, wie ein User vielleicht auch danach suchen würde. Z.B. macht es bei “*Google Chrome*” keinen Sinn “*ogle Chrome*” einzufügen, da dies keiner sucht. Dadurch haben die einzufügenden Strings meist einen “lesbaren” Prefix, welche zusätzlich oft gleich sind - und diese gemeinsamen Präfixe werden vom Radix Tree sowieso in eine Knoten komprimiert.

#### 4.1.3 Priorität [L]

Somit haben wir eine Datenstruktur mit der effizient alle Strings/Files die *matchen* gefunden werden können. Wie kann man nun schnell jene  $k$  Elemente mit maximaler Priorität finden?

Für Query Probleme wie diese ist die erste Idee klarerweise ein Segment tree - und falls dies nicht funktioniert ein Treap.

---

#### Kurzer Überblick zu Segment trees und Treaps

Ein Segment tree ist ein perfekter binärer Baum, den man üblicherweise über ein Array aufbaut. D.h. die Blätter des Baums repräsentieren das eigentliche Array - und die Knoten darüber jeweils die Vereinigung der beiden Knoten darunter. So ein Baum hat offensichtlich  $\leq 2^{\lceil \log_2 n \rceil + 1} - 1$  Knoten (in den meisten Implementierungen sogar exakt so viele), wenn das Array zuvor  $n$  Elemente hatte. Dauert das mergen von zwei Knoten also  $O(f(n))$  kann man also einen Segment tree in  $O(f(n) \cdot n)$  Zeit aufbauen (in den meisten Fällen also einfach  $O(n)$ ).

Mit diesem Baum können dann effizient Range Queries und Range Updates ausgeführt werden. Dies funktioniert dadurch, dass man die Vereinigung von  $O(\log n)$  Knoten im Baum bildet, indem man “links” und “rechts” am “Rand” des abzufragenden Intervalls nach unten traversiert. Die Teile dazwischen sind dann einfach das Rechte (bzw. im “Rechte-Fall” das Linke) Kind der Knoten. Dadurch, dass der Baum  $O(\log n)$  hoch ist, funktionieren diese Abfragen in  $O(\log n)$ .

Für Updates funktioniert es analog, aber statt alle Knoten des Subtrees sofort neu zu berechnen, was zu linearer Laufzeit führen würde, markiert man die betroffenen Knoten nur zum Update (lazy update). Im Falle einer Abfrage wird einfach nach unten propagiert.

Mit Techniken wie Heavy-Light Decomposition kann man Segment trees (bzw. jede Datenstruktur für Operationen auf Arrays, somit auch Treaps) auf Bäume anwenden.

Treaps (= [Binary Search] Tree + Heap) sind randomisierte binäre Suchbäume, die nicht nur einen Key, sondern auch eine Priorität für jeden Knoten verwalten. Der Baum ist so aufgebaut, dass sich einerseits ein BST über die Keys, aber andererseits ein (meistens max) Heap über die Prioritäten bildet. Stellt man diese Paare als Punkte ( $x/y$ ) in der Ebene da, nennt man den entstehenden Baum Cartesian Tree.

Man kann beweisen, dass sich bei zufällig gewählten Prioritäten eine durchschnittliche Höhe von  $O(\log n)$  ergibt.

Diese Datenstruktur zeigt ihr volles(?) Potential aber erst, wenn man implizite Keys verwendet (Implizit Treap). Anstatt für jeden Knoten einen Key zu verwalten, speichert man nur noch dessen Priorität. Als Key verwendet man die Position im Baum (d.h. der Index des Knotens, falls man eine Inorder-Traversierung des Baums durchführt. Oder für die Implementierung brauchbarer:  $1 + \text{Größe vom linken Teilbaum des aktuellen Knoten} + \text{Key des ersten Parent Knoten}$ , von dem dieser Knoten im rechten Teilbaum liegt (bzw. 0 falls so ein Knoten nicht existiert)). Dadurch, dass die Keys also dynamisch sind, kann man implizite Treaps auch als dynamische Arrays verstehen. Beim Entfernen eines Knotens (bzw. analog einer Position im Array), werden “automatisch” alle Indizes rechts davon “um eins nach links verschoben”.

Wie bei Segment trees kann man in den Knoten jeweils Zusatzinformationen speichern, welche sich aus Vereinigungsmenge der beiden Kindern und des Knoten selbst ergibt. Das erlaubt Range Queries und Updates (auch hier wieder mit lazy propagation). Diese werden aber im Vergleich zum Segment tree meistens mithilfe der Join und Split Operationen realisiert (die offensichtliche Rekursion in  $O(\log n)$ ).

Dadurch, dass die Keys implizit sind, können auch Funktionen wie “drehe das Interval  $[l,r]$  um” in  $O(\log n)$  Zeit realisiert werden (in dem Fall einfach left/right Pointer tauschen).

Treaps haben aber im Vergleich zu Segment trees einen höheren konstanten Faktor (sind natürlich um das Mächtiger) und sind nicht so cache-effizient. Deshalb ist ein Treap meist die “zweite Wahl”.

---

Wie sich schnell herausstellt, kann ein Segment tree hier nicht angewandt werden (wie etwa den Tree in eine Euler Tour zu zerlegen und darüber einen Segmenttree aufzubauen).

Doch die Idee für Range Queries, wie sie aus Segment trees bekannt ist, kann in abgewandelter Form angewendet werden - und zwar ähnlich wie bei impliziten Treaps:

Man speichert einfach pro Knoten die Ergebnismenge für diesen Subtree. Diese kann als Vereinigungsmenge der Kinder sowie des Knoten selbst berechnet werden (in diesem Sinn auch die Ähnlichkeit zum impliziten Treap).

Das ist nur dadurch möglich (speichermäßig), dass wir  $k \leq 20$  beschränkt haben. Jeder Knoten speichert ein Array der  $k$  Strings/Files mit höchster Priorität in diesem Subtree. Diese können dann in  $O(k)$  mit dem Parent gemerged werden.

Hier lohnt sich die Komprimierung des Radix Trees ein weiteres mal, da auch wirklich nur Knoten mit “verschiedenen” Subtrees betrachtet werden.

Somit braucht implementierungsmäßig nur am Ende des rekursiven Aufrufs neu gemerged werden, wie es beim Code von Segment trees u.ä. üblich ist.

Dadurch ist beim Suchen das Ergebnis schon praktisch “vorberechnet” und braucht nur noch zurückgegeben werden - die Komplexität bleibt also  $O(|s| \cdot |\Sigma|)$ .

Beim Einfügen und Löschen muss neu gemerged werden, somit also um einen Faktor  $k$  langsamer. Doch wieder aus dem Grund dass  $k \leq 20$  ist das Vertretbar:  $O(|s| \cdot |\Sigma| \cdot k)$ .

Für die Implementierung ist noch auf eine Sache zu achten: Da die Strings auf mehrere Arten eingefügt werden (siehe 4.1.2) muss man beim Mergen beachten, dass zwei verschiedene Subtrees das selbe File liefern könnten.

Wir haben das damit gelöst, dass zusätzlich noch eine ID gespeichert wird - und somit “gleiche” Ergebnisse erkannt werden können.

Da die einzelnen Mengen keine doppelten Elemente enthalten, bleiben nach dem Mergen von zwei Mengen mit  $a$  und  $b$  Elementen mindestens  $\max(a, b)$  davon übrig. Damit ist unsere Anforderung wieder erfüllt.

**4.1.3.1 Add Priority [L]** Dadurch kann jetzt auch  $\text{addPriority}(s, \Delta)$  realisiert werden:

Das Hinzufügen (oder Entfernen) von Priorität funktioniert fast analog zum Suchen. Unterschied ist, dass natürlich bei den entsprechenden Leafs die Priorität um  $\Delta$  inkrementiert werden muss. Zusätzlich ist es daher auch notwendig, dass am Ende der Rekursion wie beim Einfügen/Löschen neu gemerged wird,  $O(|s| \cdot |\Sigma| \cdot k)$ .

#### 4.1.4 Rename [L]

Somit stellt sich noch die Frage wie das Umbenennen realisiert werden kann. Fügt man beispielsweise die Strings “`/somewhere/folder1`”, “`/somewhere/folder1/file1`”, “`/somewhere/folder1/file2`”, dann soll beim Umbenennen von “`folder1`” nicht jedes File darin entfernt und neu eingefügt werden müssen.

Um dieses Problem zu lösen, sind wir wie folgt vorgegangen: Neben des eigentlichen Suchbaums wird auch ein zweiter Baum verwaltet, welcher die Ordnerstruktur abbildet (Knoten repräsentieren Ordner). Zudem speichert jedes File im Suchbaum in den Metadaten nicht den eigentlichen Pfad, sondern einen Pointer zu einem Knoten im zweiten Baum, welcher den dazugehörigen Ordner darstellt.

Um herauszufinden, in welchem Ordner eine gegebene Datei (Metadata) liegt, reicht es den Parent-Pointern des zweiten Baums bis zum Root zu folgen und den Pfad zu konstruieren. Beim Umbenennen genügt es nun den “Namen” des Ordners im zweiten Baum zu ändern, weil ja keine der Metadaten den eigentlichen Pfaden speichert sind und dieser erst zur Abfrage-Zeitpunkt aufgebaut werden.

Neben der Möglichkeit des effizienten Umbenennen, spart dieser Ansatz auch Speicher: Anstatt einen Pfad wie “`/Users/Florian/Desktop/....`” für jedes File am Desktop zu speichern werden nur 3 Knoten im zweiten Baum erstellt. Da der Ordnername aber auch suchbar ist, wird dieser in beiden Bäumen abgebildet (und beim Rename auch für beide geändert).

Durch diese Lösung ist also das Verschieben möglich, ohne die eigentlichen Files in dem Ordner zu “verändern”: Es reicht den jeweiligen Knoten im zweiten Baum “umzuhängen”. Da das eigentlich Objekt dasselbe bleibt, werden implizit auch die dazugehörigen Dateien verschoben.

Zusätzlich bleibt die Zeitkomplexität der anderen Operationen die selbe.

Wird eine Datei umbenannt, die sich nicht im Suchbaum befindet, funktioniert dies also in  $O(|s|)$ , asymptotisch optimal. Ist die Datei im Suchbaum, ist es zusätzlich mit einem rename (remove + insert) verbunden, also  $O(|s| \cdot |\Sigma| \cdot k)$ .

#### 4.1.5 Save [L]

Um das Dateisystem nicht jedes mal neu indizieren zu müssen, soll der Inhalt der Datenstruktur natürlich auch persistiert / abgespeichert werden können.

Die Save-Methode muss hierzu die eingefügten Strings, inklusive deren Priorität, speichern. Doch dies ist technisch keine Schwierigkeit und nur eine Frage der Tiefensuche.

Es stellt sich nur die Frage in welchem Format gespeichert werden soll.

Wir haben uns hierbei dafür entschieden, einfach die Strings Zeile für Zeile in eine Datei zu schreiben. Und am Ende der Zeile, jeweils mit ; getrennt, die dazugehörigen Prioritäten.

Dadurch werden zwar viele Pfade (beispielsweise “/Users/Florian/Desktop/...”) doppelt und dreifach gespeichert, aber es ist einfacher zu verwalten und zu implementieren. Denn ob auf der Festplatte ein paar MB mehr oder weniger verbraucht sind, macht im Endeffekt keinen Unterschied (im Gegensatz zum RAM).

#### 4.1.6 Load [L]

Zu guter Letzt gilt es noch die gespeicherten Daten wieder zu laden und die Datenstruktur aus dem gespeicherten Stand wiederherzustellen. Durch unser einfaches Speicherformat geht das Einlesen problemlos vonstatten:

Strings Zeile für Zeile einlesen und mit der Priorität wieder ein Insert aufrufen.

#### 4.1.7 Implementierung [L]

Ausgerüstet mit einer theoretischen Lösung für die Such-Engine galt es nun diese auch in Code umzusetzen. Dadurch, dass sowohl die Mac Version wie auch James für Windows auf dem Such-Algorithmus aufbauen, stellte sich ein weiteres mal die Frage welche Technologie wir anwenden.

Für unsere Anforderungen suchten wir eine Programmiersprache die auf beiden Plattformen unterstützt wird und sich verhältnismäßig leicht in unsere bestehenden Programme einbinden lässt. Da wir den Punkt Performance mit berücksichtigen mussten und die Einbindung

mittels eines Wrappers in Swift/C# nicht allzuschwer schien (Abschnitt 4.2), setzte sich C++ gegenüber Java und diverser anderer Skriptsprachen durch.

Für kleine Details die sich unter Windows bzw. am Mac unterscheiden eignen sich in C++ der \_WIN32 Macro, der unter Windows definiert ist. So kann man noch etwas plattformspezifischen Code hinzufügen.

Bei uns ist das für den Separator der Dateinamen notwendig, da dieser unter Windows ein Backslash ist und unter OS X ein Slash darstellt.

```
#ifdef _WIN32
    _declspec(dllexport) const char *SearchEncodeLower(const char *s);
    #define strcasecmp _stricmp
    const char separator='\\';
    #pragma warning(disable : 4996)
#else
    const char separator('/');
#endif
```

Abb. 22: Verwendung des \_WIN32 Macros

**4.1.7.1 Umlaute [L]** Bei der Implementierung stießen wir aber auf ein Problem, dass Filename auch diverse Sonderzeichen - oder auch Zeichen anderer Sprachen enthalten können. Folgendes ist beispielsweise ein akzeptabler Dateiname am Mac:



ðƒʃ,øf@Δ€øΣ~雪

Abb. 23: Filenamen am Mac

In Sprachen wie Swift und C# ist dies kein Problem, da deren Strings solche Zeichen “out of the box” unterstützen. In C++ hingegen leider nicht.

Unser erster Ansatz dieses Problem zu umgehen war, statt 1 Byte pro Zeichen 2 zu verwenden. Doch das bringt mehr Nachteile als es wirklich hilft:

- Speicherverbrauch verdoppelt
- Manche Zeichen benötigen mehr als 2 Bytes (z.B. Chinesische und Japanische Zeichen)

Daher wurde diese Idee schnell verworfen.

Unser zweiter Ansatz war es dann diese “nicht unterstützen Zeichen” einfach zu ersetzen, beispielsweise aus ä ein ae zu machen. Doch hierbei ist das Problem, dass es weit mehr Sonderzeichen als ä, ö und ü gibt. Dies fängt schon bei französischen Axons (á, é, ...) an. Weiters müsste für jedes dieser Zeichen dann ein eigener Sonderfall hinzugefügt werden. Also ist auch dieser Lösungsansatz nicht akzeptabel.

Doch die Idee, Zeichen zu ersetzen, scheint dennoch nicht schlecht zu sein, weil so im eigentlichen Algorithmus keine Änderungen gemacht werden müssen.

