KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

02/2013

Jahrgang 6

KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

Sie können die elektronische Form des KaffeeKlatschs
monatlich, kostenlos und unverbindlich
durch eine E-Mail an

abo@bookware.de

abonnieren.

Ihre E-Mail-Adresse wird ausschließlich für den Versand
des KaffeeKlatschs verwendet.

Editorial

Immer wieder neu

Alle Jahre wieder steht die CeBIT vor der Tür. Anfang März öffnet *die* IT-Messe schlechthin ihre Pforten. Prominenz kommt und verkündet, dass dies und jenes den Segen der Politik hat, gut für die Wirtschaft ist oder das Leben besser macht. Als ob die davon wirklich etwas verstehen würden.

So wird auch diese Messe wieder irgendwelche *Gadgets* und *Gizmos* bringen. Vielleicht ein kleines Handy oder ein großes oder ein ganz billiges. Oder ein neues Smartphone, das klüger ist als die meisten seiner Benutzer. Ein Roboter vielleicht, der alte Menschen pflegen kann oder den Familienhund ersetzt.

Für uns spielt das Alles eigentlich gar keine Rolle. Ganz unten hat sich nichts geändert. Unten ist eine Von-Neumann-Maschine mit mehr oder weniger Registern, mehr oder weniger linearem Speicher und einem ausreichenden Befehlssatz.

Nicht, dass wir mit diesen Dingen direkt zu tun hätten, denn schließlich haben wir unsere Programmiersprachen. Sie abstrahieren die zugrunde liegende Architektur, vereinheitlichen die verfügbaren Befehle und erlauben das Überleben von Software über die Hardware-Generationen hinweg.

Natürlich gibt es da auch Änderungen. So haben sich etwa in der jüngeren Vergangenheit die Speichermodelle geändert und sind deutlich präziser geworden. Nur dadurch ist es uns möglich, Software so zu schreiben, dass sie auch einem nebenläufigen Kontext standhält. Aber im Wesentlichen ist alles beim Alten geblieben.

Vielleicht müssen wir aber den Dialekt ändern, etwa von *Java* nach *C#*, von *C++* nach *Objective-C* oder von *Haskell* zu *Scala*. Wenn wir verstanden haben, worum es eigentlich geht, bedeutet das nur, dass groß geschrieben wird was vorher klein war, dass gruppierende Klammern eckig statt geschweift sind, Zeichen anders auf Zeilen verteilt werden und die *Compiler* andere Optionen haben.

Vielleicht mögen die Bibliotheken eine Rolle spielen, so dass die implementierten *Patterns* wie *MVC* durch eine andere Buchstabenkombination wie *MVVM* ersetzt werden. Also alles in allem nichts, was man nicht in wenigen Tagen umlernen könnte.

Das soll nicht etwa heißen, dass Neuerungen keine Herausforderungen mit sich bringen. Schließlich muss man neue Erfahrungen sammeln, bis man sie gewinnbringend und qualitätssichernd einsetzten kann.

Also kann man gespannt sein, was es denn dieses Jahr wieder Neues gibt. Aber im Grunde ist es nur gut, dass für uns alles mehr oder weniger beim Alten bleibt.

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 6, Nummer 2, Februar 2013
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Editorial 3

Beitragsinfo 4

Inhalt 5

Leserbriefe 6

User Groups 27

Werbung 29

Das Allerletzte 30

Artikel

Zweites Date mit Nancy

und Web-Entwicklung mit .NET macht wieder Spaß ... 8

Der Typ ist sicher!

Constraint Code Generator für Java 14

Kolumnen

Bloomige Angelegenheit

Des Programmierers kleine Vergnügen 22

Am bestensten gemeint und doppeltgemoppelt

Deutsch für Informatiker 24

Machtworte

Kaffeesatz 26

Zweites Date mit Nancy

und Web-Entwicklung mit .NET macht wieder Spaß

von Timothée Bourguignon 8

Sagen wir mal, dass wir eine *Web-API* für eine bestehende Datenbank bauen wollen – und zwar schnell – und dass wir dafür *.NET* verwenden möchten. Wir könnten *ASP.NET* (*MVC3* oder *MVC4 Web API*)oder lieber ein *Lightweight Framework* wie *Nancy* verwenden. Im letzten Artikel wurde gezeigt wie man *Nancy, Simple.Data* und *MongoDB* kombinieren, konfigurieren und miteinander verbinden kann. Dabei konnte man sehen, wie Nancy auf eine bestimmte *Route* reagiert, einen *View* anzeigt und View-Daten von und zum View übermittelt. Ebenso wurde gezeigt wie man Simple.Data verwendet, um Daten zu schreiben bzw. von einer Datenbank zu lesen. Dazu wurde MongoDB genutzt und demonstriert, wie über die *Interactive Shell* Objekte hinzugefügt und gelesen werden können. Das Ganze innerhalb von zehn *C#*- und kaum mehr *cshtml*-Zeilen. Das war ein erster Blick auf den „Super-Duper-Happy-Pfad“. Ich nehme an, dass wir diese Kenntnisse anwenden wollen!

Der Typ ist sicher!

Constraint Code Generator für Java 14

von Heiner Kücker

Moderne Programmiersprachen wie *Haskell* oder *Scala* beeindrucken mit hochentwickelten Typ-Systemen und versprechen die Vermeidung von Fehlern bereits zur Kompilierzeit. Mit einem Code-Generator ist manches davon auch mit dem (guten) alten *Java* möglich.

Leserbriefe

Leserbrief

von Steffen Guhlemann

bezogen auf den Artikel

Var-um?

Über Sinn und Unsinn variabler Typen in C#

von Tobias Krügel

KaffeeKlatsch 2012/10

Ich bin ebenfalls ein Freund starker Typisierung und habe die *Resharper*-Empfehlung lange bewusst ignoriert. Im Laufe der Zeit sind mir jedoch mindestens 2 Situationen aufgefallen, in denen die Verwendung von *var* sinnvoll ist. In einer davon wird durch die Verwendung von *var* sogar die Typsicherheit erhöht.

1. In *foreach*-Schleifen würde ich für die Schleifen-Laufvariable immer *var* verwenden. Dies erhöht die Typsicherheit.

Das Problem liegt darin, dass *foreach* – zumindest unter Umständen – die untypisierte Variante von *IEnumerable* benutzt.

Beispiel:

Angenommen es existiert eine Klasse *Person* und *Abteilung*:

```
class Person {
  public void A(){...}
  public void B(){...}
}
class Abteilung {
  public IEnumerable<Person> GetMembers(){...}
}
```

Client-Code, der etwas mit Personen macht, sieht dann z. B. so aus:

```
foreach(Person person in abteilung.GetMembers())
person.A();
```

In unserer Firma hat ein Entwickler im Rahmen eines *Refactoring* versucht überall anstelle konkreter Klassen *interfaces* einzusetzen, die dann auch nicht alle Methoden haben. Also:

```
interface IPerson {
  void B();
}
interface IAbteilung {
  IEnumerable<IPerson> GetMembers();
}
```

Wenn man jetzt über *Compiler*-Fehler versucht herauszufinden, welche Methoden der Klasse in das Interface müssen, bekommt man keine. Der Compiler erkennt nicht, dass

```
foreach(Person person in abteilung.GetMembers())
person.A();
```

jetzt nicht mehr funktioniert.

GetMembers liefert jetzt *IPerson*-Objekte zurück und keine *Person*-Objekte, und *IPerson*-Objekte haben die Methode *A* zur Zeit nicht mehr.

Der Compiler nutzt hier die nicht generische Implementierung von *IEnumerable*, die eine Auflistung von *object* liefert und castet dann. Das gibt viele hübsche Laufzeit-*Exceptions*, wenn mal jemand eine andere Klasse als *Person IPerson* implementieren lässt.

Mit *var* lässt sich ganz einfach Typsicherheit herstellen:

```
foreach(var person in abteilung.GetMembers())
person.A(); //Compiler-Fehler,
```

da *person* jetzt vom Typ *IPerson* ist und *IPerson* keine Methode *A* hat.

2. Das zweite Einsatzgebiet von *var* ist nicht so klar vorteilhaft, aber ich bevorzuge es trotzdem.

Es erleichtert das Refactoring von *return*-Typen in verwendenden Methoden, wenn diese den *return*-Typ nur weiterreichen.

Gibt eine Methode z. B. bisher *IList<int>* zurück, die einen Zustand encodiert, welches jetzt in eine Klasse *state* gekapselt wird, so gibt es viele Stellen

```
IList<int> state=blub.GetState();
blob.SetState(state);
```

nach dem Refactoring.

```
State state=blub.GetState();
blob.SetState(state);
```

So muss ich zumeist manuell hunderte Code-Stellen ändern, an denen sich inhaltlich nichts geändert hat – don't repeat yourself.

Analog: bedingte Compilierung, die einfach zwischen 2 Typen umschaltet – einmal wird *A* und einmal *B* verwendet. Gemeinsames Interface funktioniert nicht. Beispielsweise ein Optimierungsprogramm, das schnell sein kann, wenig Speicher braucht, wenn es *float* verwendet aber dann recht ungenau wird. Oder aber es wird auf *double* umgestellt.

Ein paar Stellen brauchen wirklich Code ala

```
#if USE_FLOAT
using ValueType = System.Single;
#else
using ValueType=System.Double;
#endif
```

Aber die meisten Dateien in der selben *Assembly*, die diesen Code verwenden, sind gut mit *var* bedient.

Fazit

1. Aufgrund der blödsinnigen Implementierung von *foreach* durch den *C#-Compiler*, sollte *var* in *foreach*-Schleifen „immer" verwendet werden.

2. An anderen Stellen sind die Aspekte der Verwendung von *var* nicht nur Faulheit und geringere Lesbarkeit, sondern die leichtere Änderbarkeit von Code-Teilen, die sonst durch Repeat-Yourself entstehen würden. Im Gegensatz zu Copy-And-Paste-Code weist hier aber der Compiler auf die Notwendigkeit der Änderung hin, sodass jeder selbst abwägen muss.

Viele Grüße,
Steffen Guhlemann

Zweites Date mit Nancy

und Web-Entwicklung mit .NET macht wieder Spaß

von Timothée Bourguignon

Sagen wir mal, dass wir eine *Web-API* für eine bestehende Datenbank bauen wollen – und zwar schnell – und dass wir dafür *.NET* verwenden möchten. Wir könnten *ASP.NET* (*MVC3* [1] oder *MVC4 Web API* [2]) oder lieber ein *Lightweight Framework* wie *Nancy* [3] verwenden. Im letzten Artikel [4] wurde gezeigt wie man *Nancy, Simple.Data* [5] und *MongoDB* [6] kombinieren, konfigurieren und miteinander verbinden kann. Dabei konnte man sehen, wie Nancy auf eine bestimmte *Route* reagiert, einen *View* anzeigt und View-Daten von und zum View übermittelt. Ebenso wurde gezeigt wie man Simple.Data verwendet, um Daten zu schreiben bzw. von einer Datenbank zu lesen. Dazu wurde MongoDB genutzt und demonstriert, wie über die *Interactive Shell* Objekte hinzugefügt und gelesen werden können. Das Ganze innerhalb von zehn *C#*- und kaum mehr *cshtml*-Zeilen. Das war ein erster Blick auf den „Super-Duper-Happy-Pfad". Ich nehme an, dass wir diese Kenntnisse anwenden wollen!

