Ausgabe 1/2011 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

JMX4Perl
Perl und Java, zwei ungleiche Schwestern

WebSockets
... mit WebSockets und Perl in HTML5 eintauchen

Perl und SVG
Scalable Vector Graphics

Nr 17

Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de

VORWORT

Neues Jahr, neues Glück...

Wir gehen in das 5. Jahr von $foo - mit etwas anderen Bedingungen. Die Smart-Websolutions Windolph und Bäcker GbR wurde aufgelöst und ich mache als Einzelunternehmen weiter. $foo ist damit weiterhin gesichert, dem Lesevergnügen steht nichts im Wege.

Damit sind auch einige Veränderungen hinter den Kulissen verbunden. Neuer Server (leistungsfähiger) und für die Kommunikation steht jetzt ein OTRS bereit, in dem alle Bestellungen und alle Feedbackmails landen. Damit wollen wir sicherstellen, dass keine Frage und kein Feedback untergeht. Uns sind Wünsche und Anregungen sehr wichtig, denn sie helfen uns besser zu werden.

Außerdem sind wir gerade in der Testphase von $foo in einem weiteren Format: epub. Für Abonnenten wird es das Format automatisch dazu geben. Ich hoffe, wir können noch vor der Mai-Ausgabe anfangen, die einzelnen Ausgaben online zu stellen.

Natürlich wird das alles mit Perl erstellt.

Aber auch bei Perl ändert sich in diesem Jahr etwas: Es wird ein neues stabiles Release von Perl geben - Perl 5.14. In dieser Ausgabe finden Sie einen Artikel, der einen Teil der Neuerungen beschreibt. Es gibt viele gute und spannende Änderungen.

Seit dem 01.01.2011 ist unter *http://planet.perl-magazin.de* ein Blog-Planet mit deutschsprachigen Perl-Blogs zu finden. Wenn es noch solche Blogs gibt, die in die Liste mit aufgenommen werden soll, dann schreiben Sie uns doch bitte an feedback@perl-magazin.de

Viele Grüße,
Renée Bäcker

Die Codebeispiele können mit dem Code

34lepys

von der Webseite www.foo-magazin.de heruntergeladen werden!

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe:
http://del.icio.us/foo_magazin/issue17

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker, Katrin Bäcker
Anzeigen:	Katrin Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

ALLGEMEINES

6 Über die Autoren
8 JMX4Perl
17 Perl und SVG
24 OTRS-Community-Meetings
51 Rezension - "Effective Perl Programming"
54 Rezension - Bücher zur IT-Geschichte

WEB

26 WebSockets

PERL

35 Was ist neu in Perl 5.14?

MODULE

39 WxPerl Tutorial - Teil 6
46 Moose Tutorial - Teil 3

TIPPS & TRICKS

55 HowTo

NEWS

57 Neues von TPF
59 CPAN News
61 Termine

62 LINKS

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.
Websites:

- http://www.thomas-fahle.de
- http://Perl-Suchmaschine.de
- http://thomas-fahle.blogspot.com
- http://Perl-HowTo.de

Alexander Halle

Alexander Halle arbeitet im IT-Servicedesk der radprax Gesellschaft für Medizinische Versorgungszentren mbH in Wuppertal. Dort betreut er unter anderem das OTRS-System, wodurch sein Interesse für Perl endgültig geweckt wurde. Aktuell beschäftigt er sich mit dem Aufbau einer geeigneten CMDB für den Support, sowie der Mitarbeit im OTRS Community Board.

Dr. Roland Huß

Roland hat schon das rosa Kamel geritten, das immer noch einen Ehrenplatz in seiner Bibliothek einnimmt. In der letzten Dekade hat er hauptsächlich als Java Zauberer gewirkt, der riesige XML Dokumente in riesige Stacktraces verwandeln kann. Er leitet die Abteilung "Research & Development" bei dem Münchner IT-Beratungs und Softwarehaus ConSol* und ist insbesondere in der Sparte "Open Source Monitoring" aktiv, bei der er den Bereich JEE Monitoring betreut und sich auf knifflige Herausforderungen freut (ja, man kann uns kaufen). Dabei gelingt ihm auch immer wieder der Spagat zwischen High-End JEE Entwicklungen und seiner Muttersprache Perl. Privat ist er oft leidender Fan des 1. FCN und passionierter Chili-Liebhaber mit mehr oder weniger erfolgreichem eigenem Anbau.

