Und erlebe meinen ersten Begeisterungssturm über Rubys Fähigkeiten.

Während ich mich nämlich mühselig – Schrittchen für Schrittchen für Schrittchen – durch Railsdokumente und -Sourcen, durch Ruby-Handbücher und Tutorials durchwurstele, mit automatischer Initialisierung (oder deren Fehlen) von diversen Variablenarten kämpfe, feststelle, was Rails so alles an Hintergrundarbeit treibt, um es mir einfach zu machen (und was der Preis dafür ist, wie beispielsweise ständig neue Instanziierungen all der hübschen Railsklassen) ...

... während also die Zeit vergeht, wo ich mir vorkomme wie ein blinder Maulwurf in der hellen Sonne ...

... verfalle ich irgendwann auf die Idee, einen meiner Lieblingstricks aus meiner alten 4GL auszuprobieren. Dort hatte ich sehr ausgiebig mit „Indirektionen" gearbeitet, also mit Texten, die von dieser 4GL ausgewertet und dann in die jeweiligen Objekte oder deren Methoden übersetzt wurden. Dies hatte den Vorteil, dass sich ganze Programmbausteine als Textelemente auslagern (und damit in externen Datenbanken speichern) ließen – sehr praktisches Feature.

Programmbausteine als Textelemente

Dies kam mir nun in den Sinn, als ich die die Dateien meiner Datenbank einlas und aufarbeitete – denn natürlich macht es auf Dauer wenig Sinn, nur eine einzige Datenbank zu betrachten und ich rechnete genauso natürlich schon damit, dass Rails es mir später auch recht bequem erlauben würde, weitere Systeme zu integrieren ...

... also wollte ich den Source-Code dafür bereits empfänglich machen – wenn ich schon dabei bin, frischen Code zu erstellen, kann ich auch gleich versuchen, ihn so flexibel zu gestalten, dass ich dereinst keine (oder kaum) Arbeit mehr damit habe ...

... was wiederum bedeutete, dass ich den Aufruf der Funktion „database" im Application-Controller „offen halten" wollte, der zu jenem Zeitpunkt noch die angebundenen Dateien meiner aktuellen Rails-Connection „Datei" über das Modell „Datei" einlas:

```
@database, notice = Datei.database(notice, update)
```

Also fügte ich eine kleine lokale Variable „conn“ ein, die mit dem Klassennamen „Datei“ meines Modells vorbelegt wurde und diesen ersetzte:

```
@database, notice = conn.database(notice, update)
```

was ganz genauso natürlich zu einer Fehlermeldung seitens Ruby führte, weil die angesprochene Methode selbstverständlich keine Methode eines Strings („conn“) ist. Einer plötzlichen Eingebung folgend schrieb ich also:

```
@database, notice = eval(conn).database(notice, update)
```

und das funktionierte!

Zuerst funkte es noch gar nicht bei mir, zu mühselig ist die Kleinarbeit, sich Befehlszeile für Befehlszeile durch unbekanntes Terrain voran zu kämpfen ...

... doch allmählich trat die Begeisterung ein – denn genau mit dieser Fähigkeit hat Ruby ein Versprechen bereits eingelöst, das ich mir von dieser Sprache gemacht hatte: Per Textbausteinen programmieren zu können, was, wie gesagt, ein wesentlicher Bestandteil meiner Kernel-Architektur und beileibe nicht selbstverständlich für viele Sprachen ist, die ich kenne – Ruby hat es ganz beiläufig, ohne viel Aufhebens getan.

Was wohl zuviel von Ruby verlangt wäre, ist, Klassennamen auch hinsichtlich der eigenen (Rails-)Applikation auf Widerspruchsfreiheit zu überprüfen – so stolpere ich wieder über eine Namensgebung, die ich besser gelassen hätte: „Data“ mit der interessanten Meldung, dass „allocate“ nicht definiert ist dafür.

Dabei habe ich diese Wortwahl sogar gezielt ausprobiert, um den Begriff „Database“ zu umgehen. Obwohl jener Begriff nur in MySQL reserviert ist und damit im Ruby/Rails-Kontext natürlich hätte verwendet werden können, meide ich solche Schlüsselworte jetzt generell, um nicht jedesmal prüfen zu müssen, welches Wort wo gültig ist.

Reservierte Worte generell meiden

Warum ich eine eigene Klasse, die nun „datenbank“ heißt, eigentlich brauche?

Weil ich eine „normale“ Klasse außerhalb von Rails benötige, deren vollständiger Lebenszyklus sich unter meiner Kontrolle befindet – ich will den Inhalt der eingelesenen Datenbanken nicht immer wieder neu ermitteln, doch die

ständige Initialisierung der Rails-Klassen (die in Ruby auch nichts anderes als Objekte sind) verhindert eine einfache Zwischenspeicherung in Klassenvariablen. Zwar bewahren die Helfer-Module von Rails auch Klassenvariablen auf, doch ziehe ich im Moment noch „richtige Objekte“ vor.

Außerdem ...

... wer weiß, wozu es gut ist, dass nun die gesamte Einlese-Logik ausgelagert ist.

Caching-Strategien auf die spätere Produktiv-Umgebung ausrichten.

Nachtrag:
Zu beachten ist, dass die Lebensdauer auch der eigenen Klassen und Klassenvariablen von der Art ihrer Speicherung abhängt.

Eine weitere Eigenschaft Rubys als Skriptsprache mag für Programmierer interessant sein, die bisher nur Compilierfehler bei mangelhafter Syntax kennen. So stolperte ich bereits recht früh über die Tatsache, dass gerade die für mich wichtigen Angebote von Rails (auf Dateien zuzugreifen) sehr von der jeweiligen Datenbank abhängen – was sich aber einfach bedingen lässt:

```
attributes["datenbank"] =
        Datei.connection.current_database
        if attributes["adapter"] == "MySQL"
```

Da die Variable „current_database“ nur beim MySQL-Adapter vorhanden ist, umgeht die bedingende Anweisung

```
if attributes["adapter"] == "MySQL"
```

einfach den Zugriff darauf: Ist die Prüfung nämlich nicht erfolgreich, überspringt Ruby den Rest insgesamt und stößt damit erst gar nicht auf den Bezeichner, der bei einem anderen Objekt als MySQL nicht existieren würde und damit einen Fehler provozieren müsste.

Trotz der ansprechenden Kürze entfernte ich diese Abfrage später wieder, als ich feststellte, wie viele der für mich maßgeblichen Funktionen datenbankabhängig waren – also beschloss ich, mich vorerst auf MySQL zu beschränken.

Und so geschah es, dass ich tatsächlich meinen ersten Zwischenschritt erfolgreich absolvierte – das Einlesen einer angebundenen Datenbank und das Abspeichern der interessierenden Daten in meiner Datei: die Klasse „datenbank“.