Innerhalb eines halben Jahres hat sich in diesen Projekten viel getan. Simple.Data hat seit Juli drei *v1.0 Release Candidates* ausgerollt, den letzten im November 2012. Die Nancy Community hat neun (inkl. vier *Major*-) Versionen mit vielen Neuigkeiten herausgebracht und Mitte Februar die *v0.16* veröffentlicht. Eine dieser Nancy Neuigkeiten werden wir uns hier genauer anschauen. Die *Content Negotiation*, eine sehr nützliche Funktionalität um APIs zu bauen... Perfekt!

Unser Heutiges Ziel

Als Beispiel werden wir eine Postleitzahl-API anhand von Nancy, Simple.Data und MongoDB bauen. Bei Angabe einer Postleitzahl wollen wir den Städtenamen zurückliefern, oder anhand des Städtenamens die Postleitzahl. Falls die Abfrage mit einem Browser gemacht wird (der *Content* also *text/html* ist), wollen wir auf einer Karte (*GoogleMaps*) anzeigen, wo sich die Stadt befindet. Wenn nach einem *JSON*- oder *XML*-Extrakt gefragt wird, wollen wir die Daten entsprechend zurückliefern; diese können dann etwa in einem *Rich-Client* für die Postzustellung, die Geo-Lokalisierung oder ähnlich genutzt werden.

Plan C: „Not Invented Here"-Syndrom, d.h. „ich mach lieber alles selbst"!

Im letzten Artikel haben wir eigentlich schon fast alle Konzepte gesehen, die wir benötigen um eine API bauen zu können. Wir können Routen definieren. Wir wissen wie die MongoDB-Datenbank mittels Simple.Data zu erreichen ist. Wir wissen wie man dynamische Objekte mit Simple.Data binden und an Nancy zurückliefern kann. Wir haben gesehen, wie man einen View baut und Daten hin und her schiebt. Wir wissen also fast alles.

Die einzige richtige Neuigkeit wäre den *http-Header* zu untersuchen, um herauszufinden welches Format gerade gewünscht ist. XML/JSON-Serialisierung ist nichts Neues, darum müssten wir uns aber trotzdem kümmern. Wenn wir auch auf Datei-*Extensions* reagieren wollen, sodass *mydomain/irgendeinURI.json* oder *mydomain/irgendeinURI.xml* die richtigen Daten zurückgeben,

müssten wir es auch „zu Fuß“ mit anderen Routen machen.

Wie gesagt, mühsam, würde aber klappen.

Plan B: „Das Rad muss man nicht immer neu erfinden“!

Den Serialisierungs-Teil bietet Nancy eigentlich schon. Dafür müssen wir uns die Struktur der zurückgegebenen Objekte etwas genauer anschauen.

Im letzten Artikel haben unsere Routen drei Typen von Objekten zurückgegeben:

- Zuerst haben wir im „Hallo Welt!“-Beispiel den *String* „Hello world!“ zurückgeliefert. Diese Zeichenkette wird von Nancy als <*body*> einer *HTML*-Seite interpretiert und zurückgeliefert.
- Dann, um einen View anzuzeigen, haben wir *View["viewName"]* zurückgegeben. Der View wird durch die geladene *ViewEngine* behandelt (anhand der View Datei-Extension).
- Zuletzt, um eine Weiterleitung durchzuführen, haben wir *RedirectResponse("/")* verwendet. Dieses *Response*-Objekt wurde direkt bei Nancy verwendet und es ist eine verzauberte Weiterleitung passiert!

Nota: Aus *RedirectResponse* ist jetzt in der neuen Version von Nancy *Response.AsRedirect("/")* geworden.

Im ersten und dritten Fall haben wir implizit und explizit ein *Response*-Objekt zurückgegeben. Ein Response-Objekt kapselt ein *http-Response*-Objekt, kümmert sich um verschiedene Umwandlungen (z. B. vom *Integer* zum *System.Net.HttpStatusCode*) und bietet eine Reihe von *Formatters*, die uns helfen diese Antwort zu generieren:

- Response.AsRedirect()
- Response.AsFile()
- Response.AsText()
- Response.AsImage()
- Response.AsJson()
- Response.AsXml()

All diese Formatters liefern eine spezialisierte Version eines Response-Objekts. Damit könnten wir unsere Serialisierung schon abdecken. Wir holen unsere Daten von der Datenbank mit Simple.Data und geben diese via *Response.AsJson()* oder *Response.AsXml()* ab.

Bezüglich Plan C bleiben noch der Header und die Extensions zu schreiben... und schön wäre, wenn Nancy da was für uns hätte.

Plan A: „Nancy, kannst du es bitte für mich machen?“

Wie schon erwähnt, hat Nancy einige Neuigkeiten im Bauch. Eine davon würde dieses ganze Szenario deutlich erleichtern, dass so genannte *Content Negotiation-Feature*.

Content Negotiation ist ein Weg um festzustellen, welche Art von Daten wir zurückliefern. Der Client sagt was er gerne hätte, der Server sagt was er zu bieten hat, die Content Negotiation entscheidet.

Sobald man etwas anderes als ein Response-Objekt (oder irgendwas, dass direkt in ein Response-Objekt implizit umgewandelt werden kann) zurückgibt, wird dieses Objekt durch die *Content Negotiation Pipeline* geschleust.

Da die Views keine Response-Objekte sind, werden sie bei dem *Default ViewProcessor* so behandelt. Dieser *Default Response Processor* kommt mit zwei anderen, nämlich einem *JsonProcessor* und einem *XmlProcessor* – also alles was wir an dieser Stelle brauchen. Bräuchten wir ein anderes Format, müssten wir nun eine dedizierte Klasse, die *IResponseProcessor* implementieren würde, bauen. Wie jedes Nancy-Objekt, wäre dieser Prozessor dann automatisch beim *Nancy-Bootstrapper* geladen.

Mit dieser Negotiation können wir zum Beispiel für die Route */customer/3* die folgenden Szenarien sehen:

- Ohne zusätzliche Information und dem Default Content-Type *text/http* wird ein View zurückgeliefert.
- Fügen wir im Header einen Accept-Block hinzu, z. B. *Accept*: *application/xml*, wird ein XML-Extrakt zurückgegeben.
- Ergänzen wir den URI mit *.json* also */customer/3.json*, dann wird der Accept-Teil des Headers ignoriert und wir bekommen ein JSON-Extrakt.

Zurück zu unserer Postleitzahlen-Datenbank

Auf der MongoDB-Webseite finden Sie eine JSON-Datei [7], die wir in MongoDB importieren können. Zuerst laden wir diese Datei runter, starten den MongoDB-Dienst (*mongod.exe*) in einer *Command Shell* und importieren unsere JSON-Datei via dem Kommando:

```
mongoimport --db zipDB --collection zips --file zips.json
```

Der Import läuft sofort los und sollte die Anzahl der importierten Objekte (29470) in wenigen Sekunden anzeigen. Ok, unsere Mongo-Datenbank ist bereit!

Nancy, du bist dran

Unsere Datenbank haben wir bereits aufgesetzt. In *Visual Studio* erzeugen wir ein neues *ASP.NET Empty Web Application*-Projekt mit dem Namen „ZweitesDateMitNancy“. Via Package Manager Console von *Nuget* fügen wir die Dependencies hinzu:

- Nancy installieren wir via *Install-Package Nancy.Hosting.AspNet* (d. h. wir werden also *IIS* [8] verwenden),
- Simple.Data und MongoDB Driver via *Install-Package Simple.Data.MongoDB* und
- Razor via *Install-Package Nancy.ViewEngines.Razor*.

Wir brauchen 4 neue Dateien:

- *ZipsService.cs*: der Kern unserer Anwendung
- *CityInfo.cs*: die Container-Klasse für die Daten, die wir aus der Datenbank lesen
- *SingleView.cshtml*: unser View zur Anzeige von GoogleMaps
- *MultipleView.cshtml*: der View, in dem wir eine Liste unserer Ergebnisse anzeigen

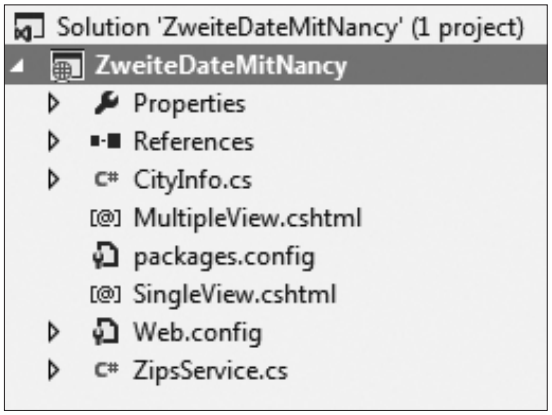


Abbildung 1: VisualStudioSolution

Als Routen schlage ich folgende vor:

- *GET /zip/{id}*
- *GET /city/{name}*

Natürlich könnte unsere Anwendung / API andere Routen anbieten, wie "Alle Postleitzahlen eines bestimmten Staates" oder "Alle Postleitzahlen, in denen die Bevölkerung größer als X Personen ist"; für dieses Beispiel halten wir es aber kurz und knackig!

Code Scaffolding

Unsere *ZipsService*-Klasse sieht also so aus:

```
public class ZipsService: NancyModule {
  public ZipsService () {
    Get["/zip/{id}"] = parameters => {
      return "The route '/zip/" + parameters.id + "' was called";
    };
    Get["/city/{name}"] = parameters => {
      return "The route '/city/" + parameters.name + "'
      was called";
    };
  }
}
```

Schon können wir das Projekt kompilieren und nachprüfen, ob die Routen die verschiedenen Strings richtig anzeigen.

Nota: Falls ein *Internal Server Error* wegen doppelter <*compilation*>-Blöcke in der *web.config*-Datei kommen sollte, einfach den <*compilation debug="true" targetFramework="4.5"/>* Abschnitt in dieser Datei entfernen.

Um die JSON-Daten, die wir importiert haben zu reflektieren, brauchen wir eine *CityInfo*-Klasse:

```
namespace ZweitesDateMitNancy
{
  public class CityInfo
  {
    public string Id { get; set; }
    public string city { get; set; }
    public int pop { get; set; }
    public string state { get; set; }
    public double[] loc { get; set; }
  }
}
```

Wenn wir *city* mit ein Großen „C“ schreiben, würde das Auto-Binding von Simple.Data nicht mehr funktionieren: die Schreibweise ist also wichtig!