Thomas Kappler

Thomas ist Informatiker und programmiert am Schweizerischen Institut für Bioinformatik in Genf. Wenn er in seiner Freizeit nicht gerade in den Bergen herumkraxelt, erkundet er gern Sprachen wie Erlang und Lisp. Aber Perl bleibt halt Perl - nicht immer hübsch, aber eine endlose Fundgrube für Praktisches und Esoterisches.

Viacheslav Tykhanovskyi

Viacheslav Tykhanovskyi programmiert seit 2005 in Perl. Die meiste Software, die er geschrieben hat, ist Open Source und ist unter http://github.com/vti zu finden. Heute arbeitet er als Freelancer. In seiner Freizeit betreibt er einen Blog über Perl: http://showmetheco.de

Dr. Roland Huß

JMX4Perl

Perl und Java, zwei ungleiche Schwestern

Es war Anfang 2009 als mein Kollege Gerhard, ein ausgewiesener Nagios-Guru, bei mir anklopfte und mehr oder weniger verzweifelt nach einer Nagios-Lösung zur Überwachung von Java-Applikationsservern suchte.

Das Problem mit den bisherigen Lösungen war, dass zwar Java von Hause aus die exzellenten Monitoring Schnittstelle JMX mitbringt, diese aber von außerhalb nur über Java selbst nutzbar ist. Da Nagios-Checks kurzlaufende, externe Prozesse sind, ist der Wasserkopf, den der Start einer Java Virtual Machine (JVM) mit sich bringt insbesondere für große Nagios Installationen mit tausenden zu überwachenden Servern nicht verkraftbar. Gerhard suchte nach einer einfacheren Lösung, die auch von Skriptsprachen wie Perl verwendbar ist. Das war die Geburtsstunde von **jmx4perl**. Angefangen hat es mit einem kleinen Java-Servlet (23 Kilobyte), das serverseitig JMX über die Java-API anspricht und auf der anderen Seite diese über HTTP exportiert, wobei die Nutzdaten in einem JSON Format übertragen werden. Dafür bietet Perl Unterstützung dank der CPAN-Bibliotheken `LWP` und `JSON`. Dieser Agent ist über die Jahre auf 150 Kilobyte gewachsen und inzwischen auch nicht nur für Perlianer interessant.

Bevor jmx4perl und seine Perl Module anhand von Beispielen ausführlich vorgestellt werden, noch ein paar Worte zu JMX, woher es kommt und was es kann.

JMX

JMX (Java Management Extensions) definiert die Architektur, die API und die Dienste zur Verwaltung und Überwachung vom Java Anwendungen. Dabei ist JMX schon eine sehr alte Spezifikation. In Java werden Standards in den *Java Specification Requests* (JSR) festgehalten, die im Rahmen des *Java Community Processes* (JCP) erstellt werden. JMX ist definiert in JSR-3 und hatte seine Geburtsstunde bereits im Jahre 1999. Im Gegensatz zu vielen anderen Java Spezifikationen (wie z.B. EJB oder JSF) war sie von Beginn an wohl durchdacht und wird in nahezu unveränderter Form bis heute ausgiebig genutzt. Seit Java 1.5 ist eine JMX Implementierung Bestandteil der Java Laufzeitumgebug, so dass sie heute jeder zeitgemäßen Java Anwendung ohne zusätzlichen Installationsaufwand zur Verfügung steht.