So kam der Ansatz eine bereits vorhandene Kodierung zu verwenden. Denn wir sind ja nicht die Ersten mit diesem Problem. Es bietet sich z.B. ein HTML Encoding aus folgenden Gründen an:

- Kompatibel mit “allen“ Zeichen
- “Normale“ Zeichen brauchen weiterhin nur 1 Byte. Sonderzeichen benötigen mehrere, aber diese sind sowieso eine Ausnahme
- Einfach zu verwenden

Da das Encoding sowieso eine Ebene höher passieren muss (über der C++ Schicht), kann z.B. die .Net und die Cocoa Implementierung ein anderes Encoding verwenden

Zuerst wird der String encoded, sodass nur noch Strings des URLFragmentAllowedCharacterSet darin enthalten sind, dann wird mit der UTF8String Methode ein C-Style String davon erzeugt.

In Objective C++ Wrapper encoden wir also wie folgt:

```
[ [ s stringByAddingPercentEncodingWithAllowedCharacters:
    [ NSCharacterSet URLFragmentAllowedCharacterSet ] ] UTF8String ]
```

Und unter Windows:

```
WebUtility :: UrlEncode( s )
```

Das ganze Encoden bringt jedoch auch ein Problem mit sich: Man kann den encodeten String nicht mehr (oder zumindest nicht mehr so einfach) toLower-/toUppercase machen. Dies ist jedoch nötig, da die Suche ja case-insensitive ist, die Suchergebnisse aber sehr wohl auf Groß-/Kleinschreibung achten sollen.

Dies könnte einerseits damit umgangen werden, indem man die Files einfach lower/uppercase in den Index gibt, und nachträglich die richtige Groß-/Kleinschreibung wieder herstellt. Doch das würde einen (langsam) Syscall an das OS benötigen - um herauszufinden wie es denn wirklich geschrieben wird (IO). Weiters funktioniert es auf case-insensitiven Dateisystemen nicht (unter OS X beispielsweise das “Mac OS Extended (Case-Sensitive)” Format).

Eine andere Möglichkeit wäre es, für den Stringvergleich eine *CompareCaseInsensitive* Methode - oder ähnliches - als Callback zu verwenden. Doch damit können keine beliebigen Substrings verglichen werden, weil man sich nie sicher sein kann welche Zeichen “wirklich” zusammengehören, nachdem beim Encoden Sonderzeichen auf mehrere aufgeteilt werden.

Aus diesem Grund entschieden wir uns dann dazu die Strings doppelt zu übergeben, einmal lowercase und einmal in ihrer Originalform: Die lowercase Variante wird zum Einfügen in die Datenstruktur verwendet, damit case-insensitiv lowercase gesucht werden kann. In den Metadaten hingegen wird der Originalstring gespeichert. Das erzeugt zwar beim Laden einen Spezialfall, dieser kann aber einfach gelöst werden:

Um nicht beide Versionen zu persistieren, speichern wir nur die originale Version ab: Doch zum Einfügen wird auch die lowercase Version benötigt. Um dies zu lösen, erstellten wir im jeweiligen Wrapper eine Callback Methode, die für einen gegebenen String die kodierte lowercase Variante zurückgibt. Mehr dazu in der Wrapper Kapitel (4.2).

## 4.2 Suchengine-Wrapper [L]

Dadurch, dass wir die Search-Engine also in C++, aber die eigentliche Applikation in C#/Swift programmierten, mussten wir den nötigen Wrapper schreiben, sodass diese Sprachen miteinander kommunizieren können.

In beiden Fällen ist es nötig einen Wrapper zu schreiben, welcher die Schnittstelle darstellt. In diesem Wrapper werden auch die vorher erwähnten Callback Methoden implementiert.

### 4.2.1 Implementierung unter Windows [M]

Unter Windows unterscheidet man grundsätzlich zwei Arten von Code:

**Managed Code** Sprachen wie Visual Basic und Visual C# oder Visual C++ werden nicht direkt zu Maschinencode kompiliert, sondern lediglich zu einer sogenannten Intermediate Language (IL). Dieser Bytecode läuft in der Common Language Runtime (CLR). Durch diese Abstrahierung mittels der CLR läuft der Code eine Schicht über dem Maschinencode und ist dadurch etwas langsamer und dafür nicht maschinenabhängig. Weiters bringt diese Schicht einige Vorteile mit sich: U.a. Memory Management inklusive eines Garbage Collectors oder auch die Möglichkeit des Just in Time “JIT” Kompilierens. Wenn diese Assembly nun in der Runtime läuft, kann man sagen, das Programm wird von der Runtime verwaltet, daher die Bezeichnung managed Code.

**Unmanaged Code** Unmanged Code (u.a. C++) wird direkt zu Maschinencode kompiliert. Dieser Code läuft daher nicht in der CLR und erhält dadurch auch nicht deren Vorteile. Deshalb muss der/die ProgrammiererInnen selbst für das Speichermanagement Sorge tragen in dem er/sie rechtzeitig die zuvor angefragten Ressourcen wieder freigibt. Bei Unachtsamkeiten kann es daher schnell zu sogenannten Memoryleaks kommen und da unsere Suche eigentlich immer läuft können solche Speicherlecks schnell gravierend werden.

Da nun der geteilte Code in unmanaged C++ Code vorliegt, brauchen wir unter Windows eine Möglichkeit, diesen auch von Managed Code aufzurufen. Um dies zu erreichen, gibt es zwei Möglichkeiten:

- **Importierung einer unmanaged DLL:** Dieser Ansatz ist der Einfachste. Man erzeugt einfach aus dem Code eine DLL. Dabei ist auf die Korrektheit der dazugehörigen .h Datei zu achten, damit die DLL richtig erzeugt wird. Diese kann dann ganz einfach in C# mittels der C# DLL Import Syntax eingebunden werden:

```
[DllImport (" [path to dll] " )]
public static extern [return params] [Methodenname] ([params]);
```

Wie man dabei sehr gut erkennen kann, ist es nötig jede Methode einzeln zu importieren, jedoch stellt dies einen überschaubaren Aufwand dar. Dieser Ansatz wird für kleinere C++ Codeteile empfohlen, da Strukturen sowie Klassen über diesen Ansatz nicht verwendet werden können.

- **Importieren einer managed DLL:** Visual C++ ist eine von Microsoft entworfene Sprache des .NET Frameworks mit einer großen Anbindung an C++. Sprich es ist eine .NET Sprache mit C++ Syntax, welche auch einige Funktionen zur Kommunikation mit unmanaged C++ bereitstellt. Dadurch stellt Visual C++ die einzige Sprache im .NET Framework dar, welche unmanaged Code aufrufen kann. Als Bestandteil des .NET Frameworks ist es ein Leichtes dieses von C# aus aufzurufen. Daher stellt das eine ideale Möglichkeit dar, unmanaged C++ Code in C# zu verwenden. Durch die zusätzlich eingeführte Abstrahierungsschicht, bietet es auch die Möglichkeiten weitere Implementierungen in dieser zu verstecken, was natürlich in eine bessere Übersicht resultiert. Da unser geteilten Code auch Klassen und Strukturen sowie mehrere Methoden besitzt, wählten wir diese Technik.

Daher erstellten wir zwei zusätzliche Projekte. Eines für die unmanaged *search-engine.cpp* Datei und eines für unsere Visual C++ Wrapper (*SearchEngineWrapper.cpp*). Der Wrapper inkludiert die *search-engine.cpp* Datei und wird zu einer DLL kompiliert, welche daraufhin unter C# mittel Referenz eingebunden wird und von der *SearchEngine.cs* aufgerufen wird. Diese Abhängigkeiten möchten wir nun anhand eines Beispieles näher beschreiben:

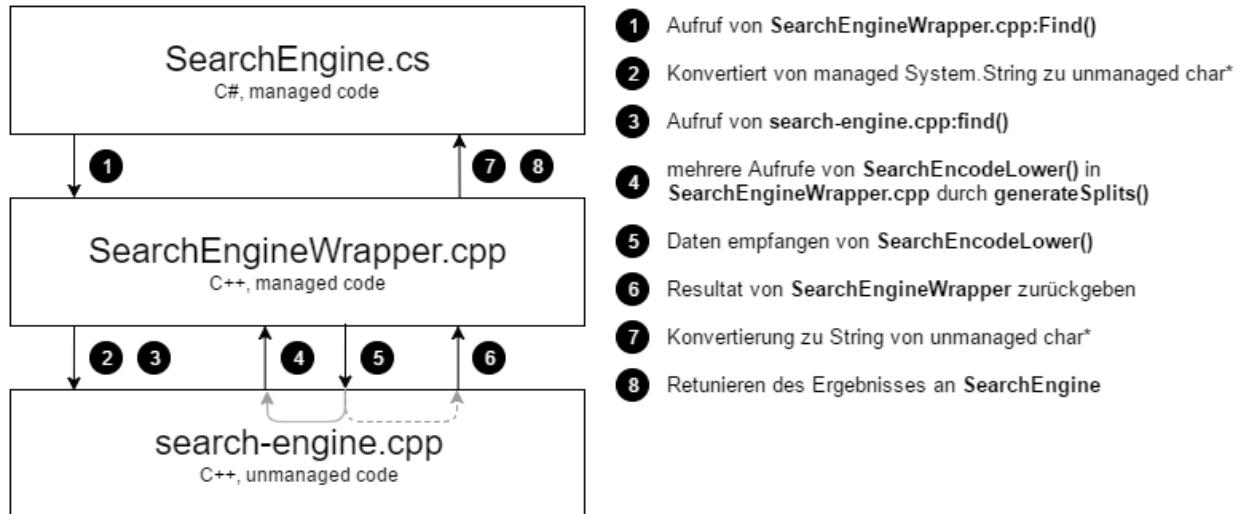


Abb. 24: Wrapper zur Überbrückung von unmanaged Code zu managed Code

Wie bereits aus der Grafik ersichtlich, muss der SearchEngine Wrapper die Methodenargumente von Managed *System.String* zu einem unmanaged C++ *char\** konvertieren. Diese Funktionalität erreichen wir mit der in .NET verfügbaren *Marshal::StringToHGlobalAnsi()*

Methode. Dies wäre zwar auch über Umwege mittel Hilfsklassen des .NET Marshal Namespaces möglich, würde für uns aber nicht reichen: Spätestens wenn *search-engine.cpp* SearchEncodeLower aufrufen muss, stellte sich heraus, dass wir mit der Wahl zur Einbindung von managed DLLs und die dadurch erhaltene zusätzliche Abstrahierungsschicht, auf die richtige Variante gesetzt haben. Der Aufruf von gemanaged Methoden von unmanaged C++ Code ist über DLL-Imports nämlich nicht möglich.

Um aus der *search-engine.cpp* unsere *SearchEncodeLower()* Methode aufrufen zu können, braucht es ein paar Schlüsselwörter. Die Methode muss als DLL-Export definiert werden. Daher sieht die Methodendefinition der *SearchEncodeLower()* Methode wie folgt aus:

```
--declspec(dllexport) const char *SearchEncodeLower(const char *s)
```

Die genau gleiche Definition wird daraufhin auch in *search-engine.cpp* eingetragen. Da dieser Eintrag Windows spezifisch ist, wird dieser innerhalb des #ifdef \_WIN32 eingetragen (siehe Abbildung 22).

#### 4.2.2 Implementierung unter OS X [L]

Um von Swift aus C++ Code zu verwenden muss man einen Wrapper in Objective C++ schreiben. Objective C++ ist eine Variante von Objective C, welche auch (einen Großteil von) C++ erlaubt (ähnlich wie C++ “einen Großteil” von C erlaubt). Und nachdem Swift mit Objective C umgehen kann (Apple will selbstverständlich kompatibel bleiben), ist es so möglich C++ Code zu verwenden.

Um generell Objective C(++) Code in Kombination mit Swift zu verwenden, braucht man ein sogenanntes BridgingHeader File. Das ist eine .h Datei, welche alle Objective C Headers angibt (d.h. inkludiert), welcher kompiliert und zum Swift Projekt gelinkt werden sollen. Im Bridging Header darf noch *kein* C++ File inkludiert werden.

Im Fall unserer SearchEngine haben wir also eine BridgingHeader.h Datei angelegt, welche unter anderem (neben anderen Objective C Libraries die wir verwenden) - SearchEngine-Wrapper.h inkludiert (natürlich alles in einen #ifndef).

Auch im SearchEngineWrapper.h war es uns noch nicht möglich, die eigentliche C++ Datei zu inkludiert. Dies funktioniert erst im der .m Datei (Objective C). Interessanterweise benötigt man als Forward-Declaration keine “*class SearchEngine*”, sondern ein “*struct SearchEngine*”. Vermutlich hat auch dies den Hintergrund, dass die .h Dateien vom Objective C Kompilier bearbeitet werden - und nur die .m Dateien dann wirklich als Objective C++ behandelt werden (das *struct* Keyword gibt es in Objective C, im Vergleich zu *class*).

In *SearchEngineWrapper.h* deklarierten wir also ein (zwangsläufig von *NSObject* erbendes) *Object*, *SearchEngineAPI*:

```
@interface SearchResult : NSObject
@property NSString *file;
@property int priority;
@end
```

```
@interface SearchEngineAPI : NSObject {
    struct SearchEngine *search;
    NSArray<SearchResult*> *result;
}
```

Wie man erkennt, ist auch eine Wrapper Klasse für die eigentlichen Suchergebnisse nötig. Um das Ergebnis Array nicht jedes Mal neu allokieren zu müssen, erstellen wir dies nur einmal und überschreiben die Daten.

Die eigentlichen Methoden werden dann im Standard Objective C Style deklariert. Z.B.:

- (void) insert: (NSString\*) s prio: (int) prio;
- (NSArray<SearchResult\*>\*) find: (NSString\*) s;
- (void) addPriority: (NSString\*) s delta: (int) delta;

Im .m File (SearchEngineWrapper.m) sind diese Methoden dann zu implementieren:

```
@implementation SearchEngineAPI
- (id) init
{
    .....
    return self;
}

...
- (void) insert: (NSString*) s prio: (int) prio
{
    search->insert(SearchEncode(s), prio);
}

...
@end
```

Wie man sieht, wird z.B. vor dem insert der String noch encoded. Diese Methode ist auch im Wrapper definiert. Jedoch nicht im SearchEngineAPI Objekt, sondern außerhalb, sodass der C++ Code diese auch verwenden kann. Im C++ Code ist, im Vergleich zum Windows Wrapper, hierfür nichts zu ändern.