In unserer sehr übersichtlichen *SingleView* listen wir die Properties auf und rufen die *GoogleMaps API v2* (es zeigt ein statisches Bild an):

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>City Locator</title>
  </head>
  <body>
    <h1>City Locator</h1>
    <ul>
      <li>City: @Model.city</li>
      <li>Zip: @Model.Id</li>
      <li>State: @Model.state</li>
```

```
            <li>Population: @Model.pop</li>
            <li>Coordinates: @Model.loc[1], @Model.loc[0]</li>
          </ul>
          <img src="http://maps.googleapis.com/maps/api/
          staticmap?center=@Model.loc[1],@Model.loc[0]
          &zoom=10&size=800x600&sensor=false">
        </body>
      </html>
```

In der *MultipleView* listen wir die verschiedenen Treffer mit ihren Properties, aber ohne das Anzeigen von GoogleMaps:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>City Locator</title>
  </head>
  <body>
    <h1>City Locator</h1>
    @foreach (dynamic cityInfo in Model)
    {
      <ul>
        <li>Zip: @cityInfo.Id</li>
        <li>Population: @cityInfo.pop</li>
        <li>Coordinates: @cityInfo.loc[1], @cityInfo.loc[0]</li>
      </ul>
    }
  </body >
</html >
```

To the point!

Achtung, es wird relativ unspektakulär:
Zuerst müssen wir unsere Daten von der Datenbank lesen (nicht vergessen, dass der MongoDB-Dienst laufen muss). Dafür brauchen wir ein Datenbank-Objekt, das wir direkt als *private Member* in der *ZipsService*-Klasse hinzufügen. Dann können wir unsere Datenbankabfrage nach *Id* und *Name* ausführen. Um nachzuprüfen, dass alles funktioniert, können wir die Views anzeigen. Die *ZipsService*-Klasse sieht dann so aus:

```
public class ZipsService: NancyModule {
  private const string ConnectionString =
    @"mongodb://localhost:27017/zipDB";
  private readonly dynamic db =
    Database.Opener.OpenMongo(ConnectionString);
  public ZipsService () {
    Thread.CurrentThread.CurrentCulture =
      CultureInfo.CreateSpecificCulture("en-GB");
    Get["/zip/{id}"] = parameters => {
      var cityInfo = db.zips.FindById(
        parameters.id.ToString()
      );
      return View["SingleView", cityInfo];
    };
    Get["/city/{name}"] = parameters => {
      var cityInfoList =
        db.zips.FindAllBy(
          city:parameters.name.ToString()
        );
      return View[
        "MultipleView", cityInfoList.ToList()
      ];
    };
  }
}
```

Hier sind die verschiedenen Tests, die wir ausprobieren können:

- Wenn wir z. B. die Route */zip/60623* aufrufen, landen wir in Chicago.
- Wenn wir die Route */city/CHICAGO* aufrufen, bekommen wir eine Liste aller zugehörigen Postleitzahlen.
- Rufen wir aber */zip/60623.json* auf, bekommen wir eine schöne Fehlermeldung. Wir haben also noch keine Negotiation.

Abbildung 2: Chicago Google Maps

Nota: Die GoogleMaps API benötigt Koordinaten, die mit einem '.' formatiert sind. Die deutsche Repräsentation von *Float*-Objekten nutzt bekanntlich ein ','. Aus diesem Grund erzwinge ich die englische Kultur.

Nota: Die ID ist als String in der Datenbank abgelegt, daher müssen wir *ToString()* auf den Parametern der *FindById()*-Funktion aufrufen. Die ID ist in dem JSON-Model, das wir importiert haben und entspricht dem *zipCode*.

Nota: Die *FindAllBy()*-Funktion prüft genau den String, den wir angeben. Also werden „Chicago“ und „CHICAGO“ unterschiedlich behandelt. Genau so können wir in der Form dieses Beispiels „LOS ANGELES“ nicht suchen, da dieser vom Browser in „LOS%20ANGELES“ umgewandelt wird.

Don’t shoot, we want to negociate!

Um die Negotiation zu verwenden, benutzen wir die Negotiate-Helferklasse. Diese müssen wir anhand von Extension-Methoden konfigurieren. Mit dieser Konfiguration werden wir quasi der Negotiation-Pipeline sagen, auf welche Art und Weise die Aushandlung passieren soll.

Unsere Klasse sieht dann so aus:

```
public class ZipsService : NancyModule {
  private const string ConnectionString =
    @"mongodb://localhost:27017/scratch";
  private readonly dynamic db =
    Database.Opener.OpenMongo(ConnectionString);
  public ZipsService () {
    Thread.CurrentThread.CurrentCulture =
      CultureInfo.CreateSpecificCulture("en-GB");
    Get["/zip/{id}"] = parameters => {
      CityInfo cityInfo =
        db.zips.FindById(parameters.id.ToString());
      return Negotiate.WithModel(cityInfo)
      .WithView("SingleView");
    };
    Get["/city/{name}"] = parameters => {
      IList<CityInfo> cityInfoList =
        db.zips.FindAllBy(
          city: parameters.name.ToString()
        )
      .ToList<CityInfo>();
      //nicht vergessen, wir benötigen System.Linq
      return Negotiate.WithModel(cityInfoList)
       .WithView("MultipleView");
    };
  }
}
```

Jetzt können wir *.json* und *.xml* auf unsere Routen aufrufen und die entsprechenden JSON- und XML-Fragmente werden zurückgeschickt.

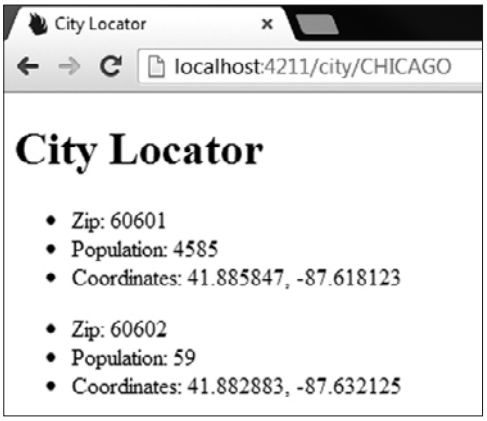


Abbildung 3: citylocatorCHICAGO

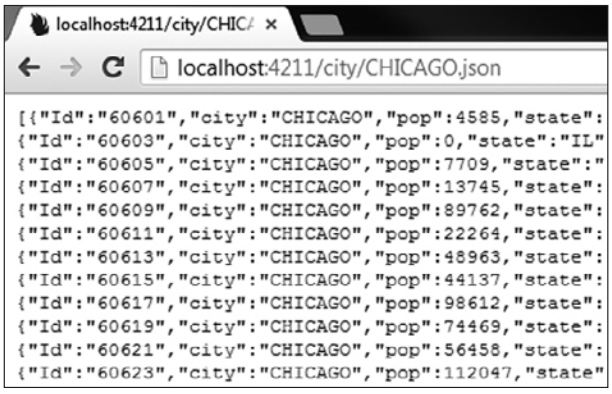


Abbildung 4: citylocatorCHICAGO.JSON

Mit *Fiddler* [9] (oder einem ähnlichen Tool) können wir auch *Accept: application/xml* oder *Accept: application/json* in den HTTP-Header hinzufügen und die Standard-Route aufrufen */zip/{id}*. Dann bekommen wir auch die entsprechenden JSON- und XML-Extrakte.

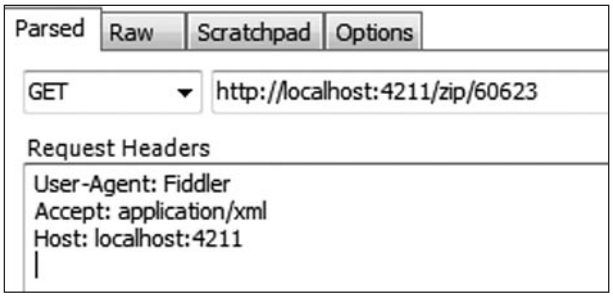


Abbildung 5: fiddlerrequest

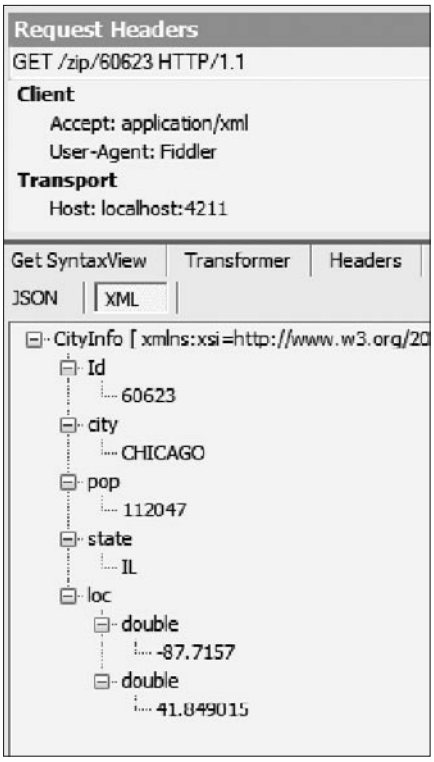


Abbildung 6: fiddlerresponse

Nota: Wahrscheinlich ist aufgefallen, dass wir jetzt nicht mehr *var cityInfo* und *var cityInfoList* sondern „static typing" *CityInfo cityInfo* und *IList<CityInfo> cityInfoList* nutzen. Das müssen wir so implementieren, weil die *WithModel(parameter)*-Methode eine Extension-Methode ist und daher ein statisches Objekt braucht. Es ist sicher nicht mehr so praktisch wie bei *return View['ViewName', DynamicObject]*, aber trotzdem sehr nützlich.

Nota: Warum verwende ich hier ein *IListe* anstatt einer *IEnumerable* Kollektion? Da wir eine XML-Serialisierung durchführen, brauchen wir – laut *MSDN* – eine vollständige Implementierung des *IEnumerable Interface*. Wenn wir ein *IEnumerable* verwenden und z. B. *city/CHICAGO.xml* aufrufen, bekommen wir eine Fehlermeldung: „To be XML serializable, types which inherit from IEnumerable must have an implementation of Add(System.Object) at all levels of their inheritance hierarchy".

Zusätzlich zu den *WithModel()* und *WithView()*, könnten wir die Negotiation anhand anderer Extension-Methoden steuern. Zum Beispiel mit *WithStatusCode*, um der Antwort einen Status-Code hinzuzufügen, nachdem die Negotiation passiert ist. Oder *WithHeaders* um eine Kollektion von *Headers* und *Values* an die Antwort hinzuzufügen. Mehr Informationen sind auf Nancy's Wiki unter Content Negotiation zu finden [10].

Abschiedskuss?

In diesen Artikel haben wir gesehen, wie uns Nancy anhand einiger Helfer-Methoden mächtige Konzepte bietet. Die Content Negotiation hilft uns den *Return Type* unserer Routen zu steuern und damit eine kleine API mit (sehr) wenig Codezeilen zu schreiben. In Nancys Schatten bietet Simple.Data wieder einmal eine sehr einfache Art und Weise, um Datenbankabfragen auszuführen – ohne Zeremonie, aber auf sehr klare Art.

Webseite und APIs, was könnten wir denn sonst mit Nancy bauen?

Nota: Das Beispiel kann man auf Github [11] finden.