Mithilfe von JMX haben Java Programme die Möglichkeit einen Teil ihrer Funktionalität für Managementzwecke zur Verfügung zu stellen. "Management" heißt hier, dass je nachdem welche JMX Schnittstellen eine Applikation zur Verfügung stellt, beispielsweise Komponenten gestoppt und neugestartet, Teile der Anwendung entfernt oder hinzugefügt und Konfigurationen zur Laufzeit verändert werden können. Darüber hinaus können JMX Komponenten Attribute zu Überwachungszwecken exportieren.

In der JMX Fachsprache heißen diese Komponenten **MBeans**. Diese werden innerhalb der Java Virtual Machine lokal bei einem **MBeanServer** registriert, der wiederum den Zugriff auf diese MBeans regelt. Jede MBean hat innerhalb eines MBeanServers eine eindeutigen **ObjectName**, über den sie angesprochen werden kann. Dieser hat das Format `"domain:key=value,key=value,..."`. Dabei legt die *domain* einen Namensraum fest, unter dem sich mehrere MBeans registrieren können. Die Liste von Key-Value Paaren zurrt dann den Namen innerhalb dieses Namensraumes fest. Die Wahl des `ObjectName`s bleibt der Java Anwendung überlassen, es gibt aber eine gewisse Namenskonvention, auf die wir später noch stoßen werden.

Ein Nutzer kann mit MBeans über drei Arten kommunizieren:

- Lesen und Schreiben von **Attributen**
- Ausführen von **Operationen** mit Argumenten
- Registrierung für **Notifikationen** (Events)

Seit Java 1.5 sind in der JVM sogenannte **MXBeans** vom Start weg registriert. Das sind MBeans mit festgelegtem `ObjectName`, die Zugriff auf verschiedene Aspekte der JVM erlauben, u.a. die Abfrage des Heap-Speichers, Anzahl der Threads, Umgebungsvariablen oder auch z.B. die Ausführung einer Garbage Collection.

Auch viele Applikationsserver wie z.B. Tomcat, JBoss, Weblogic oder Websphere kommen bereits mit einer breiten Palette von weiteren vordefinierten MBeans die sich sofort nutzen lassen. Damit lässt sich ein Vielzahl interessanter Messgrößen abfragen wie beispielsweise die Anzahl von Requests pro Minute einer Webanwendung oder die Größe von Datenbankverbindungs-Pools.

Bislang befinden wir uns noch innerhalb der JVM. Wie kommt man nun von außerhalb an diese Informationen ? Dazu hat Sun ebenfalls einen Java-Standard definiert, die sog. *JSR-160 Konnektoren*. Unglücklicherweise liegt der Fokus dieser Spezifikationen auf Java-Clients und nutzt das im letzten Jahrtausend recht populäre Programmiermodell der *Remote Method Invocation* (RMI), bei der der Client über die nahezu identischen Java-Interfaces auf den MBeanServer und die MBeans zugreift. Die Idee dahinter ist eine *transparente Remoteness*, so dass der Nutzer sich keine Gedanken machen muss, ob er lokal oder entfernt auf MBeans zugreift. Im Laufe der Jahre hat sich gezeigt, dass dieser Ansatz im realen Einsatz viele Tücken hat, so dass er aktuell nicht mehr State-of-the-Art ist. Leider ist hier JMX zurückgeblieben, die einzige standardisierte Möglichkeit, entfernt auf JMX zuzugreifen bleiben die RMI basierten JSR-160 Konnektoren. Die Spezifikation JSR-262, JMX auch über WebServices (WS-Management) zu exportieren ist im Jahre 2008 stecken geblieben.

Für uns Perl-Anwender bedeutet das, dass wir für den JMX Zugriff mit JSR-160 immer eine Java Virtual Machine starten müssen. Dies passiert entweder über einen externen Prozess, bei dem ein (selbst geschriebenes) Java Programm gestartet wird, das die Ergebnisse z.B. auf die Standardausgabe schreibt und unser Perl-Programm diese verarbeitet. Oder aber es wird eine der Perl-Java Integrationsmöglichkeiten wie `Inline::Java` eingesetzt. Egal wie, diese Ansätze sind sehr schwerfällig und verlangen zudem einiges an Java Kenntnissen.