Z.B.:

```
const char *SearchEncodeLower(const char *s)
{
    NSString *str=SearchDecode(s);
    return SearchEncode([str lowercaseString]);
}
```

Also wieder komplett im C(++) Style, jedoch immer noch mit Cocoa Objekten (NSString) - und der Aufruf dieser Methoden im Objective C Style ([str lowercaseString]).

Nachdem man in den Build-Settings den Bridging Header einträgt, kann man die SearchEngineAPI Klasse wie jede andere verwenden.

## 4.3 File Watcher [L]

Dadurch, dass wir uns für eine eigene Implementierung der Suche entschieden haben, muss diese auch up-to-date gehalten werden. D.h. der konfigurierte Search-Scope sollte konsistent mit den Einträgen in unserem Suchalgorithmus sein.

Dafür gibt es sogenannte File-Watcher. Mit ihnen kann man auf Ordner “aufpassen” und bekommt Events, falls sich deren Inhalt ändert. Die jeweilige API dafür ist natürlich vom Framework abhängig. Im Folgendem finden Sie die Details zu File Watchern in Cocoa sowie dem .Net Framework.

Neben den *create* und *remove* Events ist auch ein *rename* Event wünschenswert. Und zwar damit die Priorität auch bei einem *rename* übernommen wird.

### 4.3.1 Cocoa [L]

Zu Beginn hier die Reaktion von Professor Bauer nachdem Florian ihn bzgl. File Watcher unter Cocoa fragte: “Des is jo nur ganz knapp überm Assembler.”.

Zum “Überwachen” von einzelnen Dateien gibt es die sogenannte *kqueue* - für ganze Ordner bietet Cocoa die C API *FSEvents*.

Bis Swift 2.0 war es nicht möglich in “reinem Swift” C APIs, wie eben beispielsweise FSEvents, zu verwenden. Und zwar deshalb, weil es nicht möglich war, einen C-Style Function Pointer zu deklarieren, den aber viele dieser APIs als Callback benötigen. In Swift 2.0 (2015) wurde das Problem dann behoben.

Aus diesem Grund sind einige Objective C Wrapper entstanden um Zugang zu diesen Funktionalitäten zu bieten.

Während unserer ersten “Gehversuche” war Swift 2.0 noch nicht verfügbar. Um diesen Wrapper nicht selber schreiben zu müssen - und eine Abstraktion zwischen dem “knapp überm Assembler” zu haben, entschieden wir uns also für die Library *FileSystemEvents*, welche auf Github zur Verfügung steht (<https://github.com/Eonil/FileSystemEvents>).

Damit sollte die File Watcher Problematik eigentlich gelöst sein - doch dem ist nicht so:

- Teilweise sind Events mit ItemCreated und ItemRemoved Flags markiert
- Es gibt kein *rename* Event

Ersteres stellt kein wirkliches Problem dar, weil man auch selber prüfen kann, ob das File existiert oder nicht - also ob es sich um eine *create* bzw. *remove* Event handelt.

Das fehlende *rename* Event stellte sich aber als schwieriger da:

**4.3.1.1 Rename [L]** Anstatt ein *rename* Event zu erhalten, bekommt man zwei, nicht zwingend aufeinander folgende, *remove* und *create* Events. Es gibt nur leider keine Möglichkeit herauszufinden, welches Paar nun wirklich zu einem File gehört, da den Events grundsätzlich nur der Pfad (inkl. Dateiname) beiliegt.

Wie unter Unix üblich ist ein Umbenennen nicht zwingend nur eine Änderung des Dateinamens - sondern kann auch ein Verschieben bedeuten (oder beides, z.B. “*mv /somewhere/file1 /somewhere\_else/file2*”).

Es ist also nötig festzustellen, welches Paar an Ereignissen zusammengehört, sodass der Suchbaum aktualisiert werden kann.

In den Dokumentationen ist zwar keine maximale Zeit angegeben, wieviel diese zwei Events auseinander liegen können, doch unsere Tests ergaben, dass es sich nur um ein paar wenige Millisekunden handelt.

Darauf basierend kategorisiert die Mac Version Events (Pfad  $p$ ) wie folgt:

Falls es sich um ein *remove* Event handelt, wird  $p$  inkl. eines Zeitstempels in eine Queue gelegt, welche alle Ereignisse enthält von denen man sich noch nicht sicher ist (*remove* oder *rename* Event?). Passiert innerhalb von  $\Delta$ ms nichts, wird von der Queue wieder dequeued und das Event als *remove* behandelt.

Bei einem *create* Event wird nun diese Queue konsultiert. Ist sie leer handelt es sich um ein *create* Event, da kein *remove* nahe genug stattgefunden hat. Andernfalls wird dieses *create* als zweiter Teil eines *rename* Events interpretiert und wieder dequeued.

Diese Heuristik bietet folgende Vorteile:

- Nicht aufeinander folgende Ereignisse richtig gehandelt:  
Dadurch, dass die Queue auch mehrere noch unsichere Events verwalten kann, ist es nicht nötig die jeweiligen Event Paare hintereinander zu erhalten.

- Minimaler Speicher und Laufzeit Overhead:

Man könnte natürlich auch versuchen, Metadaten zu den Dateien zu “hängen” und so festzustellen, ob diese Datei umbenannt wurde (was beispielsweise bei Dateien ohne Schreibrechte nicht funktioniert). Andererseits wäre es auch eine Idee einen “Hash” vom Dateinhalt zu berechnen (was bei leeren Ordnern nicht zuverlässig funktioniert) und diese zu matchen.

Doch all diese anderen Ansätze benötigen mehr Speicher und/oder Laufzeit als eine Queue mit Zeitstempel.

Es gilt nur noch einen passenden Wert für  $\Delta$  zu finden. Tests ergaben (Binary Search), dass  $\Delta = 50$  vollkommen reicht.

50ms scheint auch aus logischer Sicht ein plausibler Wert zu sein:

- Es ist höchst unwahrscheinlich, dass ein(e) BenutzerIn innerhalb von 50ms eine Datei erstellt und eine andere (verschiedene) Datei löscht.

- Aus “Prozessor Sicht” sind 50ms mehr als genug Zeit diese zwei Events auszulösen.

Natürlich kann ein “motivierter” User es schaffen diese Heuristik auszutricksen, wenn er das will. Doch das Einzige was er damit erreicht ist, dass eine Priorität falsch (bzw. nicht) übernommen wird.

#### 4.3.2 .Net Framework [M]

Mit Hilfe des FileSystemWatcher existiert bereits ein Wrapper für .NET, welcher die Systemevents verfolgt. Diese Implementierung besitzt mehrere Methoden. Für uns sind nur folgende Methoden relevant:

- *Create*: Diese Funktion funktioniert wie man es sich erwartet. Man erhält einfach den Pfad des neu erstellten Elements.
- *Delete*: Beim Löschen einer Datei erhält man ebenfalls einfach ein *Delete* Event, welches unter anderem den Pfad enthält.
- *Rename*: Dieses Event besitzt eine ungewöhnliche Eigenheit. Es wirft nur *Rename* Events, falls es sich dabei um eine Datei handelt. Verschiebt man jedoch einen Ordner, wird zuerst ein *Delete* und ein darauffolgendes *Create* Event geworfen. Um dieses Problem zu umgehen, mussten wir ebenfalls, wie bereits unter Cocoa, teilweise auf eine Heuristik setzen. Damit wir die beiden Ereignisse miteinander in Beziehung setzen können, mussten wir irgend eine Entscheidung treffen. Da wir das *Delete* Event erst nach vollständiger Löschung erhalten, ist es uns nicht möglich, diesen Ordner zu markieren.

Deshalb trafen wir die Annahme, dass bei einer Verschiebung eines Ordners nicht der Name des Ordners gleichzeitig geändert wird. Durch normale Nutzung des Windows Explorer ist dies auch nicht möglich. Durch systemnahe Aufrufe, u.a. Powershell und eigene Implementierung, jedoch schon. Sollten sich die Ordnernamen der beiden Events währenddessen geändert haben, kann dadurch leider die Priorität nicht übernommen werden. Dies klingt vielleicht auf den ersten Blick ziemlich gravierend, jedoch muss man auch beachten, dass ein Verschieben mit gleichzeitigem Umbenennen des Ordners in der Praxis kaum vorkommt. Selbst wenn es zu so einem ungünstigen Fall kommt, werden die Prioritäten nur geringfügig abweichen, da die Elemente des Ordners anhand des dortigen SearchScopes neu indiziert werden.

Wie bereits bei der Cocoa Implementierung setzten wir daher auch das 50ms Zeitfenster ein. Beim Löschen eines Ordners wird der Ordnername einfach in eine Queue gepusht. Kommt unmittelbar daraufhin (50ms) ein *Create* Event mit dem gleichen Ordnernamen wird nur ein *Rename* unserer SearchEngine aufgerufen. Nach 50ms wird der Ordner, welcher zuerst das *Delete* Event gelöst hatte, aus der Suche gelöscht.

## 4.4 Swift, Cocoa [L]

Dieser Abschnitt ist speziell jenen Dingen gewidmet, die in Swift/dem Cocoa Framework schwierig waren und soll keine Einführung in Swift oder das Framework darstellen. Das Kapitel soll aber nicht nur diese “Schwachstellen” (oder unser Unwissen) aufzeigen, sondern auch einige der Swift/Cocoa Vorteile präsentieren, welche uns überzeugt haben.

Generell ist zu sagen, dass das Cocoa Framework mittlerweile schon gut 25 Jahre alt ist. Es mag zwar damals seiner Zeit voraus gewesen sein, trotzdem hätte das eine oder andere Update über die Jahre einmal nicht geschadet. So entdeckt man viele Dinge, die entweder ein bisschen veraltet wirken oder einfach aus Kompatibilitätsgründen (immer noch) dabei sind. Das Framework unterscheidet sich relativ stark vom Framework für iOS, nachdem dieses erst mit der Einführung des mobilen Betriebssystems adaptiert wurde und so um einiges moderner wirkt.

### 4.4.1 Programmiersprache [L]

Zuerst einige Kommentare zur Programmiersprache selbst. Dadurch, dass Swift erst 2014 herauskam, überzeugt es mit seiner modernen Syntax und einigen Features die bei anderen Sprachen fehlen.

**Optional Types** In Swift gibt es den postfix Operator `?`, der es, ähnlich zu C#’s `int?`, erlaubt optionale Typen zu deklarieren. Im Vergleich zu C# ist es in Swift aber nicht möglich einem Objekt ohne diesem Optional Type `null` (bzw. in Objective C/Swift Vokabular `nil`) zuzuweisen. Z.B. `var text:String = nil` kompiliert nicht, dazu muss man `var text:String? = nil` schreiben.

Dadurch kann man sich an vielen Stellen sicher sein, dass ein Objekt einfach nicht `nil` ist und erspart sich so viele Prüfungen.

Gleichzeitig gibt es natürlich viele Objekte die z.B. nur am Anfang “eine Zeit lang” `nil` sind, dann aber immer einen Wert haben. Wird aber ein Objekt nicht im Konstruktor (Initialization in Swift, init Methode) initialisiert, dann muss es als Nullable deklariert werden.

Das führt dazu, dass man es beim Verwenden immer unwrappen muss:

Um auf ein nullable Objekt zugreifen zu können, schreibt man entweder `myObject?.method()` oder `myObject!.method()`.

Der `?` Operator ist bereits aus C# bekannt. Mit `!` kann man ein Objekt force-unwrappen (d.h. ohne `nil` Überprüfung).

Dadurch gibt es jetzt in unserem Source Code einige Stellen an denen zwangsläufig bei jedem Variablen Access ein `!` dazugeschrieben werden muss, da man sich einerseits sicher ist, dass dieses Objekt nicht `nil` ist, aber es andererseits nicht von Anfang an einen Wert hat.

**Custom Operators** Im Vergleich zu C# erlaubt es Swift nicht nur Operatoren zu überschreiben, sondern auch neue Operatoren zu definieren.

So kann man sich neue Prefix, Infix und Postfix Operatoren definieren. Dies funktioniert, wie man an folgendem Beispiel erkennt, auch mit “exotischen” Zeichen. Man könnte beispielsweise einen eigenen Operator definieren, der die Wurzel berechnet:

```
prefix operator √ { }
prefix func √ (x: Double) -> Double {
    return sqrt(x)
}
```

Dann wäre beispielsweise  $\sqrt{16} = 4$ .

Für Infix Operatoren kann zusätzlich assoziativ (d.h. ob sich der Operator nach links/recht “gruppiertert”) bzw. precedence (d.h. “Wichtigkeit”, wie Punkt vor Strich) angeben werden. Z.B.

```
infix operator ± { associativity left precedence 140 }
func ±(left : Int, right : Int) -> (Int, Int) {
    return (left + right, left - right)
}
```

$1 \pm 2$  liefert dann beispielsweise das Paar  $(3, 1)$ . 140 entspricht der precedence von  $+$ ,  $-$ ,  $\&+$  (Addieren mit Overflow),  $\&-$ ,  $\text{—}$  und  $\wedge$ . Auch die  $\&+$ ,  $\&-$  sowie  $\&^*$  Operatoren geben Swift, meiner Meinung nach, einen gewissen Charme.

Interessanterweise benötigt man bei der Deklaration von Infix Operatoren vor dem *func* das Schlüsselwort *infix* nicht (bzw. es ist nicht einmal erlaubt), bei prefix/postfix Operatoren ist es genau andersherum.

**Tuples** Wie auch z.B. in Python möglich, kann man mit einer eleganten Syntax mehrere Werte als Tuples verwalten:

```
var a = (1, "hello", 0.5)
```

Dieser Code würde z.B. ein Tuple aus einem Integer, String und Double deklarieren. Mit  $.0$ ,  $.1$  und  $.2$  kann auf die Werte zugegriffen werden.

**Function currying** Wie man es aus funktionalen Programmiersprachen kennt, ist es in Swift möglich, wenn auch nicht so schön wie z.B. in Haskell, Function currying zu betreiben.

Zuerst das Konzept am Beispiel von Haskell beschrieben. Da dort die Funktionsnotation (bzw. zumindest der “Pfeil”) rechts-assoziativ ist, ist  $f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  (Haskell Notation) equivalent zu  $f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ . Also ist eine Funktion  $(\text{int}, \text{int}) \rightarrow \text{int}$  (Swift-ähnliche Notation) eigentlich eine Funktion mit nur einem Parameter, die eine andere Funktion zurückgibt. Diese zweite Funktion nimmt wieder nur einen Parameter und gibt einen Integer zurück (also  $(\text{int}) \rightarrow ((\text{int}) \rightarrow (\text{int}))$ ). Das wird als Higher Order Functions bezeichnet.

Die Anwendung dieser Funktionen ist links assoziativ in Haskell, so ist z.B.  $f \ a \ b$  eigentlich  $(f \ a) \ b$ . Bei der zweiten Notation erkennt man, dass eigentlich eine neue Funktion zurückgegeben wird und diese dann mit  $b$  evaluiert wird.