Referenzen

[1] ASP.NET *ASP.NET MVC3*
http://www.asp.net/mvc/mvc3

[2] ASP.NET *MVC4 WebAPI*
http://www.asp.net/web-api

[3] NancyFx *Leightweight Web Framework for .NET*,
http://nancyfx.org

[4] Timothée Bourguignon *Der Super-Duper-Happy-Pfad, 2 1/2 Blicke auf großartige Open-Source-Frameworks für .NET*
http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2012-07.pdf

[5] GITHUB *Simple.Data*
https://github.com/markrendle/Simple.Data/wiki

[6] MongoDB *Agile and Scalable*
http://www.mongodb.org

[7] MEDIA.MONGODB *JSON Beispiel Datei*
http://media.mongodb.org/zips.json

[8] ISS *Internet Information Services*
http://www.iis.net

[9] Fiddler *Introducing Fiddler*
http://www.fiddler2.com/fiddler2

[10] GITHUB *Nancy's Content Negotiation Wiki Page*
https://github.com/NancyFx/Nancy/wiki/Content-Negotiation

[11] GITHUB *Code on Github*
https://github.com/Timothep/ZweiteDateMitNancy

Weiterführende Literatur

- Håkansson, Andreas *Persönlicher Blog*
http://thecodejunkie.com
- Rendle, Mark *Persönlicher Blog*
http://blog.markrendle.net
- Rendle, Mark *Introduction to Nancy and Simple.Data*
http://skillsmatter.com/podcast/open-source-dot-net/introduction-to-nancy-and-simple-data

Kurzbiographie

Timothée Bourguignon ist als Senior Developer für die MATHEMA Software GmbH tätig. Sein Spezialgebiet ist die Desktop- und Web-Programmierung mit dem .NET Framework. Daneben beschäftigt er sich mit den Neuerungen der .NET-Welt und deren Communities. Als Agilist liegt sein Schwerpunkt auf agilen Methoden, guter Teamarbeit und hoher Arbeitsqualität. Hierbei setzt er insbesondere auf die Philosophie des Software Craftmanships.

Der Typ ist sicher!

Constraint Code Generator für Java

von Heiner Kücker

Moderne Programmiersprachen wie *Haskell* oder *Scala* beeindrucken mit hochentwickelten Typ-Systemen und versprechen die Vermeidung von Fehlern bereits zur Kompilierzeit. Mit einem Code-Generator ist manches davon auch mit dem (guten) alten *Java* möglich.

Design by Constract

Zurückgehend auf die Ideen der Programmiersprache *Eiffel* von Bertrand Meyer gibt es verschiedene *Frameworks*, welche die Einhaltung von *Contracten* in Java-Programmen absichern.

Man kann dies auch einfach selbst machen:

```
public void(Sring str){
  if (str==null){
    throw new IllegalArgumentException("str is null);
  }
  ...weiterer Code...
}
```

Ein Problem dieser Lösungen ist, dass die Contracts nur zur Laufzeit geprüft werden. Nur durch einen ergänzenden Test können die Contracts zur Wirkung gebracht werden. Außerdem weiss der *Compiler* nichts von den Contracten und kann nicht prüfen, ob diese beim Aufruf eingehalten werden.

Java-Compiler

Der Java-Compiler bietet mit seinem Typ-System der *primitiven Typen*, Java-Klassen und *Generics* (Typ-Parameter) einiges an Sicherheit; die Ausdrucksmöglichkeiten sind aber begrenzt und der Aufwand, die fachlichen Probleme auf Vererbung und Generics abzubilden, ist recht hoch. Manches, wie expliziter Ausschluss oder Werte-Bereiche, lässt sich einfach nicht ausdrücken. Aber der Java-Compiler bietet mit seiner eingebauten Datenfluss-Analyse eine wichtige Infrastruktur, die wir für statische Code-Analysen nutzen können.

```
final int a = 0;
...irgendwelcher Code, aber alles in einer Methode...
x = a; // a ist ein int
```

Java-Enums (leider nicht verwendbar)

Stellen wir uns vor, es gäbe ein *Enum AmpelFarbe* mit den Werten ROT, GELB und GRUEN. Die Methode *fahre()* erlaubt aber nur GRUEN und ausnahmsweise GELB, wenn es schon zu spät zum Bremsen ist.

```
void fahre(AmpelFarbe ampelFarbe){
  ...
}
```

Weil alle Ampelfarben-Enum-Ausprägungen die gleiche Klasse haben, kann man das GRUEN- oder eventuell -GELB-*Constraint* damit nicht ausdrücken. Schade, Enums helfen uns hier nicht weiter.

Java-Interfaces (leider auch nicht geeignet)

Weil es mit den Java-Enums nicht geklappt hat, versuchen wir es mal mit *Interfaces*:

```
interface AmpelFarbe{}
interface ROT extends AmpelFarbe{}
interface GELB extends AmpelFarbe{}
interface GRUEN extends AmpelFarbe{}
```

Unsere Methode *fahre* schreiben wir so:

```
void fahre(GRUEN gruen){
  ...
}
```

Leider können wir nicht verhindern, dass die Klasse, welche GRUEN implementiert und der *fahre()*-Methode als Parameter übergeben wird, außerdem noch ROT implementiert. Die Zusatzbedingung an GELB können wir dem Java-Compiler so auch nicht mitteilen.

Constraint-Expressions

Ich war einmal in einem Projekt, in welchem der unterschiedliche Preis einer Leistung je nach Tag und Uhrzeit abgebildet werde musste.

Nehmen wir mal Strom als Beispiel. Nachtstrom ist billiger und für Firmen könnte auch der Strom an Sonn- und Feiertagen und in bestimmten Uhrzeit-Intervallen günstiger sein.
Um dies abzubilden, könnte man eine Datenbank-Tabelle mit einer Wochentags-Spalte, einer von- und bis-Uhrzeit-Spalte sowie einer Prozent-Spalte einrichten. Aber dann kann man gesetzliche Feiertage noch nicht abbilden. Das Schema wird immer komplizierter.

Falls man aber die Tarif-Intervalle über *Expressions*, also Ausdrücke mit UND-, ODER-, NOT- und Kleiner- sowie Größer-Als-Operanden formuliert, kann man eigentlich alles abbilden, wenn die entsprechenden Funktionen, zum Beispiel für gesetzliche Feiertage, zur Verfügung stehen.

Prädikate

Der hier vorgestellte Code-Generator beruht auf Boolean-Expressions, die sogenannte Prädikate verknüpfen. Ein Prädikat ist eine Instanz einer Klasse, welche eine bestimmte abstrakte Klasse mit einer *test*-Methode implementiert.

```
abstract public class Predicate<CT>{
  abstract public boolean test(final CT contextObj);
}
```

Da die Prädikate in Java geschrieben sind, gibt es keine Spezial-Syntax in Kommentaren oder Annotationen. Alle sprachlichen Möglichkeiten von Java sind in den Prädikaten nutzbar, egal ob dies Null-Prüfungen, Werte-Vergleiche oder Mindest-Listen-Größen sind. Zur Verknüpfung stehen die Operanden AND, OR, NOT sowie XOR zur Verfügung, wobei XOR nicht auf zwei Operanden beschränkt ist, sondern beliebig viele Operanden mit XOR verknüpft werden dürfen.

Kontext-Objekt

Die *test*-Methode der Prädikate bekommt einen über Generics festgelegten Parameter übergeben, welcher die zu prüfenden Werte/Objekte enthält. Dieses Kontext-Objekt kann eine primitive Klasse wie *Integer* oder *String*, aber auch ein komplexes fachliches Objekt sein bzw. als *Member* mitführen. Falls ein Prädikat mehrere verknüpfte Objekte/Werte prüfen soll, müssen diese in einem dieser Kontext-Objekte verpackt werden; die Verwendung von Prädikaten mit mehreren Parametern ist zur Zeit nicht vorgesehen.

In einer Web-Applikation könnte das Kontext-Objekt auch der *HTTP-Request* sein, in welchem sich bestimmte Parameter befinden sollen und über den man auch an die *Session* des aktuellen Benutzers herankommt.

Einfach Loslegen

Nachdem wir unsere Prädikate geschrieben haben, können wir unseren konkreten Code-Generator von der Klasse *AbstractConstraintCodeGenerator* ableiten (Code bitte von der unter [Links] angegebenen Webseite herunterladen, ist zu umfangreich für die Darstellung). Noch ein paar Festlegungen wie *Packages*, Pfade und schon können wir unsere Prädikate mit den zur Verfügung stehenden Methoden AND, OR, NOT und XOR verknüpfen. Diese Methoden arbeiten mit *Java-5-varags*, benötigen also keine *Array*- oder *Collection*-Parameter um beliebig viele Prädikate zu verknüpfen.

Pfade nochmal kontrollieren, den Parameter *delete UnusedConstraintJavaFiles* erst mal auf *false* setzen, auch eine Datensicherung kann nicht schaden, und schon kann das Generieren los gehen.

Constraint-Java-Klassen

Anhand der festgelegten Constraint-Expressions werden Java-Klassen generiert, deren Name die zugrundeliegende Expression abbildet:

and(**new** A() , **new** B()) wird zu ANDB_A_B_ANDE

wobei ANDB und ANDE für Beginn und Ende eines AND stehen. Ähnlich ist es beim OR und XOR. NOT_ ist das Präfix für negierte Prädikate oder Klammer-Abschnitte.

Constraints, die nur aus einem einzigen Prädikat bestehen, bekommen das Postfix Constraint, zum Beipiel *AConstraint*, damit Constraint und Prädikat nicht den gleichen Java-Klassen-Namen haben, was zur Verwirrung beim Import führen könnte.

Die generierten Constraint-Java-Klassen sind *final* und erben nur von *Object* (kann man schließlich nicht vermeiden). Sonstige Vererbung ist nicht vorgesehen.

Jede generierte Constraint-Java-Klasse besitzt eine *test*-Methode zum Prüfen der Einhaltung des Constraints und einen Konstruktor, in welchem eine eventuelle Constraint-Verletzung mit einer *IllegalArgumentException* abgewehrt wird.

Verwendung der generierten Klassen

Der natürliche Lebensraum der generierten Java-Constraint-Klassen ist der Methoden-Parameter.

```
public void execute(
   final ANDB_A_B_ANDE constraint
) {
   ...
}
```

Der Methodenkopf ist die Stelle, an welcher die aufrufende Seite das Constraint einhalten muss und sich die aufgerufene Seite auf das Constraint verlassen kann. Somit kann sich der Aufrufer darauf verlassen, dass die aufgerufene Methode mit dem Constraint klarkommt, also den gesamten erlaubten Wertebereich verarbeiten kann. Eine Methode gliedert nicht nur den Quellcode, sondern verkleinert auch den vom Compiler zu analysierenden Zustands-Raum.

Das explizite Ausformulieren eines Constraints an einer Methodengrenze hilft dem Compiler oder anderen statischen Checkern beim Problem der Entscheidbarkeit. Wenn ein Constraint nicht explizit ausformuliert, sondern im Code versteckt ist, zum Beispiel ein *int*-Wert, der irgendwo als Array-Index verwendet wird und somit nicht negativ sein darf, so muss der Checker dieses verborgene Constraint finden und zusätzlich dessen Einhaltung absichern, ähnlich wie die *Scala-Type-Inferenz* den korrekten Typ von Variablen finden muss.