An dieser Stelle springt nun **jmx4perl** in die Bresche.

Jmx4perl als Mittler zweier Welten

Jmx4perl ist eine Lösung für den entfernten JMX-Zugriff jenseits der JSR-160 Konnektoren. Dazu muss ein Agent auf dem Zielserver oder einem Proxy-Server installiert werden, der dann entfernte HTTP Anfragen in lokale JMX Aufruf übersetzt und die Ergebnisse über HTTP zurückliefert. Die Nutzlast der Anfragen und Resultate werden in einem JSON-Format repräsentiert. Dieser Ansatz hat mehrere Vorteile. Zum einen ist die Kombination HTTP/JSON ein wohl etabliertes Gespann, das auch ausserhalb der Java Welt eine breite Unterstützung besitzt. Damit ist es Clients, die in anderen Programmiersprachen entwickelt sind, ohne großen Aufwand möglich auf JMX Informationen zuzugreifen. Dadurch, dass die Anforderung einer lokalen Java Installation wegfällt, ist diese Lösung auch wesentlich schlanker. Andererseits ist es dem Agenten aber auch möglich, Funktionen anzubieten, die in JSR-160 fehlen. Der Jmx4perl Agent bietet beispielsweise zusätzlich Bulk-Requests, ein verfeinertes Sicherheitsmodell und noch einiges mehr.

Ein weiterer großer Unterschied zwischen der Kommunikation über HTTP/JSON im Vergleich zu JSR-160, bei dem im wesentlichen serialisierte Java-Objekte übertragen werden, ist, dass sie typlos ist. Das hat Vor- und Nachteile. Der Vorteil ist, dass auf der Client-Seite keine Typinformationen vorliegen müssen, was den Installationsaufwand bei komplexen MBeans weiter reduziert. Der Nachteil ist, dass für die Deserialisierung der Anwender selber wissen muss, mit welchem Datentypen er zu tun hat und diesbezüglich keine Unterstützung von der Laufzeitumgebung erhält.

Für jmx4perl stehen verschiedene Agenten zur Verfügung:

WAR Agent

Dieser Agent unterstützt alle Java Applikationsserver, die einen sogenannten Servlet-Container mitbringen. Dazu gehören ausgewachsene JEE Server wie JBoss, Weblogic oder Websphere aber auch reine Servlet-Container wie

Tomcat oder Jetty. Die Installation funktioniert wie bei jeder anderen Java-Webanwendung auch. Es wird ein *WAR* (Web-ARchive) installiert, das dann von außen über eine bestimmte URL erreichbar ist. Das Deployment funktioniert typischerweise einfach über das Kopieren der Datei in ein bestimmtes Verzeichnis, über Kommandozeilen-Tools oder über eine Weboberfläche.

OSGi Agent
OSGi ist eine Java-Servertechnologie, die einen starken Fokus auf Modularität legt. Dieser Agent wird in der Form eines *OSGi-Bundles* installiert und unterstützt alle gängigen OSGi-Plattformen (Felix, Equinox, Vertigo).

Mule Agent
Mule ist ein weitverbreiteter Open-Source Enterprise Service Bus (ESB), der eine eigene Schnittstelle für JMX Agenten anbietet. Diese nutzt dieser jmx4perl-Agent, um sich nahtlos in den Mule ESB zu integrieren.

JVM Agent
Schließlich existiert ein sehr generischer Agent, der die *JVM Agentenschnittstelle* nutzt um seine Dienste anzubieten. Diese Schnittstelle existiert seit Java 5 und wird hauptsächlich von Java Entwicklungstools wie Profilern genutzt, um in den Classloading Prozess einzugreifen. Zusätzlich nutzt dieser Agent den in Sun/Oracle JVM ab Version 6 integrierten HTTP-Server. Mit diesem Agenten ist es möglich jeder beliebige Java 6 Applikation zu instrumentieren, z.B. auch Desktop Applikationen wie Eclipse oder Cloud Anwendungen wie Hadoop.