Das ist aus dem Grund hilfreich, da es so für jede Funktion möglich ist, diese nur mit einem Teil der Parameter “zu verwenden” (eigentlich ausgeführt wird sie natürlich erst am Ende). Am Beispiel der *map* Funktion:

```
map (+1) [1..5]
```

*Map* benötigt eine Funktion (mit einem Parameter) welche auf jedes Element der Liste (zweiter Parameter) angewendet werden soll. Der Plus Operator benötigt zwei Zahlen. Dank Function currying liefert  $(+1)$  aber eine Funktion die nur einen Parameter benötigt (der Erste ist bereits 1) und kann daher für *map* verwendet werden. Somit liefert dieses Beispiel  $[2,3,4,5,6]$ .

Dieses Function currying ist, in ein bisschen abgewandelter Form, auch in Swift möglich. Leider muss aber dazu die Funktion anders deklariert werden:

```
func add(a: Int)(b: Int) -> Int {
    return a + b
}
```

Je nach Anzahl an Parameter würde man einfach mehr Klammerpaare hinzufügen. Jetzt wäre der Aufruf  $\text{add}(1)(b : 2)$  möglich.

Mit dieser Art von Notation ist es nun möglich zu definieren, dass z.B. für die “erste Version” der Funktion ein gegebenes Set an Parametern nötigt ist, dann ein weiteres usw. (jeder Klammerblock muss immer vollständig angegeben werden, sodass man eine Funktion von den restlichen Parametern erhält).

Dadurch, dass dieses Currying aber nur bei spezieller Deklaration funktioniert, ist es nicht so praktisch wie z.B. in Haskell - aber natürlich ist Swift auch keine funktionale Programmiersprache. Trotzdem ist es ein interessanter Ansatz so ein Konzept in die objektorientierten Welt zu bringen.

**extension Keyword** Ähnlich zu *extension methods* in C# kann man in Swift vorhandene Klassen um Methoden erweitern. Doch Swift erlaubt es auch neue berechnete Properties zu deklarieren:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
}
```

Dann würde beispielsweise `var a = 1.km` der Variable `a` einen Wert von 1000 zuweisen.

An dieser Stelle erkennt man ein anderes Feature von Swift: Man kann zur Leserlichkeit Unterstriche in die Schreibweise von Zahlen inkludieren.

Extensions funktionieren für Methoden ähnlich:

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..

```

Dann würde z.B.

```
3.repetitions({
    print("Hello!")
})
```

3 mal "Hello!" ausgeben. An diesem Beispiel erkennt man auch die kurze Schreibweise von Lambdas sowie Funktionen als Parameter in Swift. Auch sieht man, dass es möglich ist, die Zählervariable einer Schleife per Underscore einfach zu ignorieren.

Leider ist es aber nicht möglich eigene nicht-berechnete Properties hinzuzufügen.

**Initialization process** In Swift, wie auch in Objective C, gibt es keinen wirklichen Konstruktor, sondern die init Methode. Dies hat sich vermutlich in Swift aus Kompatibilitätsgründen nicht geändert.

Die Problematik mit dem Initialisierungsprozess entsteht, wenn man diesen in Kombination mit Vererbung verwendet. In Objective C ist es noch so, dass die init Methode self (aka this) zurückgeben muss, d.h. das initialisierte Objekt (oder nil falls es nicht funktioniert hat). In Swift ist das return self nicht mehr nötig.

Trotzdem ist das Objekt im Falle von Vererbung aber erst "richtig" initialisiert, nachdem super.init () aufgerufen wurde. Somit kann man vor dem super.init () Aufruf noch keine Methoden des eigenen Objekts aufrufen, da das Objekt ja noch nicht fertig initialisiert ist. Das ist jedoch in Kombination mit den nullable Typen problematisch. Diese müssen nämlich zum Zeitpunkt des super-Calls schon initialisiert sein, da sie ja per Definition nicht nil sein dürfen.

Dadurch hat man oft folgendes Problem: Eine Variable soll durch eine Methode initialisiert werden (welche man eventuell wo anders im Objekt auch noch benötigt), soll aber gleichzeitig nicht nullable sein. Dann muss man sich entscheiden entweder den Code doppelt zu haben, einmal in der eigentlichen Methode und einmal vor dem super.init () Aufruf. Oder man deklariert das Objekt als nullable, obwohl es sowieso sofort nach dem super.init () Aufruf einen Wert besitzt und man damit immer force-unwrappen muss.

**Reference counting** Swift verwendet zur Garbage Collection Automatic Reference Counting (ARC). D.h. für jedes Objekt wird mitgezählt, wie viele Referenzen noch vorhanden sind - und falls dieser Counter auf 0 fällt, wird das Objekt gelöscht. Gegen das übliche Problem mit ARC, cyclic references, gibt es in Swift zwei Lösungsansätze:

**Weak References:** Mit dem Schlüsselwort *weak* kann eine Variable als weak reference deklariert werden. Diese zählen dann nicht zum ARC dazu. Wenn die referenzierte Variable nil wird, werden automatisch die darauf zeigenden weak references auf nil gesetzt. Somit muss eine weak reference immer einen nullable Type haben. Verwendet werden weak references beispielsweise vom Framework selbst um UI Elemente an den Controller zu binden: Wird die View zerstört, werden die Referenzen im Controller auf nil gesetzt und die Views können deallokiert werden.

**Unowned References:** Unowned references funktionieren sehr ähnlich wie weak references, sind aber nicht nullable. D.h. eine unowned reference muss immer einen Wert haben. Wenn der Garbage Collector das Objekt dahinter deallokiert kann er somit die dazugehörigen unowned references nicht auf nil setzen - versucht man diese trotzdem zu verwenden bekommt man einen Runtime Error.

Unowned references werden mit dem Schlüsselwort *unowned* deklariert.

**Attributes** In Swift werden Annotations als *Attributes* bezeichnet. Problem ist aber, dass man diese nicht selber definieren kann und somit an die Standard Attributes gebunden ist. Dies macht speziell die UI via Reflection (siehe Abschnitt 4.7.3) schwieriger.

#### 4.4.2 Framework [L]

Cocoa baut auf dem Model-View-Controller (MVC) Design Pattern auf. Da dies ähnlich der .NET Variante ist (Abschnitt 4.5.3), hier nur Unterschiede.

**Data Binding** Es gibt die sogenannten Cocoa Bindings, welche es erlauben, Werte aus dem Controller an die View zu binden. Im Hintergrund funktioniert dies durch key-value coding (KVC) und key-value observing (KVO).

KVC erlaubt es auf Properties von Objekten mit einem String als Key zuzugreifen (ähnlich zu Reflection). Dies ist auf alle Subklassen von NSObject möglich die zumindest ein Property haben (Int, String usw. sind NSObjects, aber KVC funktioniert nicht). Dazu besitzen diese Objekte die Methoden `.valueForKey("key")` sowie `.setValue(value, forKey: "key")`.

KVO hingegen ist ein Mechanismus mit dem man sich bei Properties "registrieren" kann und bei deren Änderungen benachrichtigt wird (Observer Pattern). Dies baut auf KVC auf und ist die Grundlage für Cocoa Bindings.

In Swift müssen die entsprechenden Variablen dazu mit dem dynamic Keyword deklariert werden. Falls man das nicht macht erhält man ein One-Way Binding.

Es gibt dazu dann beispielsweise den NSArrayController, der es erlaubt ein Array an Daten an das UI zu binden. Blöderweise funktioniert KVC wie gesagt “nur” auf Properties und somit kann sich z.B. nicht an ein String-Array an das UI binden. Es ist immer eine wirkliche Model-Klasse dahinter notwendig.

Aus meiner Sicht ist es interessant, dass diese Bindings so stark auf Strings aufbauen. Dadurch muss man z.B. im UI Designer bei den entsprechenden Views die Property-Namen im Controller angeben. Wenn sich diese ändern schafft es die IDE allerdings nicht einen darauf hinzuweisen.

Dies ist genau eine dieser Stellen, wo ein Update einmal gut getan hätte. Zur Zeit als das Cocoa Framework neu war, waren diese Bindings sicherlich revolutionär, doch mittlerweile gibt es alternative Ansätze.

**Keine ListView** Anstatt einer ListView bietet Cocoa nur eine TableView (NSTableView). Der Ansatz ist dann einfach nur eine Spalte zu verwenden, Tabellen Header abzuschalten und das Control passend umzustylen. Doch im Hintergrund (einerseits im Framework, aber auch im Controller Code) bleibt trotzdem eine 2D Struktur erhalten. Bei vielen Methoden erhält man so natürlich nicht nur die Row, sondern auch die Column.

**Storyboards** Storyboards sind der Weg mit dem man in Cocoa User Interfaces aufbaut. Sie haben ihre Vorgänger, die NIB Files, ersetzt. In XCode kann man diese Storyboards auf graphischem Weg designen und so den “flow” der Applikation sehen:

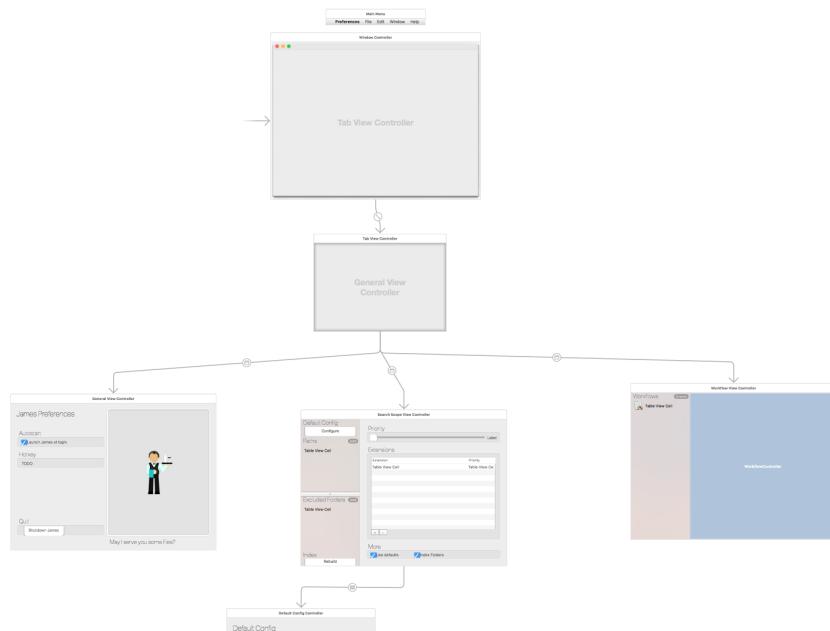


Abb. 25: Storyboards

Dieses Bild zeigt einen Ausschnitt des Storyboards für die James Einstellungen. Jeder View

Controller bekommt ein Fenster - und diese hängen am View Controller des Windows.

**Constraints** Damit die Views sich richtig resizen muss man in Cocoa sogenannte Constraints (Abhängigkeiten) angeben. D.h. eine Reihe linearer Gleichungen welche die Position des Objekts eindeutig bestimmen. Z.B.  $\text{box1.x} = \text{box2.x} - 100$  oder  $\text{box1.height} = 200$ . Man kann diese in XCode (mehr oder weniger) einfach anlegen und bearbeiten. Problem dabei ist nur, dass beim Verschieben einer View (im Designer) natürlich viele dieser Constraints wieder "ungültig" werden und man praktisch wieder von vorne beginnen kann. Gleichzeitig kann es oft schwierig zu "debuggen" sein, wieso ein gewisser Constraint falsch ist/ignoriert wird/nicht eindeutig ist und eine View nicht richtig resized. Speziell dann, wenn man schon eine dementsprechende Anzahl anderer Constraints hat.

**Serialisieren** Um in Cocoa Objekte zu serialisieren muss das NSCoder Protokoll (aka Interface) implementiert werden. Dies verlangt die zwei Methoden initWithCoder: sowie encodeWithCoder:. Beide erhalten ein NSCoder Object und sollen ein Objekt aus diesem Coder erstellen bzw. es abspeichern.

Aus Benutzersicht kann man sich den NSCoder als Dictionary vorstellen, welches Keys auf Werte mappt.

Um das Objekt nun zu serialisieren muss man sich in der encodeWithCoder: Methode alle nötigen Informationen in dem Coder Objekt abspeichern und diese dann im Gegenstück, initWithCoder:, wieder zusammenbauen.

Oder kurzgesagt: Eine Serialisierungs-API, die zu wünschen übrig lässt.

Aus anderen Sprachen ist man es gewohnt, dass diese Dinge von selber passieren. Sprich die Properties via Reflection abgespeichert werden.

Um diese Methoden für James nicht selber implementieren zu müssen, haben wir uns an dieser Stelle für eine der zahlreichen Libraries entschieden, die dieses Problem lösen. In unserem Fall EVReflection. Um Objekte mit dieser Library zu serialisieren müssen sie von EVObject erben (was selber subclass von NSObject ist, d.h. KVC funktioniert). Dann ist die Arbeit mit Methoden wie `.saveToTemp("filename")` bereits getan.

Die Library bietet auch Möglichkeiten, Objekte in JSON umzuwandeln und wieder zu decodieren, was sich speziell beim Speichern/Laden der Workflows als hilfreich herausstellte.

#### 4.4.3 Quick Look [L]

Nach einigen Details zur Programmiersprache sowie dem Framework jetzt noch ein Feature, welches uns viel Zeit und Nerven gekostet hat: Quick Look.

Quick Look ist ein Features des Macs, welches es erlaubt einen "schnellen Blick" auf Dateien zu werfen. Mit einem Klick auf Space öffnet sich ein Fenster, welches eine Vorschau der Datei anzeigt, ohne das eigentliche Programm zu öffnen. Das funktioniert für Programme, Ordner

(Ordnergröße, Anzahl an Dateien etc.), Videos, Audio, Bilder, PDFs und andere Dokumente, Source Code, ...

Dadurch, dass das eigentliche Programm dafür nicht gestartet wird, ist Quick Look innerhalb vom Bruchteil einer Sekunde verfügbar und bietet so eine schnelle Möglichkeit sich einen Überblick zu verschaffen.

Daher wollten wir für James das Feature bieten, mit einem Klick auf die Alt Taste, die Quick Look Vorschau des selektierten Suchergebnis zu sehen (welche mit ESC wieder geschlossen werden kann).