Da meist die Methodendeklaration/-definition und der Methodenaufruf im Quellcode mehr oder weniger weit entfernt sind, ist hier eine statische Prüfung eine willkommene Hilfe.

Sicherer und unsicherer Bereich

Schützen wir jetzt mal eine Methode mit einem *int*-Parameter dahingehend, dass der Wert des *int*-Parameters nicht negativ sein darf.

```
void arrayAccess(
   final GreaterOrEqZeroConstraint constraint
){...
```

Die Einführung dieses Constraints wirkt viral, in jeder weiteren aufrufenden Schicht muss jetzt das Constraint statt des eventuell negativen *int*-Wertes übergeben werden. Doch irgendwo bekommen wir einen Wert, den wir nicht unter Kontrolle haben, vom Benutzer, der Datenbank, dem Netzwerk oder vom Dateisystem. Diesen Wert kann der Compiler nicht kontrollieren, er kennt ihn nicht.

Zum Erwerben des Constraints müssen wir den Konstruktor der Constraint-Klasse aufrufen, um eine Constraint-Instanz zu erzeugen. Der Konstruktor bekommt das Kontext-Objekt, zum Beispiel ein *Integer*, übergeben. Nach dem erfolgreichen Aufruf des Konstruktors befinden wir uns im sicheren Bereich. Falls der übergebene Wert ungültig ist, erhalten wir eine *IllegalArgumentException*. Vorher befinden wir uns im unsicheren Bereich.

Null-Problem(o)

Wenn das Constraint in der Methoden-Signatur als Parameter aufgeführt wird, kann es einfach dadurch umgangen werden, dass man ein *null* übergibt. Dies muss durch einen *Guard*

```
void xxx(YConstraint constraint){
  if (constraint == null){
    throw new IllegalArgumentException(
      "constraint is null"
    );
  }
  ...weiterer Code...
}
```

oder ein ergänzend eingesetztes Laufzeit-Constraint-System vermieden werden. Wenn das im Constraint enthaltene Kontext-Objekt im weiteren Programmverlauf verwendet wird, fällt die Übergabe eines *null* ebenfalls irgendwann im Test auf.

Konvertierungs-Methoden zum allgemeineren Constraint

Oben formuliertes Constraint

```
and( new A() , new B() )
```

ist kompatibel, also ohne Verletzung einer Bedingung umwandelbar, zu

```
new A()
```

sowie

```
new B()
```

Dies stellt der Code-Generator automtisch fest und erzeugt die entsprechenden Konvertierungsmethoden:

```
convertToAConstraint();
```

sowie

```
convertToBConstraint();
```

Die Voraussetzung für die Kompatibilität ist die Erfüllung der einfachen Implikation, in der Literatur meist als Doppelpfeil (*fat arrow*) dargestellt:

(A and B) => A

(A and B) => B

Wie ich oben schon ausführte, eignet sich die Java-Typ-Kompatibilität über Vererbung nicht für unsere Zwecke, die Konvertierungs-Methoden sind die einzige Möglichkeit der Constraint-Kompatibilität.

Spezialisierung von Constraints

Haben wir in einer Fassade Methoden mit spezielleren Constraints, die wiederum Methoden mit allgemeineren Constraints aufrufen, reichen die oben beschriebenen Konvertierungs-Methoden aus. Es kann aber vorkommen, dass in einer Methode über eine *if-else*-Kaskade oder *switch-case* in Wertebereiche bzw. Rechte/Rollen des ursprüglichen Constraints/Wertes spezialisiert werden muss, weil die aufrufende Methode sozusagen mehrfach verwendet wurde. Dafür gibt es im Constraint-Code-Generator die *SwitchDefinition*, welche das Generieren einer nicht-statischen inneren abstrakten Klasse in der Constraint-Klasse auslöst.

Die Benutzung einer abstrakten Klasse mit einer zu implementierenden Methode für jeden Zweig sichert ab, dass für keinen Zweig die Implementierung vergessen wird. Leider kann man mit Java-*switch-case* oder *if-else*-Kaskaden nicht über den Compiler ausschließen, dass ein Zweig vergessen wird.

Die generierte abstrakte innere Klasse wird mit

```
constraint.new XxxSwitch(){
   ...zu implemtierende Methodenrümpfe...
 }
```

angelegt, in der *Eclipse* wird das erforderliche Gerüst automatisch erzeugt.

Die BOOLSCHEN Ausprägungen der einzelnen Zweige dürfen sich nicht überlappen (müssen *disjoint* sein), damit nicht ein zufällig vor einem anderen Zweig geprüfter Zweig gewinnt. Die spezialisierenden *case*-Methoden bekommen einen Constraint-Parameter, welcher durch die *And*-Verknüpfung des ursprünglichen Constraints (welches spezialisiert wird) und des Zweig-Constraints erzeugt wird.

Dabei werden die definierten *Includes* und *Excludes* (siehe weiter unten) zur Vereinfachung des neuen spezialisierten Constraints benutzt. Falls nicht für jede BOOLSCHE Ausprägung des ursprünglichen Constraints ein gültiger Zweig exisitiert, wird eine zu implementierende (abstrakte) *caseDefault*-Methode erzeugt. Für die *caseDefault*-Methode wird kein Constraint-Parameter erzeugt.

Includes

Nehmen wir einmal an, wir hätten eine fachliche Hierarchie aus Teamleiter und Bearbeiter und zwei Geschäftsbereiche, z. B. Ladengeschäft und Online-Geschäft. Der Teamleiter-Ladengeschäft würde wahrscheinlich die außerdem vorhandenen Prädikate *EbeneTeamleiter* und *GeschaeftLaden* ebenfalls erfüllen.

Diese Konstellation kann man dem Code-Generator folgendermaßen mitteilen:

```
public final class TeamleiterLaden
extends PrimitivPredicate<T> {
  /**
  * Konstruktor.
  */
  public TeamleiterLaden() {
    super(
      //includePredicateSet
      CollectionUtil.hashset(
        new EbeneTeamleiter() ,
        new GeschaeftLaden()
      ) ,
      //safeValueSet
      null
    );
  }
  ...
}
```

Eigentlich ist das Prädikat *TeamleiterLaden* damit überspezifiziert und die includierten (oder sollte man besser sagen implizierten) Prädikate sind immer *And*-verknüpft. Die Includes sind nützlich, wenn solche überspezifizierten Prädikate aufgrund vorhandener Strukturen in Applikationen einfach praktisch sind (sparen Schreibaufwand). Die Abbildung einer Hierarchie, zum Beispiel *Team-Leiter* darf Vorgänge seiner untergeordneten Bearbeiter sehen, kann über die Includes nicht abgebildet werden. Dies muss explizit ausformuliert werden, weil das "Sehen" eventuell erlaubt ist, aber das "Weiterleiten" nicht, wenn der *TeamLeiter* in diesem Status/Workflowschritt (in *BPMN* die Schwimmbahn) nicht die entsprechende Berechtigung hat.

Bei der Generierung der spezialisierten Constraint-Parameter der *case*-Methoden des *AbstractSwitch* werden die includierten Prädikate aus den Constraint-Expressions entfernt, wenn diese bei jeder gültigen BOOLSCHEN Belegung von einem anderen includierenden Prädikat überdeckt werden. Das bedeutet, dass die spezialisierten Constraint-Expressions bei vorhandenen passenden Includes vereinfacht werden. Beim Durchlaufen aller BOOLSCHEN Ausprägungen im *BruteForceSolver* werden die

Includes beachtet. Es ist nicht möglich, dass ein Prädikat erfüllt ist, aber ein direkt oder indirekt rekursiv includiertes Prädikat nicht erfüllt ist.

Excludes

Im Exclude-Objekt werden Gruppen (*ExcludeGroup*) von sich ausschließenden Prädikaten aufgeführt. Wenn sich die Rollen Teamleiter und Bearbeiter ausschließen, kann dies im Exclude festgehalten werden, so muss man nicht "Teamleiter und nicht-Bearbeiter" bzw. "nicht-Teamleiter und Bearbeiter" formulieren, "Teamleiter" oder (ausschließend-alternativ) "Bearbeiter" ist ausreichend.

Genauso wie die Includes sind die Excludes eine Schreibvereinfachung und werden auch bei den spezialisierten Constraint-Parametern der *case*-Methoden der *AbstractSwitches* zur Vereinfachung verwendet.

Beim Durchlaufen aller BOOLSCHEN Ausprägungen im *BruteForceSolver* werden die Excludes beachtet. Es ist nicht möglich, dass zwei oder mehrere Prädikate erfüllt sind, die sich in der gleichen *ExcludeGruop* befinden. Dabei werden natürlich auch includierte Prädikate, die zu *ExcludeGroups* gehören, beachtet. Das Flag *isClosedWorldAssumption* bestimmt, ob mindestens ein Prädikat der *ExcludeGroup* innerhalb einer Constraint-Expression erfüllt sein muss.

Range-Check

Manchmal knabbert man an einem einfach aussehenden Feature wochenlang rum und plötzlich bekommt man ein spektakuläres Feature geschenkt. So ging es mir mit den Includes/Excludes und dem Range-Check.

Die Include-/Exclude-Vereinfachungen der Constraint-Parameter der *case*-Methoden der *AbstractSwitches* haben mich lange beschäftigt und ich war im Zweifel, ob sich der Aufwand lohnt. Eines Tages habe ich einem Kollegen meinen Code-Generator erläutert und zur Abgrenzung (dies kann der Code-Generator nicht) folgende Anforderung formuliert: Gegeben sei eine Zahl größer-als minus 10 und kleiner-als plus 10. Für diese soll ein *AbstractSwitch* für die Bereiche kleiner-als 0, gleich 0 und größer-als 0 generiert werden.

Constraint: $x > -10 \;\&\&\; x < 10$

Switch:
- Case $x < 0$
- Case $x == 0$
- Case $x > 0$

Nach dem Gespräch fiel mir auf, dass dies doch möglich ist. Drei Zutaten machen dieses Rezept schmackhaft:

- parametrisierbare Prädikate
- dynamisches Include
- dynamisches Exclude

Den Effekt oben nennt man übrigens Emergenz.

Parametrisierbare Prädikate

Bisher waren Prädikate einfach binär gültig oder nicht. Durch die Angabe eines Parameters kann ein Prädikat genauer spezifiziert werden und an mehreren Stellen in der Constraint-Definition unterschiedlich für Werte-/Bereichs-Prüfungen verwendet werden:

```
new IntGreater( –10 ) // x > –10
new IntLesser (  10 ) // x <  10
```

Für die Generierung der *test*-Methode und des Konstruktors der Constraint-Java-Klassen muss eine Methode implementiert werden, welche ein Stück Java-Code zum Erzeugen des Prädikats zurückgibt. Diese Methode muss dafür sorgen, dass zum Beispiel *String*-Parameter der Prädikate passend für Java-Code *encodet* werden (umschließendes Apostrophe, inneres Apostrophe encodet, Backslashes encodet).