Alle Java-Agenten sind Bestandteil von jmx4perl bis Version 0.74. Diese wurden im Oktober 2010 in ein neues Projekt namens **Jolokia** ausgelagert. Ab der Version 0.80 greift jmx4perl somit auf diese Agenten zurück.

Der andere Bestandteil von jmx4perl ist die Perl-Schnittstelle. Diese kann auf verschiedenen Ebenen genutzt werden. Auf der obersten Ebene existieren mitgelieferten Kommandozeilenprogramme, die einen komfortablen Zugriff auf die Agenten ermöglichen. `jmx4perl` erhält die Requestparameter über Kommadozeilen-Optionen, während `j4psh` einen interaktiven Zugriff erlaubt. `check_jmx4perl` ist ein mächtiges Nagios Plugin, das eine Vielzahl von Konfigurationsmöglichkeiten kennt.

Diese Tools nutzen die Perl-Module unterhalb von `JMX::Jmx4Perl`, die einen programmatischen Zugriff auf den Agenten realisieren. Diese Module werden im Folgenden anhand von Beispielen ausführlich vorgestellt.

Schließlich ist es aber auch möglich direkt über HTTP/JSON Bibliotheken mit dem Agenten zu sprechen, wobei diese Möglichkeiten natürlich nicht auf Perl beschränkt ist.

JMX::Jmx4Perl

Zum Ausprobieren der folgenden Beispiele brauchen wir zunächst einen Java Applikationsserver. Dies kann z.B. ein einfacher Tomcat sein, auf den der jmx4perl Agent deployed wird:

- Tomcat 7 herunterladen und installieren - siehe Listing 1.
- Download des `j4p.war` Agenten - siehe Listing 2.
- Starten des Tomcat:

```
$ cd ..
$ bin/catalina.sh start
```

- Überprüfung des Agenten im Browser - siehe Listing 3.

Die Perl-Module werden entweder über `cpan JMX::Jmx4Perl` automatisch installiert bzw. manuell aus dem Archiv *http://search.cpan.org/~roland/jmx4perl*. Für die JMX Shell `j4psh` und das Nagios-Plugin `check_jmx4perl` sollten die optionalen Module ebenfalls installiert werden.

```
$ wget http://www.apache.org/dist/tomcat/tomcat-6/v6.0.29/bin/apache-tomcat-6.0.29.tar.gz
$ tar zxvf apache-tomcat-6.0.29.tar.gz
$ cd apache-tomcat-6.0.29
```
Listing 1

```
$ cd webapps
$ wget http://labs.consol.de/maven/repository/org/jmx4perl/j4p-war/0.74.0/j4p-war-0.74.0.war
$ mv j4p-war-0.74.0.war j4p.war
```
Listing 2

```
$ wget -q -O - http://localhost:8080/j4p/version
{"timestamp":1290421423,"status":200,"request":{"type":"version"},
   "value":{"protocol":"4.0","agent":"0.74"}}
```
Listing 3

```
use JMX::Jmx4Perl;
my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
print "Heap-Memory used: ",
   $j4p->get_attribute("java.lang:type=Memory","HeapMemoryUsage","used")," Bytes\n";
```

Listing 4

```
# Zähle die alle Anfragen an alle Servlets zusammen:
 use JMX::Jmx4Perl;

 my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
 my $val_hash = $j4p->get_attribute("*:j2eeType=Servlet,*",undef);
 map { $total += $_ } map { $_->{requestCount} } values %$val_hash;
 print "Gesamtzahl der Servlet-Requests: ",$total,"\n";
```

Listing 5

Im ersten Beispiel sehen wir uns die Hauptspeicherauslastung mit einer einfacher JMX `READ` Operation an - siehe Listing 4.