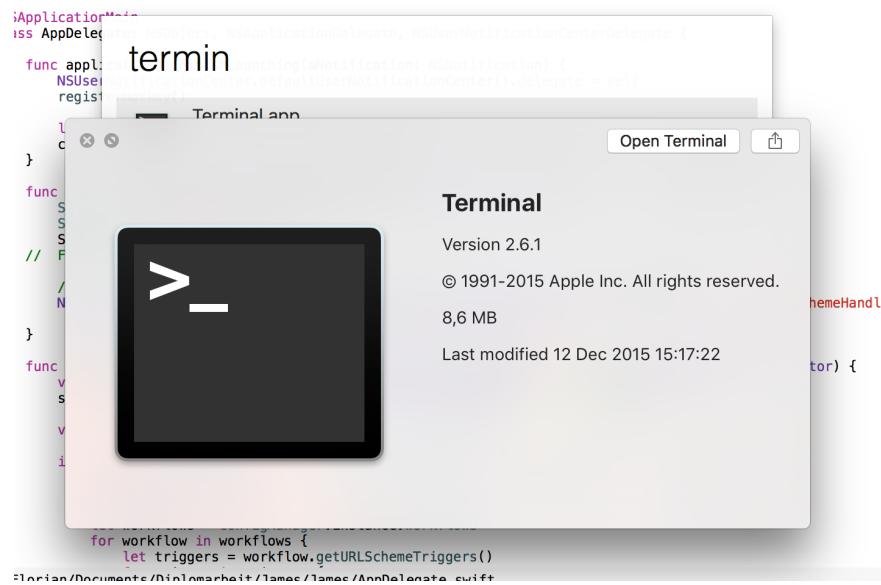


Abb. 26: Quick Look

Hier als Beispiel James mit Quick Look auf die Terminal App.

Die Implementierung dieses Features stellte sich jedoch schwieriger als erwartet dar.

Als ersten Ansatz kommt natürlich die Quick Look Funktion des bekannten oh-my-zsh Plugins “osx” in den Sinn, welche es erlaubt via Command-Line Quick Look auf Dateien anzuwenden. declare -f quick-look liefert:

```
quick-look () {
    (( $# > 0 )) && qlmanage -p $* &> /dev/null &
}
```

Also sollte eigentlich ein Aufruf von qlmanage, was in der dazugehörigen man-Page als *Quick Look Server debug and management tool* beschrieben wird, reichen. Die Beschreibung des -p Parameters: *qlmanage -p displays the Quick Look generated previews for the specified files.*

Bisher hat dieser Quick Look Befehl auch immer gut funktioniert, doch seit dem Update auf OS X 10.10, ist dem nicht mehr so (zumindest auf unserem System). Der Aufruf öffnet nur

noch das qlmanage Programm, zeigt aber keine Quick Look Vorschau mehr an. qlmanage ist also anscheinend wirklich nur das debug tools, als welches es beschrieben wird.

Ein Suche leitet dann auf den sogenannten *Introduction to Quick Look Programming Guide* der Apple Dokumentation. Blöderweise erklärt dieser aber nicht wie man Quick Look “verwenden” kann, sondern wie man Erweiterungen dafür programmieren kann, sodass andere Dateitypen unterstützt werden.

Nach Möglichkeiten die Quick Look Vorschau anzuzeigen sucht man *fast* vergeblich. Viele Seiten schlagen den *qlmanage* Befehl vor, der aber wie beschrieben leider nicht (mehr) funktioniert.

Irgendwann stößt man dann auf Ciarán Walsh’s Blog, <http://ciaranwal.sh/2007/12/07/quick-look-apis>, der (angeblich) eine Lösung beschreibt. Und zwar indem man das Framework unter /System/Library/PrivateFrameworks/QuickLookUI.framework lädt. Vor allem der *Private-Frameworks* Teils des Pfades gab uns zu denken. Aber wie es sich herausstellte, gibt es die QuickLookUI.framework Datei sowieso nicht mehr, somit ist auch dieser Versuch gescheitert.

Eine längerer Suche lässt einen dann die QLPreviewPanel Klasse aus dem Quartz Framework finden, welche diese Funktionalität bieten soll. Beim Versuch diese Klasse in XCode einzutippen stürzt jedoch die IDE ab (genauer: die automatische Textvervollständigung - und als unmittelbare Folge die ganze IDE). Nach dem dritten Versuch kopierten wir den String dann aus dem Editor herein, was zumindest dieses Problem löste.

Wie sich herausstellte muss man dem QLPreviewPanel Objekt eine Data Source angeben, wozu man das QLPreviewPanelDataSource Protokoll implementieren muss. Dieses verlangt zwei Methoden:

```
public func numberOfRowsInSection(panel: QLPreviewPanel!) -> Int
public func previewPanel(panel: QLPreviewPanel!, previewItemAtIndex index: Int) -> QLPreviewItem!
```

Also die Anzahl an Items von denen eine Vorschau angezeigt werden soll, sowie die eigentlichen Items.

Wie sich aber herausstellt, ist QLPreviewItem ein Protokoll welches - wie wir später herausfanden - von NSURL implementiert wird. Vorerst wollen wir aber einfach mal versuchen, was mit einem *return nil* passiert:

Wie erwartet stürzt das Programm in diesem Zustand ab. Aber nicht wie man glaubt wegen des return nils, sondern schon beim Setzen der Datasource, weil unser Objekt laut einer Exception nicht der Controller des QLPreviewPanel’s ist.

Zuversichtlich entdeckt man dann das Field *currentController* der QLPreviewPanel Klasse - das jedoch readonly ist.

Nachforschungen sowie weiteres exzessives Googeln liefern das Ergebnis, dass QLPreviewPanel die responderChain nach oben läuft und jeden in dieser responderChain fragt, ob er der Controller sein möchte. Dazu bieten sich die Methoden:

```
func acceptsPreviewPanelControl(panel: QLPreviewPanel!) -> Bool
```

```
func beginPreviewPanelControl(panel: QLPreviewPanel!)
```

```
func endPreviewPanelControl(panel: QLPreviewPanel!)
```

Laut Dokumentation sind diese im QLPreviewPanelController Protokoll implementiert. Interessanterweise mussten wir jedoch nirgends dieses Protokoll implementieren um diese Methoden überschreiben zu können. Bei uns funktionierte das bei allem, was von NSObject erbt (aus unerklärlichen Gründen).

Es galt jetzt also, sich ein Objekt innerhalb der responderChain dieses Objekts zu suchen, welches sich als Controller registrieren kann. Sich in diese einzutragen probierten wir vergeblich, entdeckten dann aber, dass die AppDelegate Teil dieser responderChain ist.

Also entschlossen wir uns hier, diese 3 Methoden zu überschreiben und die Datasource auf unseren QuickLook-Wrapper zu setzen.

Zum großen Erstaunen “funktionierte” das soweit. D.h. das Programm stürzt wegen des return nils ab - und nicht schon davor.

Ein Blick in den Source Code von QLPreviewItem liefert:

```
extension NSURL : QLPreviewItem {  
}
```

Sprich NSURL implementiert dieses Protokoll und es sollte somit reichen eine NSURL, die zu dieser Datei zeigt, zurückzugeben.

Das funktionierte dann auch - ein Quick Look Fenster mit der entsprechenden Datei erscheint.

Leider lies sich unsere Suchbox nach dem Schließen des Quick Look Fensters nicht mehr fokussieren.

Wie man sich denken kann, stellt sich das Debuggen eines solchen Problems als äußerst schwierig heraus. Einerseits gibt es zu Quick Look selber schon wenig Suchergebnisse - und dementsprechend bei so einem speziellen Fehler noch viel weniger. Andererseits ist der Fehler auch code-technisch schwer zu finden, da ja kein eigener Code für diese Dinge zuständig ist.

Nach stundenlangem Suchen war es uns dann möglich, den Fehler zu entdecken:

Im UI Designer haben wir die Checkbox “Title Bar” deaktiviert, nachdem unser Suchfenster so minimal wie möglich sein soll.

Wie sich aber herausstellt hat ein NSWindow Objekt zwei Properties, canBecomeKeyWindow und canBecomeMainWindow, die für Fenster ohne Title Bar default false zurückgeben. D.h. unser Suchfenster wurde beim Start der Applikation geöffnet und fokussiert, nachdem

man aber das Quick Look Fenster schließt ist dies aber nicht mehr möglich, weil diese zwei Properties false zurückgeben.

Um das Problem ein für alle mal zu lösen, reicht es dann also eine Subclass von NSWindow zu erstellen, welche diese zwei Properties überschreibt und true zurückgibt.

Vermutlich ist Quick Look ein besonders extremes Beispiel, aber es bestätigt unsere generelle Meinung über das Cocoa Framework für OS X.

Es hätte eine einfache Methode gereicht - doch man muss Controller des Objekts werden und eine Datasource erstellen.

Vielleicht ist es nur unserer Unwissenheit geschuldet, aber auf jeden Fall bietet das Cocoa Framework viele solche "Fallen", mit denen man viel Zeit verschwenden kann. Die Programmiersprache Swift kann noch so modern sein, das Framework dahinter stört einfach.

## 4.5 WPF und .NET Framework [M]

Da wir uns auf der Windows Plattform für WPF entschieden haben, werden wir dieses Framework nun etwas näher betrachten, sowie etwaige Implementierungsdetails.

### 4.5.1 Neue Technologien [M]

**XAML (Extensible Application Markup Language)** basiert auf XML und beschreibt den Aufbau von Anwendungsoberflächen deklarativ. Dadurch erreicht man eine klare Trennung von der Anwendungsoberfläche und dem Code.

**DataBinding** Ein mächtige Technologie in WPF ist das DataBinding. Damit können Daten einfach vom Code an das UI gebunden werden. Zusätzlich bietet es die Möglichkeiten im Code veränderte Daten automatisch in dem UI aktualisieren.

**Styles** Bietet die Möglichkeiten eigene Stile einmal zu deklarieren und diese auf mehrere Elemente anzuwenden.

**Routed Events** Je nach Implementierung des Events können diese auch "tunneln" (Event wird an den Parent weitergeleitet) und "bubbeln" (Event wird an die Childs weitergeleitet). Dies geschieht so lange bis ein Element das Property Handled auf true setzt.

**"lookless" Controls** In WPF wird der Style eines Controls von der Funktion getrennt. Dadurch ist es möglich den Style zu überschreiben, was eine hohe Flexibilität mit sich bringt.

#### 4.5.2 Trennung vom Code und Layout [M]

Einer der größten Vorteile von WPF ist die deutliche Trennung vom Layout und dem Verhalten. Dies wird mit den sogenannten XAML Files erreicht. Das Layout wird nicht in Programmiersyntax sondern durch einen XML ähnlichen Ansatz definiert. Dadurch, dass in WPF ein Element eine unendliche Anzahl an anderen Elementen beinhaltet kann, bewahrt man mit diesem Ansatz einen besseren Überblick, da die XAML Code ja in einer Baumstruktur ähnlich HTML definiert wird.

Weiters spielen sogenannte Type-Converter eine wichtige Rolle:

**Type-Converter** Type-Converter ist einer von vielen Gründen, wieso XAML deutlich übersichtlicher und klarer strukturiert ist. Sehen wir und dazu folgendes Beispiel an:

```
<Button Margin="10" Content="OK" />
```

In dieser einen Zeile wird ein *Button* Objekt mit einem *Margin* von einer *Thickness* 10, sowie dem Inhalt "OK" angelegt.

```
Button b = new Button();
b.Margin = new Thickness(10);
b.Content = "OK";
```

Wie man sehen kann, ist der Code für das Anlegen eines solchen Buttons in C# mit einem deutlich höheren Aufwand als in XAML verbunden. Doch wie funktioniert das? Dies geschieht durch eine Vielzahl von bereits in WPF implementierten Type-Converter. Diese haben die Aufgabe wie einen String, wie z.B.: "10" bei Margin="10" in ein brauchbares Objekt zu konvertieren. Dafür existiert in diesem Beispiel ein sogenannter ThicknessConverter, welches den String "10" parset und ein *Thickness* Objekt mit dem Wert 10 anlegt. Natürlich besteht in WPF auch die Möglichkeit eigene Type-Converter zu implementieren, wovon wir auch bei unserem Projekt Gebrauch machten.

**GridHelper** Da man in WPF häufig mit Grids arbeitet und die Definition der Zeilen und Spalten sich als sehr mühsam herausstellte, implementierten wir unseren eigenen sogenannten "GridHelper". Der Vorteil dieses GridHelpers wird an folgendem Beispiel dargestellt:

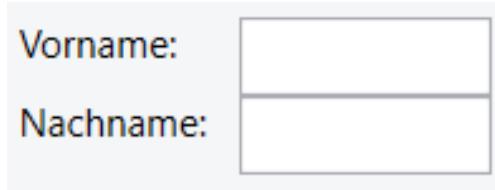


Abb. 27: Beispiel für unseren GridHelper

Dieses Beispiel besteht aus einem Grid mit 2 Spalten sowie Zeilen. Folgende Eigenschaften sollen gesetzt werden:

- Beide Reihen sind gleich hoch.
- Die 1. Spalte ist 70 breit.
- Die 2. Spalte soll den restlichen Platz füllen.

Ohne unseren GridHelper schaut die XAML Definition folgenderweise aus:

```
<Grid Width="150" Height="50">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="70"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="0">Vorname:</TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="0">Nachname:</TextBlock>
    <TextBox Grid.Row="0" Grid.Column="1"/>
    <TextBox Grid.Row="1" Grid.Column="1"/>
</Grid>
```

Mithilfe unseres GridHelper verkürzt sich die Anzahl an notwendigen Zeilen um rund die Hälfte und wir erreichen eine deutlich übersichtlichere XAML-Definition:

```
<Grid Width="150" Height="50" helperClasses:GridHelper.ColumnDefinition="70|*"
      helperClasses:GridHelper.RowDefinition="*|*">
    <TextBlock Grid.Row="0" Grid.Column="0">Vorname:</TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="0">Nachname:</TextBlock>
    <TextBox Grid.Row="0" Grid.Column="1"/>
    <TextBox Grid.Row="1" Grid.Column="1"/>
</Grid>
```

#### 4.5.3 MVC [M]

Zur Trennung des Projektes kam bei uns das bewährte MVC (Model - View - Controller) Muster zum Einsatz. Dies ermöglicht die Teile der Applikation in 3 unabhängige Teilen:

**Model** Das Model beinhaltet die darzustellenden Daten sowie die Routinen für die Programmlogik.

**View** Zeigt die Daten eines Models an und leitet gegebenenfalls Benutzeraktionen an den Controller weiter.

**Controller** Der Controller verwaltet ein oder mehrere Views. Er behandelt die vom View erhaltenen Ereignisse und führt gegebenenfalls die Benutzeraktion aus.