Dynamisches Include

Die parametrisierbaren Prädikate müssen die Methode

```
public boolean isCompatibleTo(
  final Predicate<CT> otherPredicateToCheck
);
```

implementieren.

Durch diese Methode kann der Code-Generator dynamisch informiert werden, dass ein

```
new IntGreater( 0 ) // x  > 0
```

ein

```
new IntGreater( –10 ) // x > –10
```

includiert und zur Vereinfachung der *Switch*-Parameter und zur Bewertung der Gültigkeit einer BOOLSCHEN Belegung einer Constraint-Expression mit herangezogen werden kann.

Dynamisches Exclude

Zur oben genannten Methode kommt für die parametrisierbaren Prädikate noch die zu implementierende Methode

```
public boolean isExcludeTo(
  final Predicate<CT> otherPredicateToCheck );
```

hinzu. Anhand dieser Methode merkt der Code-Generator, dass

```
and(
  new IntLesser ( 0 ) , // x < 0
  new IntGreater( 0 ) ) // x > 0
```

nicht möglich ist. Alle anderen Gimmicks der nicht-dynamischen Excludes funktionieren natürlich auch.

SafeMember

SafeMember dienen dazu, den Code-Generator und damit den Java-Compiler auch für Nebenprobleme der Constraint-Prüfung einzuspannen. Nehmen wir folgendes Szenario an:

```
and(
  RolleBearbeiter ,
  VorgangStatusInBearbeitung ,
  AllOffenenPunktegeklaert
)
```

Der aktuelle Benutzer hat die Rolle Bearbeiter, der aktuell im Programm geladene Vorgang (fachlicher Begriff) ist im Status "In-Bearbeitung" und alle offenen Punkte wurden abgearbeitet/geklärt. Offensichtlich kann der Bearbeiter den Vorgang auf "Erledigt" bzw. "Geschlossen" umschalten. Wenn der Workflow des Vorgangs schön ordentlich als Zustandsautomat oder *State-Pattern* modelliert wurde, also jeder Status eine eigene Java-Klasse ist, muss nun im Programm der aktuelle Status aus dem Vorgangs-Objekt gelesen und zum Status "In-Bearbeitung" gecastet werden. Dies erfolgt im Anwendungscode durch einen Anwendungsprogrammierer und kann schief gehen:

```
final VorgangStatusInBearbeitung vorgangStatusInBearbeitung =
  // kritischer Cast
  (VorgangStatusInBearbeitung) vorgang.getStatus();
final VorgangStatusErledigt vorgangStatusErledigt =
    vorgangStatusInBearbeitung.schliessen();
vorgang.setStatus(
  vorgangStatusErledigt
);
vorgang.save();
```

Durch die Definition einer *SafeMember* am Prädikat *VorgangStatusInBearbeitung* kann man dem Code-Generator den Cast überlassen. Die *SafeMember* wird an einem Prädikat definiert und enthält den Java-Typ, den Member-Namen und ein Stück Java-Code zur Initialisierung der *SafeMember*. Wenn das entsprechende Prädikat bei jeder boolschen Belegung des Constraint erfüllt ist, wird die *SafeMember* in der Constraint-Java-Klasse als *public final Member* generiert:

```
final VorgangStatusInBearbeitung
  vorgangStatusInBearbeitung =
  // den Cast hat der Code-Generator bereits erzeugt
    constraint.vorgangStatusInBearbeitung;
```

Falls irgendwann einmal das Constraint so geändert werden sollte, dass die *SafeMember* nicht mehr generiert wird, ist der Code nicht mehr kompilierbar. Außerdem wird der Anwendungscode knapper und ausdrucksstärker. Falls die Status nicht unterschiedliche Java-Klassen sind, hilft wenigstens noch ein aussagekräftiger Member-Name beim Prüfen. Prinzipiell könnte man auch eine *Safe-Get*-Methode generieren, diese ist aber noch nicht im Code-Generator eingebaut.

SafeOperations

SafeOperations bilden über eine Operation ein Constraint auf ein anderes ab. Hat man beispielsweise ein Constraint "Zahl > 0" und addiert die vermerkte Zahl um 1, dann ist das Ergebnis ein Constraint "Zahl > 1". Es handelt sich um eine mathematische Abbildungsvorschrift, welche statt eines Wertes auf einen anderen Wert, ein Constraint auf ein anderes Constraint abbildet. Im downloadbaren *Zip* befindet sich das Beispiel *StrLenConstraintCodeGenerator*, in welchem für einen constraint-gesicherten String eine sichere *SubString*-Operation generiert wird. Damit sollten *IndexOutOfBoundsExceptions* zur Vergangenheit gehören. Voraussetzung für eine sichere *SubString*-Operation ist die Mindestlänge des ursprünglichen Strings.

Die Definitionsklasse der *SafeOperation* muss die Gültigkeit der mit bestimmten Parametern bestückten *SafeOperation* prüfen und dem Code-Generator über zu implementierende Methoden, den in die Constraint-Klasse einzufügenden Java-Code, übergeben. Es gibt zwei Arten von *SafeOperations*, einmal *ValueSafeOperation*, welche die Welt der aktuell definierten Constraints verlassen und irgendein Objekt zurückgeben und andererseits *DestinationConstraintSafeOperation*, welche wie oben ausgeführt ein Ziel-Constraint besitzen.

Die *DestinationConstraintSafeOperation* sind wiederum in *NewDestinationConstraintSafeOperation* (erzeugen ein neues Ziel-Constraint) und *TransformDestinationConstraintSafeOperation* (transformiert das ursprüngliche Constraint) unterteilt. Die Definitionen der *DestinationConstraintSafeOperation* müssen noch das Ziel-Constraint als Prädikat an den Code-Generator übergeben. Im *SubString*-Beispiel ist das Ziel-Constraint genauso wie das ursprüngliche Constraint auf einen String bezogen.

Für die eigentlich viel interessanteren Übergänge von einem Context-Objekt-Typ zum anderen (etwa Um-

wandlung *String* in *Double* oder anderes fachspezifisches) gibt es die *OtherContextTypeDestinationConstraintSafeOperation*. Prinzipiell gehe ich davon aus, dass es in einem nicht mehr ganz winzigen Projekt, welches meinen Code-Generator verwendet, mehrere Arten von Generatoren mit jeweils unterschiedlichen Typen der Kontext-Objekte geben wird. Diese verschiedenen Code-Generatoren werden in einer *ConstraintCodeGeneratorSuite* zusammengefasst und informieren sich gegenseitig über erforderliche Constraints.

SafeOperations sind im Gegensatz zu den *SafeMembern* an den Constraints definiert. Dies ist sicher noch nicht der Weisheit letzter Schluß. Aber gerade im *SubString*-Beispiel wären

```
( String.length() / 2 ) ^ 2
```

SafeOperations möglich, die, wenn sie alle generiert werden würden, sicher die Größengrenze für eine Java-Klasse von 64 kByte sprengen würden. Sie verwenden natürlich nicht nur *SubString*-Operationen, sondern die für Ihr Projekt jeweils wichtigen höherwertigen fachlichen Operationen in sicherer Weise.

Philosophisches (zu den Safe-Operations)

Die Methode *transformPredicate* in *TransformDestinationConstraintSafeOperation* ist sehr abhängig von anderen Prädikaten/Expressions. Eine Änderung in anderen Prädikaten bzw. Expressions kann dazu führen, dass diese Methode falsch arbeitet und nachgezogen werden muss. Im Grunde wird darin fachliche Logik abgebildet, die ohne den Code-Generator in normalem Code abgebildet worden wäre. Man könnte argumentieren, dass die generierten Constraint-Klassen mehrfach verwendet werden können und dies auf normale Logik (ohne Code-Generator) nicht zuträfe. Das stimmt aber nicht. Die normale Logik könnte auch in Methoden/Klassen gekapselt und mehrfach, sogar parametrisiert, aufgerufen/verwendet werden. Also bleibt als Vorteil für den Code-Generator nur die virale Wirkung der Compiler-Prüfung. Damit ist dies eine Frage der Wirtschaftlichkeit.

Falls der Code aber weiter in Bausteine zerlegt wird, was unweigerlich eine Trennung von Realisierung und Verwendung von Bausteinen zur Folge hat, treten wieder Fernwirkungen auf, für die Compiler-Prüfungen die Sicherheit vor Fehlern erhöhen. Vielleicht gibt es sogar eine emergente Wirkung. Im Bereich der Safe-Operations suche ich noch nach Anwendungs- bzw. Testfällen. Im Moment ist es so, als ob man einen *Parser* für eine Sprache schreibt, für die man keine Grammatik hat. Ich muss zugeben, ich habe sowas schon gemacht. Da ich die gesamten Zusammenhänge der *SafeOperations* noch nicht verstanden habe, wird sich hier wahrscheinlich noch einiges ändern.

Performanz

Bezüglich der Performanz muss die Laufzeit des mit Constraints abgesicherten Programms und die Dauer des Generierungslaufes unterschieden werden. Die Laufzeit des mit Constraints abgesicherten Programms ist mit Sicherheit schlechter als ohne Absicherung. Dabei sollte man aber bedenken, dass die meisten Performanz-Probleme bei Zugriffen auf die Datenbank auftreten und die Java-Laufzeit meist nur eine untergeordnete Rolle spielt. Und was nützt schon ein Programm, das schnell aber nicht korrekt ist.

Der Generierungslauf bricht mit zunehmender Anzahl von Constraints stark in der Laufzeit ein, weil die zu lösenden Probleme einen exponentiell ansteigenden Aufwand haben. Hier werde ich noch einiges optimieren.

Historie und Ausblick

Dieser hier vorgestellte Code-Generator mag ziemlich simpel erscheinen, ist aber das Ergebnis von Überlegungen und Versuchen, die ich in den letzten zwei Jahren angestellt habe. Anregungen und Lösungsansätze dafür haben mir die Veranstaltungen der JUG-Erlangen und der Herbstcampus gegeben, die mir geholfen haben neben dem profanen Ansatz ein wenig die theoretischen Hintergründe zu verstehen.

Die Möglichkeit, den Compiler mit Hilfe von Typen zur Vermeidung von Programmierfehlern einzusetzen, beschäftigt mich andererseits, seit ich Java benutze. Der vorher von mir verwendete *Clipper*-Compiler hat so schwach geprüft, dass jede Programmerweiterung zahlreiche Fehler nach sich gezogen hat. Eine auf jeden Fall geplante Erweiterung ist die Ersetzung der boolean-Rückgabewerte bei den *test*-Methoden der Prädikate durch *BoolStrTupel*, welche neben dem Ergebnis auch einen Fehlertext zur genauen Beschreibung des aufgetretenen Problems liefern.

Fast alle IT-Technologien nutzen oder beruhen auf Bausteinbildung. Bisher ist im Code-Generator davon nichts zu sehen. Damit der Code-Generator eine Zukunft hat, muss es wahrscheinlich mal so etwas wie benannte (Sub-)Expressions mit Parametern oder andere Formen von Bausteinen geben.

Ab *Java8* wird das Checker-Framework [Link] mit seinen Typ-Annotationen in den Java-Standard aufgenommen. Man könnte glauben, dass dieser Code-Generator dann obsolet ist. Da ich aber beim *Checker-*

Framework noch nichts über spezialisierende *Switche*, Kompatibilität, *RangeCheck* und *SafeOperations* gelesen habe, glaube ich dies nicht. Das Checker-Framework könnte aber die Null-Problematik (siehe Abschnitt Null-Problem(o)) dieses Generators absichern.

Ich gehe davon aus, dass sich das Checker-Framework und dieser Generator gut ergänzen werden. Mit dem weiteren Einsatz des Code-Generators werden weitere Schalter und Optionen hinzukommen. Aber wie jemand mal bei einem Vortrag sagte „ein System das alles kann, kann auch nichts“. Also weitere Optionen erhöhen die Flexibilität, machen die Benutzung aber auch schwieriger.

Bisher wird der Java-Code im Generator durch *Concatenation* hart codierter Strings erzeugt. Man könnte über Code-Templates nachdenken, aber hier gilt wie oben gesagt, mehr Flexibilität heißt auch mehr Verantwortung. Weiteres Tooling, zum Beispiel *Eclipse-Plugins*, gibt es bisher auch nicht.

Weiterführende Literatur

- Kuecker, Heiner *ConstraintCodeGenerator*
  www.heinerkuecker.de/ConstraintCodeGenerator.html
- The Jass Page *JASS Java with Assertions*
  http://csd.informatik.uni-oldenburg.de/~jass
- cofoja *Contracts for Java*
  http://code.google.com/p/cofoja
- Object Computing, Inc. *Software Engineering Tech Trends*
  September 2011, Design by Contract in Java with Google
  http://sett.ociweb.com/sett/settSep2011.html
- C4J *Design By Contract for Java*
  http://c4j.sourceforge.net/
- jContractor *Design by Contract for Java*
  http://jcontractor.sourceforge.net/
- OVal *The object validation framework for Java™ 5 or later*
  http://oval.sourceforge.net/userguide.html
- github GContracts *Programming by Contract for Groovy*
  https://github.com/andresteingress/gcontracts
- Google-Suche *AOP-Frameworks für Design by Contract*
- Google-Suche *Design by Contract with JML (Java modeling Language)*
- kindsoftware *Extended Static Checker for Java version 2 (ESC/Java2)*
  http://kindsoftware.com/products/opensource/ESCJava2
- JSR *Type Annotations (JSR 308) and the Checker Framework*
  http://types.cs.washington.edu/jsr308

Kurzbiographie

Heiner Kücker ist freiberuflicher *Java*-Entwickler seit 2000 und fühlt sich einfach sicherer, wenn der *Compiler* ihn vor seinen Flüchtigkeitsfehlern (macht er ständig) schützt. Zur Rationalisierung seiner Arbeit schreibt er sich kleine *main-Methoden*-Tools. Zwischen 2002 und 2005 hat er eine Fluss-Steuersprache für *JSP*-Apps entwickelt. Daneben interessiert er sich für Programmiersprachen und -techniken, wie funktionale Programmierung. (http://heinerkuecker.de/ConstraintCodeGenerator.html, http://heinerkuecker.de/Cnc.html)

Des Programmierers kleine Vergnügen

Bloomige Angelegenheit

von Michael Wiedeking

Bit-Mengen scheinen immer eine gute Idee zu sein, wenn die abzubildende Menge entsprechend überschaubar ist. Wird aber die zu verwaltende Menge zu groß, würde auch die Bit-Menge zu groß werden. Abhilfe schafft hier die Idee des Herrn Bloom, anstatt für jedes Element je ein Bit zu verwenden, die einzelnen Bits geschickt mehrfach zu vergeben.

Für sich genommen ist es ja eigentlich ganz naheliegend, dass man Mengen auf Bit-Sequenzen abbildet, um herauszufinden, ob ein bestimmtes Element in dieser Menge enthalten ist. So konnte man schon im Oktobervergnügen 2010 sehen, dass man Primzahlen sehr kompakt in einem *Bit-Array* speichern kann [1]. Werden allerdings die abzubildenden Mengen oder die zu verwaltenden Elemente zu groß, hat man das Problem, dass auch die assoziierte Bit-Sequenz – trotz aller Platzersparnis – zu viel Speicher beansprucht. Es stellt sich also die Frage, ob man nicht auch eine beliebig große Menge auf eine Bit-Sequenz mit fester Größe abbilden kann.

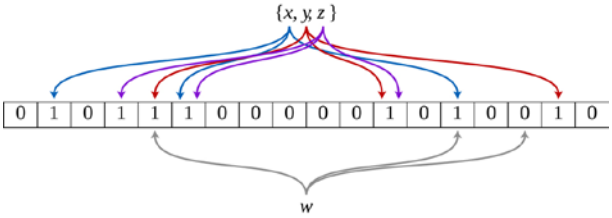
Diese Frage lässt sich nur dann beantworten, wenn man weiß, was man mit einer solchen Abbildung auf eine Bit-Sequenz erreichen möchte. Bei den gespeicherten Primzahlen in [1] ging es darum festzustellen, ob eine konkrete Zahl in einem vorgegebenem Intervall prim ist. Nur wenn das entsprechende Bit gesetzt war, wusste man, dass die die Zahl prim ist. Also musste man zu jeder möglichen Zahl ein eindeutiges Bit haben, das den Zustand (prim oder teilbar) repräsentiert.

Genügt es einem aber, nur zu wissen, dass eine Eigenschaft „nicht“ existiert, dann kann man Bits einfach mehrfach verwenden. Hat man beispielsweise eine Datenbank in der etwa 10 Zahlen von 0 bis 99 abgespeichert werden, dann könnte man diese Information auch in einem Bit-Array mit 100 Bits vorhalten. Möchte man nun aber nur 10 Bit dafür verwenden, so bedarf es einer passenden Abbildungsfunktion h, damit man tatsächlich mit den gegebenen 10 Bit auskommen.

So könnte man für h einfach die Zehnerstelle als Index für das Bit-Array verwenden. h(16) liefert also 1, $h(73)$ entsprechend 7. Enthält die Datenbank nun die Zahlen 38, 66 und 72 ergibt sich eine Bit-Sequenz, in der die Bits 3, 6 und 7 gesetzt sind. Will man nun feststellen, ob die Zahl 44 enthalten ist, schaut man an der Stelle $h(44)$ nach; und weil das Bit 4 dort nicht gesetzt ist, weiß man sicher, dass die Zahl 44 nicht enthalten ist.

Will man prüfen, ob die 37 enthalten ist, stellt man fest, dass das Bit $h(37) = 3$ schon gesetzt ist. Das bedeutet aber leider nur, dass 37 „vielleicht" enthalten ist, da ja bei h nur die Zehnerstelle berücksichtigt wird. Dennoch ist der Fehler bei dieser Verteilung hier „nur“ 30 %, denn in 70 % der Fälle wird durch diesen trivialen Test korrekt ermittelt, dass das Element nicht enthalten ist.

Ein Herr Burton Howard Bloom hat wohl in den Siebziger Jahren herausgefunden, wie man dieses Vorgehen enorm verbessern kann, wenn man mehrere Funktionen h verwendet. Diese *Hash*-Funktionen, die für einen beliebigen Datentyp T eine ganze Zahl berechnen, sollen möglichst verschieden sein, so dass für jedes T möglichst verschiedene Bits betroffen sind.



Damit die Ergebnisse für die Anwendung optimal sind, muss ein ausgewogenes Verhältnis von Speicherplatz (m Bits) und der Anzahl von Hash-Funktionen (k) gefunden werden (in der Grafik ist $m = 18$ und $k = 3$).

Mit Hilfe eines solchen Bloom-Filters kann also sehr effizient entschieden werden, ob ein Element in einer Menge enthalten ist oder nicht. Entscheidend dabei ist, dass nicht mehr der komplette Datensatz zwischengespeichert werden muss, um entscheiden zu können, ob er vorhanden ist oder nicht, wie das etwa eine Hash-Tabelle machen muss.

Will man also beispielsweise wissen, ob für einen Schlüssel ein Eintrag in einer Datenbank existiert, muss man niemals auf die „langsame" Datenbank zugreifen, wenn der Bloom-Filter das Element nicht enthält. Nur wenn die Bits aller Hash-Funktionen gesetzt sind, muss man noch auf die Datenbank zugreifen.

Bei optimalem k braucht man pro Datensatz – unabhängig von seiner Größe – nur 9,6 Bit, wenn man nur noch in 1% der Fälle auf die Datenbank zugreifen möchte. Interessanterweise braucht man nur 4,8 Bit mehr, um diesen „Fehler" um Faktor 10 zu reduzieren [2].

Der Einsatz eines Bloom-Filters kann also enorm Speicher sparen. Ob er immer einer Hash-Tabelle überlegen ist, hängt davon ab, wie groß der Fehler ist. Wie immer geht es also um das korrekte Verhältnis von Speicher und Laufzeit, das hier signifikant von der Qualität der Hash-Funktionen abhängt – was ein guter Grund wäre, sich einmal mit solchen zu beschäftigen.

Aber das ist wie so oft eine ganz andere Geschichte und muss auf eines der nächsten Vergnügen verschoben werden.

Referenzen

[1] Wiedeking, M. *Des Programmierers kleine Vergnügen – Primzahlen im Paket*, KaffeeKlatsch, Jahrgang 3, Nr. 10, S. 12, Bookware, Oktober 2010
http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2010-10.pdf

[2] Wikipedia *Bloom Filter*
http://en.wikipedia.org/wiki/Bloom_filter

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

Deutsch für Informatiker

Am bestensten gemeint und doppeltgemoppelt

von Alexandra Specht

Wir erhielten doch diesen unterhaltsamen Leserbrief von Herrn Förster, den wir Ihnen letzten Oktober abgedruckt haben. Darin ging es um den „bestbewertesten aller vergangenen Herbstcampusse“. Nachdem es bei der Steigerung von Adjektiven und Adverbien und auch bei anderen Stilmitteln, die der Steigerung des Gesagten dienen sollen, bemerkenswerte Tatsachen gibt, heute ein Artikel über ebensolche Steigerungen.

Die Komparation (von lat. comparare „vergleichen“) ist in der Sprachwissenschaft die Steigerung von Adjektiven und Adverbien.

Es werden im Deutschen die folgenden drei Steigerungsformen unterschieden:

- Positiv hart
- Komparativ härter
- Superlativ am härtesten

Im Deutschen enden regelmäßig gesteigerte Adjektive im Komparativ auf *-er* und werden mit *als* zum Vergleichsobjekt verbunden.

Der Superlativ endet mit *-st* bzw. *-est* nach dem Konsonant und wird mit *am* verbunden:

- wild – wilder – (am) wildeste(n)

Unregelmäßige Steigerungen sind Adjektive, die von der oben genannten Regel abweichen (anderer Wortstamm oder Veränderung eines Konsonanten). Unregelmäßig sind zum Beispiel:

- viel – mehr – (am) meiste(n)
- gerne – lieber – (am) liebste(n)
- hoch – höher – (am) höchste(n)

Der Superlativ (lat. superlatio „Übertreibung“) ist der höchste Steigerungsgrad der Eigenschaftswörter.