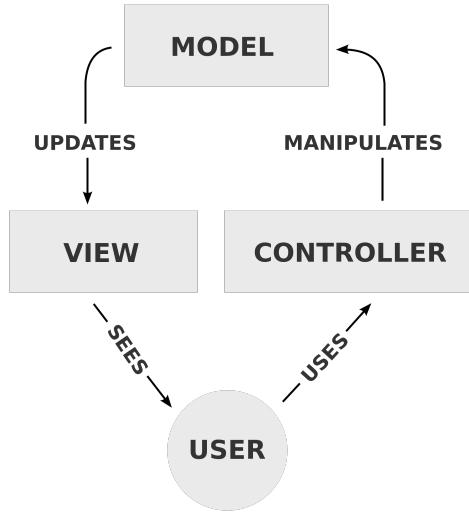


Abb. 28: Das Model-View-Controller Pattern

#### 4.5.4 MahApps [M]

Zur grafischen Verschönerung nutzten wir Mahapps. Dies ist ein quellcodeoffenes Toolkit für WPF. Es ermöglicht ohne großen Mehraufwand WPF Anwendungen ansprechend zu gestalten, da es die Standard-Styles von WPF überschreibt. Weiters erweitert Mahapps einige praktische Controls, auf welche wir auch zurückgriffen:

- HotKeyBox
- ToggleButton bzw. DataGridToggleButton
- Flyouts
- NumericUpDown

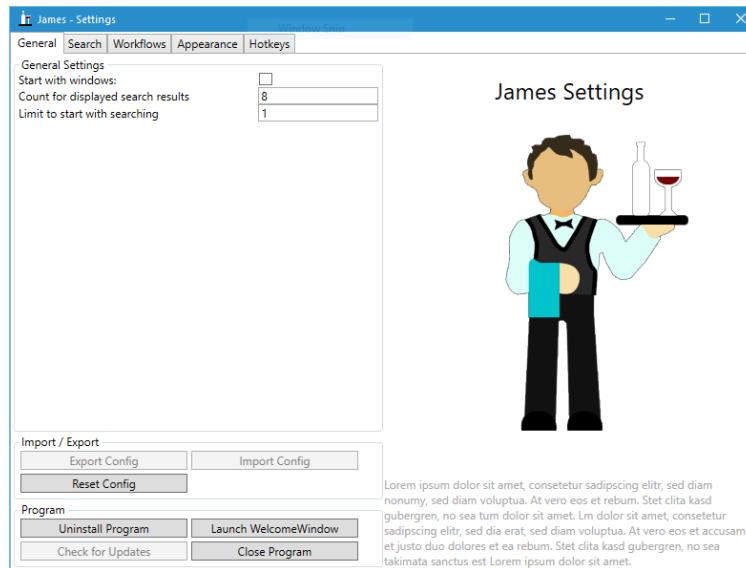


Abb. 29: Ohne Mahapps

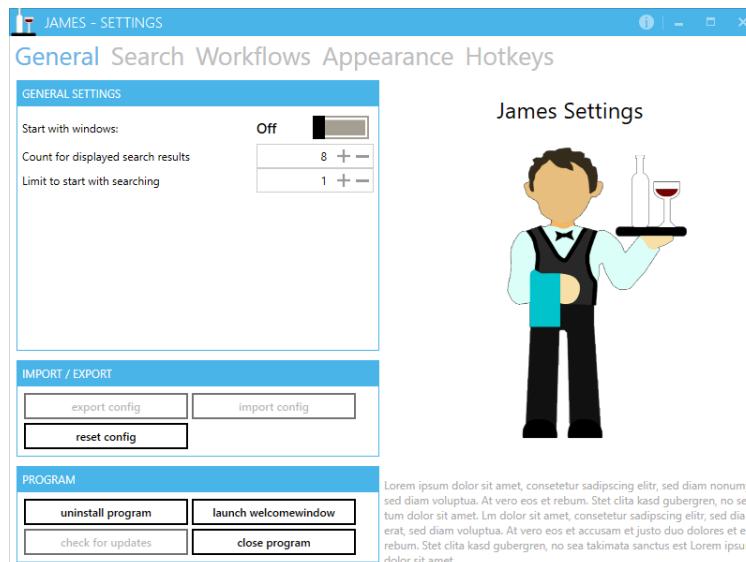


Abb. 30: Mit Mahapps

#### 4.5.5 Aufgetauchte Problematiken [M]

**ListBox Performance Problem** In den ersten Anfangsstunden mussten wir leider ein grobes Performance Problem unter WPF feststellen. Beim Eingeben in das SearchBox-Fenster verzögerte das gesamte UI, da automatisch bei jeder Änderung neu gesucht wird und diese Ergebnisse daraufhin angezeigt werden. Das Problem war nicht, dass wir über den UI Thread suchten, sondern lediglich die im Hintergrundthread erzeugten Ergebnisse in das User Interface zum aktualisieren. Dies ist für uns natürlich eigentlich unausweichlich.

Anfangs verwendeten wir zur Anzeige der Ergebnisse das *ListView* Controll. Nach kurzer Recherche wechselten wir auf *ListBox*, da *ListView* von *ListBox* erbt und nur zusätzliche Funktionen hinzufügt, welche wir auch nicht benötigten. Dies brachte jedoch nicht den gewünschten Effekt. Schnelle Änderungen durch die von der Community empfohlenen Möglichkeiten wie deaktivieren des DataBindings und Aktivierung der Virtualisierung brachten leider nur kleine Verbesserungen und führten nicht zur vollständigen Zufriedenheit.

Ein kurzes Prototyping mittels dem *DataTree* Control brachte auch nicht den gewünschten Erfolg. Zwar erreicht dieses Control schon fast unsere gewünschte Geschwindigkeit, jedoch sind die Gestaltungsmöglichkeiten stark begrenzt.

Meistens versucht man bei Performance Probleme in eine tiefere Schicht zu gehen und sich diese Funktionalität selbst zu schreiben, was auch bei uns zu dem gewünschten Ergebnis führte. Wir nahmen das *FrameworkElement* als Basis. Dieses Objekt ist gerade gut genug, DrawingVisuals anzuzeigen. DrawingVisuals ist ein Container für extrem leichtgewichtige Elemente, welche sich nicht selbst zeichnen können u.a. Texte, Bilder und Rechtecke. Wir erstellten daraufhin eine eigene Klasse *SearchResultElement*, welche die *SearchResults* darstellt. Dabei werden in das DrawingVisual an von uns definierten X, Y Positionen Text, Bilder und Rechtecke eingefügt. Zum Schluss wird dieses Visual in den VirtualTree unter dem *FrameworkElement* angehängt.

**Global Hotkeys** Da unser Programm so gut wie möglich ohne Maus auskommen soll, sind Tastenkombinationen natürlich unausweichlich. James hat mehrere Standard Kombinationen:

- **Alt + Space** bringt das SearchBox-Fenster von James in den Vordergrund und ist dadurch sofort einsatzbereit, um nach Dateien zu suchen, oder Workflows zu starten. Diese Tastenkombination ist die einzige Standardtastenkombination, welche global (systemweit) funktionieren muss, sprich auch während ein anderes Fenster im Fokus steht, soll es möglich sein James mittels dieser Kombination in den Vordergrund zu versetzen.
- **Alt + S** Liegt das SearchBox-Fenster gerade im Fokus können mittels dieser Tastenkombination die Einstellungen geöffnet werden. Alternativ ist das auch über einen Eintrag im Kontextmenü möglich (Rechtsklick aufs Fenster).
- **Alt + L** Diese Tastenkombination öffnet unseren LargeType. Ein Fenster, wo Text über den ganzen Bildschirm, weit leserlich angezeigt werden kann. Standardmäßig wird hierbei die aktuelle Eingabe in die Suchbox angezeigt. Ist jedoch gerade ein MagicOutput fokussiert, wird stattdessen der Titel dieses Outputs angezeigt.
- **Shift + Enter** Öffnet die aktuell selektierte Datei oder den Ordner im Windows Explorer und fokussiert diesen.
- **Strg + Shift + Enter** Wie unter Windows üblich, wird oft mittels Drücken von Strg + Shift und einem Doppelklick auf ein Programm, dieses als Administrator gestartet. Diese Funktionalität wollten wir natürlich nicht missen und implementierten diese.

Natürlich sind die beiden letzteren genannten Tastenkombinationen auch mittels einem Mausklick statt dem Drücken der Entertaste möglich.

Für eine bessere Benutzerfreundlichkeit entschieden wir uns, diese Tastenkombinationen vom User selbst einstellbar zu machen. Hierbei kam uns während der Arbeiten das Release des neuen Control von Mahapps *HotKeyBox* sehr gelegen. Mit dessen ist es möglich, dem User einfach durch das Fokussieren dieses Controls und das Drücken der Tastenkombination die Tastenkombination neu zu setzen. Je nach Bedarf können über Konfigurationsmöglichkeiten des Controls Modifier-Keys vorausgesetzt werden.

Für die Funktion einer globalen Tastenkombination entschieden wir uns für das NuGet-Package *GlobalHotKey*, welches eine quellcodeoffenen, frei verfügbaren Code auf Github darstellt.

Da wir nun sowieso diese Funktionalität implementiert haben, und unser verwendetes NuGet-Package mehrere globale Tastenkombinationen umgehen kann, entschieden wir uns dazu, den Benutzer die Möglichkeit weitere globale Tastenkombinationen für den eigenen Gebrauch zu erstellen. Dafür erweiterten wir unser Einstellungsfenster mit einem weiteren Reiter:

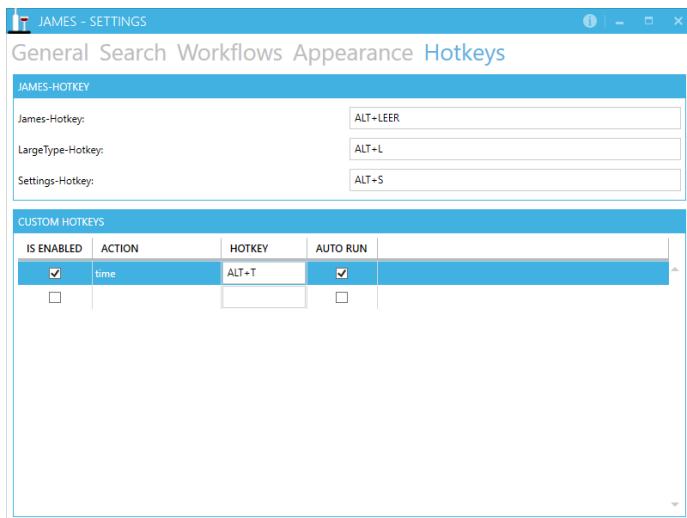


Abb. 31: Hotkey-Reiter im Einstellungsfenster

zusätzliches Drücken der Entertaste mehr nötig und dieser Workflow wird automatisch gestartet, falls ein Keyword-Trigger für die Eingabe existiert. Weiters kann auch eine Datei bzw. ein Verzeichnis automatisch geöffnet werden, wenn die entsprechende Datei oder der entsprechende Ordner existiert.

**Mutex** Da es für uns keinen Sinn macht mehrere Instanzen von James am System laufen zu lassen, suchten wir nach einer Möglichkeit, welche immer nur eine Instanz laufen lässt.

Nach einer kurzen Recherche stießen wir auf Mutex. Mutex ist ein Kernel Objekt von Windows und muss vom Usercode zuerst über einen Handle angefragt werden. Für das .NET Framework gibt es schon eine fertige Implementierung, dadurch ist der Aufwand überschaubar.

Oben können einige Standard Tastenkombination benutzerspezifisch angepasst werden. Darunter folgt eine List mit der vom Benutzer zusätzlich erstellte globale Tastenkombinationen. Der Boolean “IsEnabled” kümmert sich, wie der Name bereits vermuten lässt, ob diese Kombination aktuell aktiv ist. Der Actionstring ist jener Text, welcher beim Auslösen dieser Kombination in die SearchBox einge tragen wird und der “AutoRun” Boolean bietet die Möglichkeit die Eingabe (Actionstring) automatisch zu starten. Dadurch ist kein

Unser Programm versucht beim Starten mittels des *Mutex Object* in C# einen Mutex mit dem Namen “*James*” anzufordern. Ist dies erfolgreich, können wir uns sicher sein, dass dies die erste Instanz von James ist. Andererseits läuft bereits eine andere Instanz auf unserem System. Ist dies der Fall, prüfen wir vorm Beenden noch, ob die beim Starten der Instanz gültige Parameter für die Api mitgegeben wurde und senden der laufenden James-Instanz noch schnell dieses Argument.

**Fenster vom Windows Task Switcher verbergen** Da es während der Entwicklung zu unschönen Effekten gekommen ist, wodurch das James Programm in der *Windows Task Switcher* aufgetaucht ist, suchten wir eine Lösung für dieses Problem:

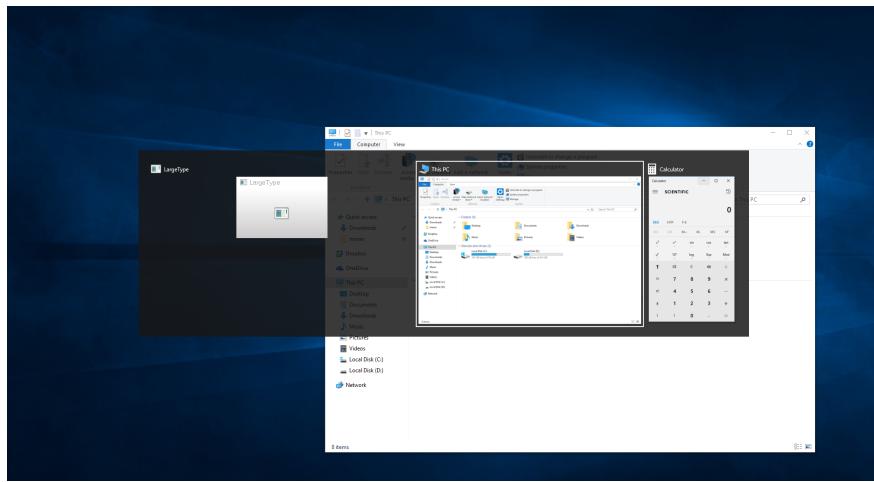


Abb. 32: Unschöne Darstellung von James in der *Windows Task Switcher*

Durch eine kurze Recherche fanden wir eine Lösung, welche jedoch nicht innerhalb des .NET Frameworks möglich war. Durch die *P/Invoke* Technik mussten wir Funktionalitäten aus der *user32.dll* importieren. Diese kümmert sich darum, das Fenster auf den Status eines *Toolwindow* zu setzen, ohne jedoch das Verhalten des Fensters auf dieses zu setzen und löste dadurch dieses Problem.

## 4.6 ASP.NET Core 1.0 [M]

Da wir uns, wie aus den Lösungsansätzen bereits ersichtlich, für ASP.NET Core 1.0 entschieden haben, stellen wir nun das Framework näher vor.

Wie bereits Abschnitt 3.5.3 kurz erwähnt, überzeugt es durch neue interessante Ansätze, auf die wir jetzt jeweils weiter eingehen:

## Open Source [M]

Dies ist ein nicht zu unterschätzender Punkt. Zwar wird das Framework aktiv von Microsoft weiterentwickelt, jedoch kann sich daran jeder willige Entwickler während der Entwicklung einschalten und auch Wünsche sowie Kritik äußern. Das auf Github gehostet Projekt ist unter folgendem URL erreichbar: <https://github.com/aspnet>. Weiters drängt eine Open Source Entwicklung auf höhere Codequalität, weil von mehreren Personen reviewt wird, was zu einer schnelleren Auffindung und Behebung von Fehlern führt.

## Plattformübergreifend [M]

Das Framework basiert auf dem ebenfalls Open Source Projekt *.NET Core*.

.NET Core is a modular version of the .NET Framework designed to be portable across platforms for maximum code reuse and code sharing.

(Quelle: [https://msdn.microsoft.com/de-de/library/dn878908\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/dn878908(v=vs.110).aspx), 31.März 2016)

Folgende Graphik soll den Systemaufbau weiter verdeutlichen:

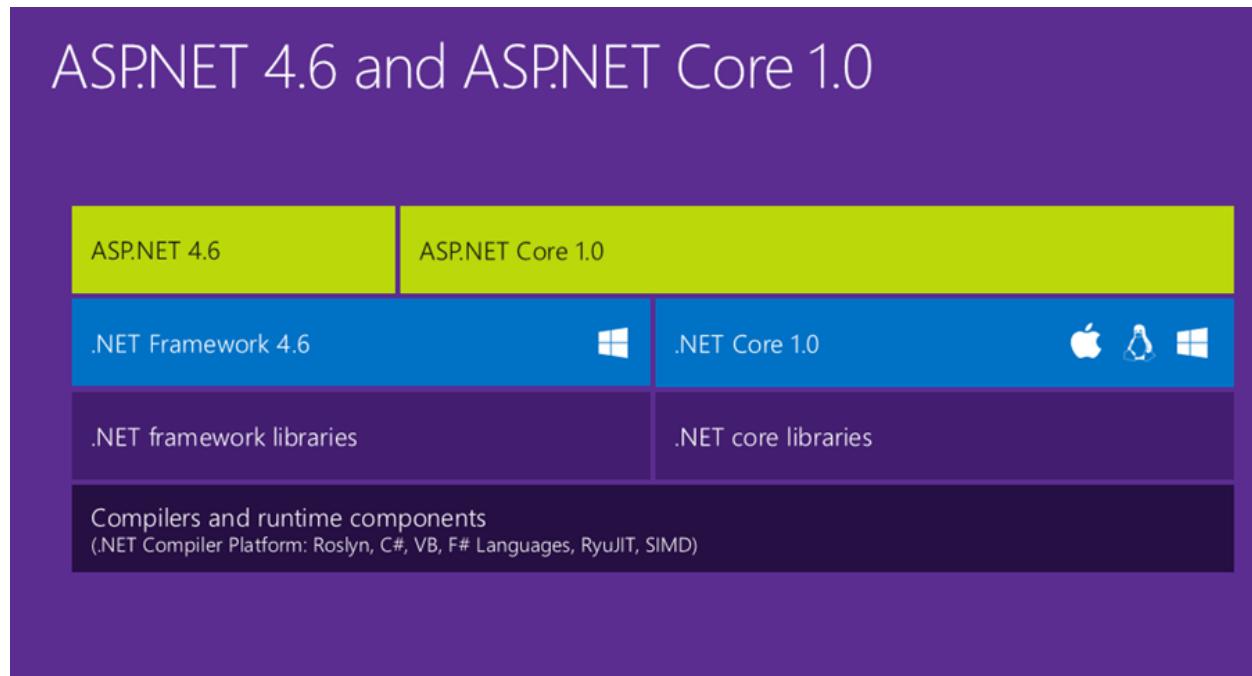


Abb. 33: ASP.NET Core 1.0 im Vergleich zu ASP.NET 4.6

Dadurch ist es möglich ASP.NET Core, wie bereits auf der Graphik ersichtlich, mittels .NET Framework 4.6 (Windows) oder .NET Core (Windows, OS X und Linux) zu starten. Zu diesem Anlass hat Microsoft auch die IDE Visual Studio Core entwickelt, welches viele Funktionen von Visual Studio auf die jeweils anderen Plattformen bringt. Man will dadurch eine größere Verbreitung erreichen.

## State of the Art [M]

Das neue ASP.NET Core weist viele Ähnlichkeiten zu Node.js auf. Beide sind quellcodeoffen sowie modular. Alle Module werden, wie auch unter Node.JS, mit einem Package Manager verwaltet (npm für Node, Nuget-Paket für ASP.NET Core). Die *project.json* bildet den Grundaufbau einer ASP.NET Core Applikation und definiert alle Dependencies sowie allgemeine Konfigurationen. Weiters wird bereits von Anfang an bei der Entwicklung auf die gängigsten clientseitigen Frameworks (u.a. AngularJS, KnockoutJS, Bootstrap) geachtet und es besteht die Option zur Integration von Bower. Das Erstellen einer ASP.NET Core Applikationen geschieht mittels Grunt. Wie man bereits aus dieser kurzen Aufzählung sehen kann, setzt Microsoft bei Core auf viele populäre sowie quellcodeoffenen Komponenten.

## Modularisiert [M]

Mit der Modularisierung hat Microsoft unserer Meinung nach den richtigen Schritt gewählt.

Das ASP.NET 4.6 Framework ist ein Teil des gesamten .Net Frameworks und stark mit diesem verflochten. Alleine die DDLs belaufen sich auf 14.2 MB. Dadurch besteht keine Möglichkeiten, sich nur einen Teil dieser Auswahl zu nehmen, den man auch wirklich benötigt, sondern muss immer den kompletten System.Web.\* Namespace importieren. Diese Lösung hat zwei große Nachteile:

1. Weiterentwicklungen sowie Fehlerbehebungen können nur immer gebündelt mit dem gesamten .NET Frameworks erfolgen.
2. Durch das Laden von nicht verwendeten Teile des Frameworks leidet auch die Performance.

Dadurch verfolgt das neues Core Framework eine andere Lösung. Es teilt den gesamten System.Web.\* Namespace in viele kleine Module auf und führt für jedes Modul eine eigene Versionierung ein. In der *project.json* werden diese nun definiert und mittels Versionsnummer wird bestimmt, welche Version des einzelnen Moduls geladen werden soll. Dadurch behebt man die Nachteile von ASP.NET 4.6. Man kann nun einzelne Module unabhängig vom gesamten Framework updaten und bietet die Möglichkeit nur die Module zu laden, welche auch wirklich benötigt wird.

## Leichtgewichtiger mit Hilfe der Modularisierung und dadurch schneller [M]

Wie bereits im oberen Abschnitt erwähnt wurde durch die Modularisierung ein deutlicher Performancezuwachs verzeichnet. Damit konnte man wieder moderneren Webframeworks wie etwa Node.JS etwas entgegensetzen.

#### 4.6.1 Namensgebung [M]

Vorherige Namen waren u.a. ASP.NET vNext oder ASP.NET 5. Jedoch wurde das Projekt später auf ASP.NET CORE 1.0 umbenannt, da man damit zeigen wollten, dass es sich um eine unabhängige Entwicklung von .NET Framework handelt und es keine Weiterentwicklung von ASP.NET 4.6 ist.

### 4.7 Workflows [M]

#### 4.7.1 Persistierung [M]

Da wir Workflows auch zwischen den Plattformen teilen wollten, mussten wir uns auf ein gemeinsames Model für die Persistierung einigen. Wir beschlossen einen gemeinsamen Aufbau für unsere *.james* Datei. Diese Datei ist einfach ein gezippter Ordner, mit folgendem Inhalt:

- config.json
- icon.png - optional
- sonstige Dateien z.B.: Skripte, Bilder, ...

Die *config.json* ist der einzige verpflichtende Bestandteil im Aufbau. Diese definiert die grundsätzlichen Eigenschaften eines Workflow. Darin werden Informationen wie Author, Verfügbarkeitsstatus und die Auflistung an Workflowkomponenten definiert.

Eine Workflowkomponente besitzt eine eindeutige Id, eine Angabe des Typen, Links zu Kinder ("connectedTo"), welcher die eindeutige Id der Kinder beinhaltet, sowie X/Y - Koordinaten zur Positionierung.

```
{
  "author": "moser",
  "enabled": true,
  "components": [
    {
      "id": 0,
      "type": "keyword",
      "connectedTo": [
        5
      ],
      "y": 5.0,
      "x": 0.0,
      //component specific attributes:
      "title": "change battery management",
      "subtitle": "bm",
      "keyword": "bm",
      "autorun": true
    }
    //other components...
  ]
}
```

Abb. 34: Grundsätzlicher Aufbau der config.json

#### 4.7.2 Importierung von Workflows [M]

Da unsere Workflows mit *.james* sowieso eine eigene Dateiendung haben, konnten wir zum Importieren diesen Vorteil auch nutzen. Auf der jeweiligen Plattform setzten wir unser Programm als Standardprogramm für alle *.james* Dateien. Dadurch ist es ganz leicht, James-Workflows zu importieren. Dies geschieht einfach mittels Starten der Datei.

**4.7.2.1 Implementierung Windows [M]** Das Eintragen von James als Standardprogramm aller *.james* Dateien geschieht über die Registry. Dort kann das FileIcon eingetragen werden. Ein Pfad zur Executable zum Öffnen ist natürlich verpflichtend. Dazu verwenden wir einfach unsere James.exe. Ähnlich wie beim Custom-Url-Protocol wird mittels NamedPipes der laufenden Instanz der Name des zu importierenden Workflows mitgeteilt. Die laufende Instanz importiert daraufhin den Workflow, falls die Datei das richtige Format aufweist.

**4.7.2.2 Implementierung OS X [L]** Unter OS X funktioniert das Ganze ähnlich, aber es gibt natürlich keine Registry dafür. Stattdessen trägt man im Info.plist File der Applikation die gewünschte Dateiendung ein.

Damit die App dann darauf reagieren kann, sollte im AppDelegate die Methode

```
optional func application(_ sender: NSApplication,
  openFile filename: String) -> Bool
```

überschrieben werden. Diese wird aufgerufen sobald eine entsprechende Datei geöffnet wird.

James reagiert darauf, indem er das File nach “/Users/{username}/Library/Application Support/James/workflows/” extrahiert und im Programm den Workflow lädt.

#### 4.7.3 UI über Reflection [M]

Da wir für jeden Workflowkomponenten ein eigenes Einstellungsfenster benötigten, suchten wir nach einer guten Möglichkeit, diese generisch zu entwickeln. Durch einen generischen Dialog würden wir natürlich auch den Aufwand zum Erstellen eines neuen Komponenten stark vereinfachen.

Daher entschlossen wir uns eine Lösung mittels Reflection zu entwickeln. Natürlich ist dies vorerst ein bedeutend größerer Aufwand, was sich jedoch bereits im Verlauf der Diplomarbeit gerechnet hat. Einen weiteren Nachteil bringt diese generische Erzeugung des Dialoges mit sich, indem man layouttechnisch nur auf eine begrenzte Anzahl von Controls zurückgreifen kann (String, Number, Boolean) und das Layout nur äußerst schwer für einen individuellen Komponenten anpassbar ist.

Natürlich eignet sich für diese Methode, die im .NET Umfeld üblichen Custom-Annotations. Durch das Fehlen der Möglichkeiten von eigenen Annotations im Cocoa / Swift Umfeld ist dies dennoch über kleinere Umwege möglich um ein ähnliches Ereignis zu erreichen.

**4.7.3.1 Windows [M]** Für die Implementierung unter Windows erstellten wir eine Custom-Annotation *ComponentFieldAttribute*, welches über einen String “*Description*” und einem Boolean Flag “*isFile*” verfügt.

Im nächsten Schritt wird über jedes Property des Komponenten dieses Attribute geschrieben. Am Beispiel des Api-Triggers sieht das Action-Feld wie folgt aus:

```
[ComponentFieldAttribute("The name of the action to listen")]
public string Action {get; set;} = "";
```

Zur Erinnerung, dabei handelt es sich um das Wort, worauf im Custom-Url-Protocol gehorcht wird.

Beim dynamischen Erstellen des Dialoges braucht man nur noch alle Properties, welche auch unsere Annotation besitzt, heraus filtern und nacheinander im Dialog eintragen. Der Typen des Controls kann über Reflection anhand des Properties gefunden werden. Zu den folgenden Typen, werden folgende Controls erzeugt:

- String: Erzeugt ein Textfeld
- Int32: Erzeugt ein NumericUpDown-Controll von Mahapps
- Boolean: Erzeugt einen Mahapps ToggleSwitchButton

Ein fertig generierter Dialog des Api-Triggers sieht wie folgt aus:

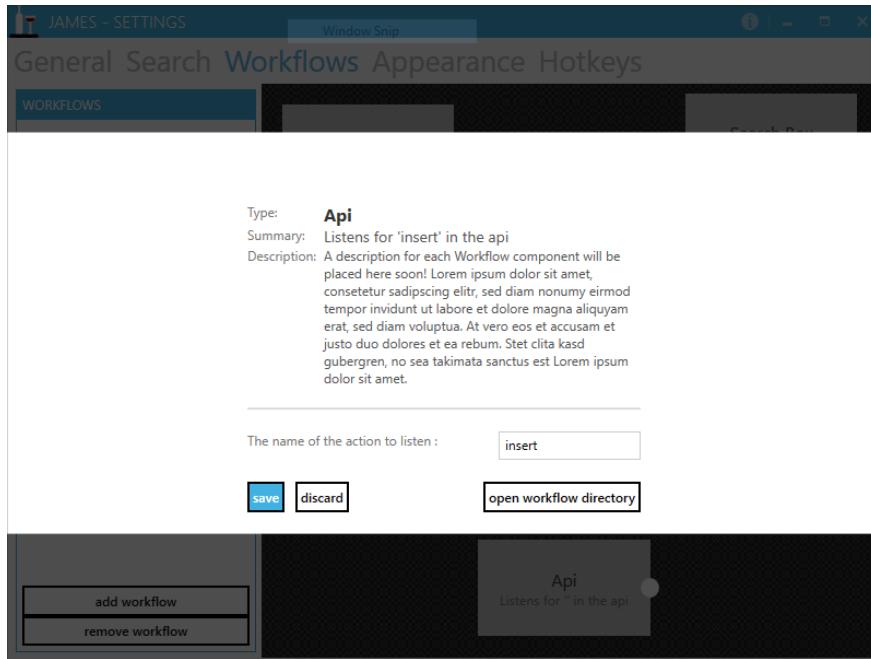


Abb. 35: Windows: Generierter Dialog des Api-Triggers

**4.7.3.2 OS X [L]** Wie bereits erwähnt, fehlt unter Swift aktuell noch die Möglichkeit Custom Annotations zu definieren. Zusätzlich gibt es keine passenden vorgefertigten Annotations für diesen Zweck.