Beispiel:

Peter hackt von allen Hackern am schnellsten.

Wenn im Satz keine Vergleichswerte genannt werden, handelt es sich jedoch um einen Elativ. Das ist eine Steigerungsform des Adjektivs. Er ist im Deutschen entweder formal mit dem Superlativ identisch oder wird durch eine Vorsilbe oder einen vorangehenden Gradpartikel gebildet.

- *Superlativ*: „Wir entwickeln den funktionalsten Code der Welt.“ (vergleichend)
- *Elativ*: „Wir entwickeln funktionalsten Code.“ (Unabhängig von anderem Code ist der Code funtionalst.)
- *Elativ (Partikel)*: „Wir entwickeln höchst funktionalen Code.“
- *Elativ (Präfix)*: „Wir entwickeln hochfunktionalen Code.“

Der Elativ wird oft von Jugendlichen verwendet:

- *Elativ (Präfix)*: Das ist megageil. Das Kleid ist obercool.
- *Elativ (Partikel)*: Das ist konkret krass. Er ist hammer faul.

Absolutadjektive werden üblicherweise nicht gesteigert, weil es nicht möglich ist, ein bisschen Teilhabe an der Eigenschaft zu haben.

Beispiele:

Schwanger, dreieckig, tot, gleich.
Oder Eigenschaftswörter, die sowieso schon den höchsten/geringsten Grad ausdrücken:

Beispiele:

leer, ganz, absolut, einzig.

Wird von diesen absoluten Adjektiven trotzdem ein Superlativ gebildet, so heißt dieser Hyperlativ.

Beispiele:

optimalste Lösung, die letzteste Alternative, die einzigste Möglichkeit, herzlichste Grüße.

Regelgerecht können sie jedoch gesteigert werden, wenn sie in übertragener oder relativer Bedeutung verwendet werden.

Beispiele:

Peters Vorträge sind lebendiger als Pauls.

Zusammengesetzte Adjektive werden ebenfalls nicht gesteigert, wenn das nominale Erstglied selbst eine Verstärkung ausdrückt, wie bei strohdumm oder eiskalt.

So, nun kommen wir zu den zusammengesetzten Adjektiven, wie bestbewertet, das Herr Förster in seinem Leserbrief erwähnte. Bei zusammengesetzten Adjektiven ist es nur möglich, den ersten oder den zweiten Teil zu steigern, nicht jedoch beide. Es kann sich dabei jedoch ein Bedeutungsunterschied ergeben.

- am höchsten fliegend (Flugzeug)
- hochfliegendste (Pläne)

Dann gibt es noch Superlativadverbien. Das ist ein Verb, welches superlative Form und Bedeutung aufweist.

Beispiele:

bestens, spätestens, mindestens, höchstens.

So, nun wissen wir Bescheid. Wenn wir mal was falsch machen, können wir aber immer noch behaupten, dass wir ein Elativ benutzt haben. (z. B. beim besten Willen, zu unserer vollsten Zufriedenheit, mit freundlichsten Grüßen.) Einzig und allein „in keinster Weise“ sollten wir aus unserem Sprachschatz tilgen, man hört es zwar des Öfteren, es ist aber einfach nur eine Stilblüte, da *kein* ein Indefinitpronomen und kein Adjektiv ist, damit ist es nicht steigerbar. Niemalst und unter keinsten Umständen.

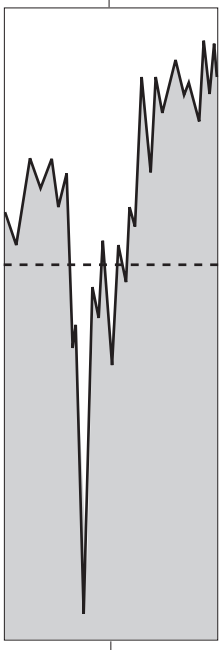
Kaffeesatz

Machtworte

von Michael Wiedeking

Es ist schon unglaublich, was man mit Hilfe des Internets alles erreichen kann. Und dazu bedarf es – wie in jüngster Vergangenheit noch nicht einmal einer großen Gruppe. So schafft es heutzutage schon ein einzelner Mensch, ohne das Haus zu verlassen, beliebig großen Schaden anzurichten.

Ich weiß nicht mehr, wo ich das her habe, aber ich habe einmal eine Geschichte darüber gehört, wie man ein Produkt schlecht machen kann. Dazu steigt man in irgendeiner Metropole mit seinem Kumpel in den Aufzug des höchsten Gebäudes und während der Fahrt erzählt man dann etwa, dass man in den Produkten der schottischen Restaurantkette Rattenknochen gefunden hat. Das wiederholt man dann noch in dem einen oder anderen Gebäude und am Ende des Tages hört man in den Lokalnachrichten der Stadt, dass das Gerücht umgeht, man hätte doch tatsächlich …



Heute funktioniert das etwas anders – und dazu muss man noch nicht einmal sein Haus verlassen. Zu dem Zweck legt man einfach einen Twitter-Account an und publiziert dann seine „Wahrheiten“ einfach dort.

So sorgte diese Vorgehensweise am 29. Januar 2013 dafür, dass der Aktienkurs der Firma Audience binnen Sekunden um 25 % sank [1]. Der Twitterer hatte sich als ein Mitarbeiter des Anlageberaters Muddy Waters ausgegeben und behauptet, die US-Justizbehörde ermittle gegen Audience wegen Betrugs. Glücklicherweise normalisierte sich der Kurs aber wieder, nachdem sich herausstellte, dass es sich um eine Falschmeldung handelte. Dass es sich um eine solche handelte, fand man – abgesehen vom Dementi echter Mitarbeiter von Muddy Waters – dadurch heraus, dass der Twitterer nur 11 Follower hatte.

Es ist nicht ganz klar, welche Ziele eigentlich mit solchen Falschmeldungen überhaupt erreicht werden sollen, sie tauchen aber immer wieder auf. So hat beispielsweise am 6. August 2012 der russische Innenminister Wladimir Kolokolzew mit Berufung auf einen Abgesandten getwittert, dass der syrische Präsident getötet oder verletzt worden sei [2]. Im Nachhinein stellte sich heraus, dass der Twitter-Account gar nicht von dem Innenminister war, was eine entsprechende Meldung über diesen Account bestätigte.

Falls Sie sich also demnächst durch Aktienspekulation ein bisschen Geld dazuverdienen wollen, vergessen sie nicht, sich ausreichend Follower zu kaufen, damit der Betrug nicht gleich auffällt.

Referenzen

[1] Reuters *Hoax tweets send Audience shares a twitter*
http://www.reuters.com/article/2013/01/29/us-audience-shares-idUSBRE90S11T20130129

[2] Reuters *Twitter user sends hoax message on Assad's health*
http://www.reuters.com/article/2012/08/06/syria-crisis-twitter-idUSL6E8J6CSK20120806

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.

Bookware
Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: javausergroupdarmstadt@googlemail.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Torsten Rentsch (torsten@jugsaxony.de)
Herr Falk Hartmann (falk@jugsaxony.de)
Herr Kristian Rink (kristian@jugsaxony.de)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o asc-Dienstleistungs GmbH
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
Universität Karlsruhe, Gebäude 50.34
http://jug-karlsruhe.de
jug-karlsruhe@gmail.com

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Alexandra Specht
(alexandra.specht@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)
Kontakt: Frau Ursula Burri

.NET User Groups

Deutschland

.NET User Group OWL
http://www.gedoplan.de/cms/gedoplan/ak/ms_net
c/o GEDOPLAN GmbH

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(dodned@googlemail.com)

.NET Usergroup Frankfurt
c/o Thomas Sohnrey, Agile IService
http://www.dotnet-ug-frankfurt.de
Kontakt: Herr Thomas 'Teddy' Sohnrey
(thomas.sohnrey@gmx.de)

.NET DGH
.NET Developers Group Hannover
http://www.dotnet-hannover.de
Kontakt: Herr Friedhelm Drecktrah
(friedhelm@drecktrah.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.NET User Group Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy.erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET User Group Paderborn
c/o Net at Work Netzwerksysteme GmbH,
http://www.dotnet-paderborn.de
(raacke@dotnet-paderborn.de)
Kontakt: Herr Mathias Raacke

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Frau Catrin Busley

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET Usergroup Rheintal
c/o Computer Studio Kogoj
http://usergroups.at/blogs/dotnetusergrouprheintal/default.aspx
Kontakt: Herr Thomas Kogoj (thomas@kogoj.com)

.NET User Group Austria
c/o Global Knowledge Network GmbH,
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Software Craftsmanship Communities

Deutschland

Softwerkskammer – Mehrere regionale Gruppen unter einem Dach, http://www.softwerkskammer.de

MATHEMA

meet the experts of enterprise infrastructure

Objektorientierte Analyse und Design mit UML und Design Patterns

Methoden und Prinzipen für die Entwicklung von OO-Modellen und die Dokumentation mit der UML
2. – 4. April 2013, 22. – 24. Juli 2013,
1.180,– € (zzgl. 19 % MwSt.)

Neues in Java 7

Die nächste Java Generation
4. – 5. März 2013, 2. – 3. Mai 2013,
835,– € (zzgl. 19 % MwSt.)

iPhone Programmierung

Mobile Anwendungen für das Apple iPhone
10. – 12. April 2013, 17. – 20. Juni 2013,
1.180,– € (zzgl. 19 % MwSt.)

Programmierung mit Java

Einführung in die Java-Technologie und die Programmiersprache Java
13. – 17. Mai 2013, 26. – 30. August 2013,
1.645,– € (zzgl. 19 % MwSt.)

Testkonzepte

Software-Tests zur kontinuierlichen Sicherung der Qualität
6. – 8. Mai 2013,
14. – 16. August 2013,
1.315,– € (zzgl. 19 % MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

Software-Entwickler (m/w)
Software-Architekt (m/w)

Arbeiten Sie gerne selbstständig, motiviert und im Team? Haben Sie gesunden Ehrgeiz und Lust, Verantwortung zu übernehmen?

Wir bieten Ihnen erstklassigen Wissensaustausch, ein tolles Umfeld, spannende Projekte in den unterschiedlichsten Branchen und Bereichen sowie herausfordernde Produktentwicklung.

Wenn Sie ihr Know-how gerne auch als Trainer oder Coach weitergeben möchten, Sie über Berufserfahrung mit verteilten Systemen verfügen und Ihnen Komponenten- und Objektorientierung im .Net- oder JEE-Umfeld vertraut sind, dann lernen Sie uns doch kennen.

Wir freuen uns auf Ihre Bewerbung!

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

Das Allerletzte

Dies ist kein Scherz!
Diese Meldung wurde tatsächlich in der freien
Wildbahn angetroffen.

Ist Ihnen auch schon einmal ein Exemplar dieser
Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im März.

Herbstcampus

Wissenstransfer par excellence

2.– 5. September 2013

in Nürnberg