Doch durch KVC ist es trotzdem möglich “reflection ähnlich” mit den Fields zu arbeiten. Es fehlt einfach die Annotation um zu definieren, welche Fields (nicht) angezeigt werden sollen.

Deshalb haben wir uns für den “Umweg” entschieden, jedem Component ein Array aus String Paaren zu geben, welche Name des Felds, sowie dessen Beschreibung definieren.

Den Dialog zu generieren, funktioniert dann analog zur Windows Variante, es wird halt einfach über ein (String, String) Array iteriert, anstatt diese Fields durch die Annotation zu finden.

Script-Actions werden im Vergleich zu Windows aber anders dargestellt. Für diese wird ein “Edit” Button generiert. Dieser öffnet dann den Default-Editor, indem man sein Script programmieren kann:

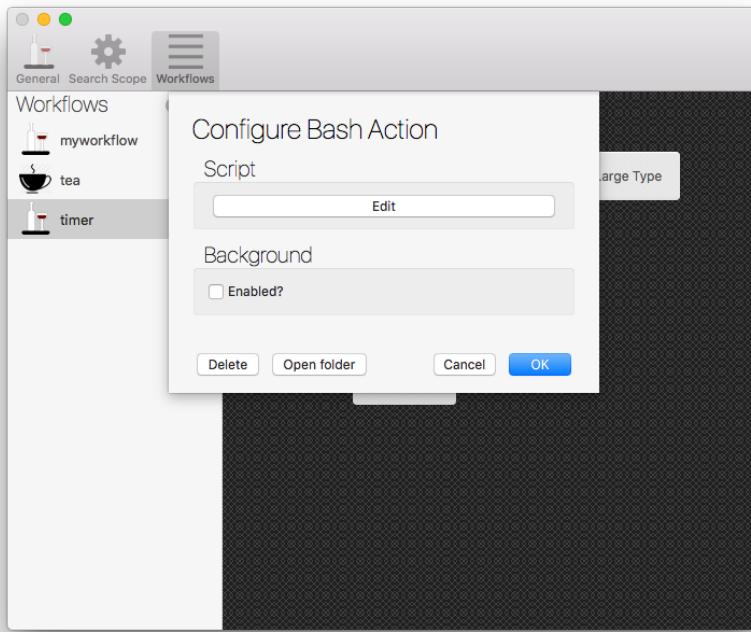


Abb. 36: OS X: Generierter Dialog der Bash Action

#### 4.7.4 Custom-Url-Protocol-Hanlder bzw. Url Scheme [M]

Wir haben uns dazu entschieden, für James eine Api zu entwickeln. Dadurch soll eine Möglichkeit angeboten werden, von anderen Programmen James Workflows mittels dem Api-Trigger zu starten. Da der Aufruf dieser Api leicht zu verwenden sein soll, haben wir uns für ein eigenes Custom-Url-Protocol (Windows) / Url Scheme (MacOS X) entschieden. Ein solches Protokol ist vom Format *james:[Api-Trigger-Keyword]//[optionale Parameter]*. Dadurch ist es zum Beispiel möglich mittels

```
<a href="james:insert / test">'Test' in Suchbox einfuegen</a>
```

und einem passenden Workflow, welcher aus einem Api-Trigger sowie SearchBox-Output besteht, James in den Vordergrund zu bringen und den Text “test” in die Suchbox anzuseigen.

**4.7.4.1 Windows - Custom-Url-Protocol-Handler [M]** Unter Windows wird ein eigenes Custom-Url-Protocol durch mehrere Registry-Einträge erzeugt. Dieses muss unter anderem auch einen Pfad zur .exe enthalten, welcher gestartet werden soll. Wir entschieden uns einfach unsere *James.exe* zu nehmen, da wir durch die Verwendung von einem Mutex leicht unterscheiden können, ob es ein Api-Aufruf oder einen Start darstellt. Läuft bereits eine Instanz von James kann dieser Mutex nicht angefordert werden. Darauf prüfen wir die Parameter vom Starten, der Form der Api entsprechen und falls dies zutrifft wollen wir dies

der aktuell laufenden Instanz mitteilen. Zur Interprozesskommunikation entschieden wir uns für die Verwendung von *NamedPipes*. Natürlich muss hierfür die laufende Instanz auf diesem Pipe im Hintergrund immer horchen.

Genauere Details zum Custom-Url-Protocol können unter [https://msdn.microsoft.com/en-us/library/aa767914\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa767914(v=vs.85).aspx) abgerufen werden.

#### 4.7.4.2 OS X - Url Scheme [L]

Auch hier ist die Lösung unter OS X im Vergleich zur Registry das Info.plist File des Programs:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLName</key>
        <string>Open james:// URLs</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>james</string>
        </array>
    </dict>
</array>
```

Für dieses URL Schema muss man sich dann beim NSAppleEventManager für ein Callback registrieren. Dies passiert also schon innerhalb der Applikation und macht keine Interprozesskommunikation notwendig.

Falls man dieses Callback nicht wieder entfernt, wird das Programm gestartet, falls so ein Event auftritt.

#### 4.7.5 Format String [L]

Eine weitere Konfigurationsmöglichkeit von James Workflows sind die Format Strings.

Viele Komponenten verlangen einen String zur Konfiguration. Der MagicOutput will beispielsweise jeweils für den Titel und den Subtitel einen String.

Doch dieser sollte oft von den übergebenen Parametern abhängig sein. Es kommen z.B. beim Supplierplan Workflow Daten von einer ScriptAction zurück, welche dann in einem MagicOutput angezeigt werden sollen.

Um dies zu realisieren, bauten wir die Möglichkeit ein, Textfelder nicht rein auf Text zu beschränken, sondern auch Placeholder zu erlauben. Man kann beispielsweise als Titel “hello {0}” angeben. “{0}” wird dann durch den ersten Parameter ersetzt. Ähnlich natürlich mit “{1}” für den zweiten Parameter usw. Nicht vorhandene Parameter werden dabei durch einen Leerstring ersetzt.

Zusätzlich gibt “{#}” die Anzahl an Parametern an und “{...}” ist die Konkatenation aller Parameter mit einem Space als Trennzeichen.

Aus technischer Hinsicht handelt es sich hierbei um ein einfaches “replace”.

#### 4.7.6 Komponenten im Hintergrund ausführen [M]

Im Normalfall wird ein Workflow gestartet und eine Aktion ausgeführt. Benötigt dieser Workflow jedoch längere Zeit, stellt sich die Frage, ab wann dieser beendet werden soll. Bei Workflows mit Internetzugriff ist dies oft der Fall: Z.B der Supplierplan Workflow der HTL Leonding (siehe Abbildung 2). Hier stellt sich die Frage, ob dieser durch das Schließen des SearchBox-Fensters, oder Neueingabe in das Suchfeld unterbrochen werden soll oder nicht. Diese Entscheidung wollten wir den ErstellerInnen überlassen.

Um dies möglichst einfach zu gestalten, gingen wir wie folgt vor: Nachdem eigentlich nur Actions davon betroffen sind, da Inputs und Output kaum Zeit benötigen, beschränkten wir uns auf diese. Wir erstellten für jede Action ein zusätzliches Feld “*background*”, welches bestimmt, ob der Workflow beim Schließen von James oder bei einer Neueingabe gestoppt werden soll oder nicht. Ist Background auf True gesetzt, wird der Workflow nicht abgebrochen und kann zu einem späteren Zeitpunkt eine Output-Komponente starten.

## 5 Evaluation des Projektverlaufs

### 5.1 Meilensteine [M]

Zu Beginn der Diplomarbeit definierten wir folgende Meilensteine:

01.10.2015	Suchalgorithmus abgeschlossen
01.11.2015	Dateisuche funktioniert
01.12.2015	Suche personalisierbar / Einstellungen fertig
01.01.2016	Workflows können erstellt und verwendet werden
01.02.2016	Workflow-Austausch über den Webstore möglich

Rückblickend können wir sagen, dass alle Meilensteine fristgerecht abgeschlossen wurden. Natürlich kommt es, wie in der Softwareentwicklung üblich, zu unvorhergesehenen Fehler, welche erst später gefunden und dann dementsprechend gefixt werden. Aber bis auf solche kleineren Bugs, war es uns immer möglich den Zeitplan einzuhalten.

### 5.2 Gelerntes [L]

Während des Projektverlaufs haben wir eine Menge gelernt. Und das nicht nur im technischen Bereich, sondern auch in Dingen wie Planung, Team Arbeit, etc.

So konnten wir mit der Entwicklung des Suchalgorithmus unser Wissen im Datenstruktur-Bereich verbessern. Gleichzeitig gab uns dieses Projekt auch die Möglichkeit zu lernen, wie man C++ in Kombination mit Swift bzw. C# verwenden kann.

Weiteres vertiefte sich unser Können im Bereich der Mac Entwicklung, aber natürlich auch im .NET Umfeld. James ermöglichte es uns, mit neuen Technologien, wie z.B. ASP.NET Core, umgehen zu lernen - aber auch das Beste aus alten (bzw. veralteten?) wie dem Cocoa Framework zu holen.

Andererseits wurde uns durch diese Diplomarbeit auch klar, dass Zusammenarbeiten in der Theorie einfacher ist als in der Praxis. Z.b. ist oft für einen eine Idee klar, benötigt aber mehr Details sodass diese auch der Partner versteht. Speziell in der Entwicklung der Workflows war diese Zusammenarbeit sehr wichtig, da es anderenfalls nicht möglich ist, kompatibel zwischen den beiden Plattformen zu bleiben.

Wir lernten auch besser mit Versionsverwaltung, wie in unserem Fall Git, zu arbeiten, um immer auf einem gemeinsamen Stand zu sein.

### 5.3 Was würden wir anders machen? [L]

Auch wenn wir beide sehr zufrieden mit dem Verlauf der Diplomarbeit sind, gibt es natürlich immer Potential nach oben:

Wir haben beispielsweise viel Zeit damit verbracht, Alternativen für eine eigene Implementierung des Suchalgorithmus zu finden. Rückblickend wären wir natürlich schneller gewesen, hätten wir diesen Schritt übersprungen.

Trotzdem mach es Sinn sich nach fertigen Lösungen umzusehen, man möchte ja grundsätzlich nicht das “Rad neu erfinden”.

Bei den Workflows hätten wir auch Zeit einsparen können, indem wir uns besser abgesprochen hätten. So haben wir Anfangs beide in einem eigenständigen Format entwickelt und mussten im Nachhinein viele Details ändern, sodass die beiden Versionen kompatibel sind.

## 6 Quellen-, Literatur- und Abbildungsverzeichnis

### 6.1 Quellen- und Literaturverzeichnis

- [1] <http://www.hanselman.com/blog/ASPNET5IsDeadIntroducingASPNETCore10AndNETCore10.aspx>
- [2] <https://de.wikipedia.org/wiki/PHP>
- [3] <https://de.wikipedia.org/wiki/ASP.NET>
- [4] [https://msdn.microsoft.com/de-de/library/dn878908\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/dn878908(v=vs.110).aspx)
- [5] [https://de.wikipedia.org/wiki/Windows\\_Forms](https://de.wikipedia.org/wiki/Windows_Forms)
- [6] [https://en.wikipedia.org/wiki/Graphics\\_Device\\_Interface](https://en.wikipedia.org/wiki/Graphics_Device_Interface)
- [7] [https://en.wikipedia.org/wiki/Windows\\_Presentation\\_Foundation](https://en.wikipedia.org/wiki/Windows_Presentation_Foundation)
- [8] Buch Windows Presentation Foundation 4.5  
2013 Thomas Claudius Huber; ISBN: 978-3-8362-1956-3
- [9] <http://getwox.com/>
- [10] <http://stackoverflow.com/questions/334326/what-is-managed-unmanaged-code-in-c>
- [11] <http://e-maxx.ru/algo/>
- [12] <http://kukuruku.co/hub/algorithms/radix-trees>
- [13] [https://de.wikipedia.org/wiki/Apache\\_Lucene](https://de.wikipedia.org/wiki/Apache_Lucene)
- [14] <https://www.alfredapp.com/>
- [15] <https://drthitirat.wordpress.com/2013/06/03/use-c-codes-in-a-c-project-wrapping-native-c-with-a-managed-clr-wrapper/>
- [16] <https://developer.apple.com/library/mac/documentation/>

## 6.2 Abbildungsverzeichnis

1	Alfred Workflows . . . . .	9
2	Supplierplan Workflow . . . . .	9
3	Search Scope Konfiguration OS X . . . . .	17
4	Search Scope Konfiguration Windows . . . . .	18
5	Win: Interaktiver Workfloweditor . . . . .	19
6	OSX: Interactive Workflow Editor . . . . .	19
7	Win: Komponente-Konfigruation . . . . .	20
8	OSX: Komponente-Konfigruation . . . . .	20
9	Beispiel Keyword mit AutoRun . . . . .	21
10	Beispiel Keyword mit AutoRun 2 . . . . .	21
11	Win: LargeType-Output . . . . .	24
12	OSX: LargeType-Output . . . . .	24
13	Win: Notification-Output . . . . .	24
14	OS X: Notification-Output . . . . .	25
15	Win: Magic-Output . . . . .	25
16	OSX: Magic-Output . . . . .	25
17	Trie . . . . .	31
18	Radix Tree . . . . .	32
19	Binärbaum Darstellung . . . . .	32
20	Split . . . . .	34
21	Join . . . . .	35
22	Verwendung des _WIN32 Macros . . . . .	40
23	Filenamen am Mac . . . . .	40
24	Wrapper zur Überbrückung von unmanaged Code zu managed Code . . . . .	43
25	Storyboards . . . . .	54
26	Quick Look . . . . .	56
27	Beispiel für unseren GridHelper . . . . .	60
28	Das Model-View-Controller Pattern . . . . .	62
29	Ohne Mahapps . . . . .	63
30	Mit Mahapps . . . . .	63
31	Hotkey-Reiter im Einstellungsfenster . . . . .	65
32	Unschöne Darstellung von James in der <i>Windows Task Switcher</i> . . . . .	66
33	ASP.NET Core 1.0 im Vergleich zu ASP.NET 4.6 . . . . .	67
34	Grundsätzlicher Aufbau der config.json . . . . .	70
35	Windows: Generierter Dialog des Api-Triggers . . . . .	72
36	OS X: Generierter Dialog der Bash Action . . . . .	73