Zunächst wird der jmx4perl Client `$j4p` erzeugt. Er benötigt als einziges Pflichtargument die URL zu dem jmx4perl Agenten (`http://localhost:8080/j4p`). Mit `$j4p->get_attribute()` kann nun ein Attribut ausgelesen werden. Das erste Argument ist der MBean-Name (`java.lang:type=Memory`), das Zweite bezeichnet das Attribut selbst (`HeapMemoryUsage`) und das dritte, optionale Argument spezifiziert einen sogenannten *Pfad*. Viele JMX Attribute sind komplexere Datentypen als Strings oder Zahlen. Auf der Seite des Agenten werden diese in JSON-Maps umgesetzt, wobei die Schlüssel die Namen der *inneren Attribute* dieser zusammengesetzten Datentypen sind. Java Collections werden in Arrays umgesetzt. In diesem Fall liefert die MBean `java.lang:type=Memory` für das Attribut `HeapMemoryUsage` ein Objekt des Java Typs `CompositeData` zurück. Dieses setzt der Client in einen Hash der folgenden Struktur um:

```
{
  committed => 85000192,
  init => 0,
  max => 129957888,
  used => 24647976
}
```

Mit einem Pfad kann nun schon auf der Serverseite auf einen Teil der gesamten Datenstruktur zugegriffen werden. Der Pfad `used` in unserem Beispiel führt dazu, dass nur der Wert dieses Schlüssels zurückgeliefert wird. Ein Pfad kann auch noch tiefer in die Datenstruktur zeigen. Zum Beispiel liefert der Pfad `loader/classloader/resources/uRLs/2/url` eine URL zurück, die tief in einem komplexen Objekt versteckt ist. `2` an der vorletzten Stelle wird genutzt, um das dritte Element eines Arrays in dieser Ebene auszuwählen. Ohne Pfad wird die gesamte Datenstruktur als Kombination von Hashes, Arrays und skalaren Werten zurückgegeben.
Eine interessante Variante des obigen Beispiels ist die Nutzung von Wildcards im MBeans oder die Abfrage von mehr als einen Attributnamen - siehe Listing 5.

Enthält ein MBean-Name Wildcards, wie sie in der JMX Spezifikation *http://download.oracle.com/javase/6/docs/api/javax/management/ObjectName.html* festgelegt sind, dann ist der Rückgabewert von `get_attribute` ein Hash, der die konkreten MBean-Namen als Schlüssel hat. Die Werte sind wiederum Hashes, die einen oder mehrere Attributnamen als Schlüssel und die Attributwerte als Werte besitzen.

Ein Attributname `undef` bezeichnet *alle* vorhandenen Attribute der MBean. Ist der Attributname ein einzelner String, dann wird dieses Attribut abgefragt. Ist es ein Arrayref von Strings, dann wird der Wert all dieser Attribute geholt. Wie man sieht, ist es sehr feingranular möglich, mehrere Attribute von mehreren MBeans in einem Rutsch zu lesen. Eine noch flexiblere Möglichkeit bieten *Bulk Requests*, die weiter unten vorgestellt werden (siehe Listing 6.)

Zu beachten ist noch, wenn der MBean-Name *kein* MBean-Muster ist, dann fällt die erste Ebene mit dem MBean Namen weg, da dieser ja schon beim Request festgelegt ist.

Analog zum Lesen von Attributen gibt es u.a. noch folgende Operationen die vom `JMX::Jmx4Perl` Client zur Verfügung gestellt werden:

set_attribute($mbean,$attribute,$value,$path)
Damit können Attributwerte gesetzt werden. Die Argumente bezeichnen den MBean- und Attributnamen, den zu setzenden Wert und wiederum einen optionalen Pfad. Dabei bezeichnet der $path Pfad zu dem inneren Attribut dem der Wert zugewiesen werden soll.

```
use JMX::Jmx4Perl;
use Data::Dumper;

my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
print Dumper($j4p->get_attribute("java.lang:*",["Verbose","Uptime"]));

$VAR1 = {
        'java.lang:type=Memory'       => { 'Verbose' => 'false' },
        'java.lang:type=ClassLoading' => { 'Verbose' => 'false' },
        'java.lang:type=Runtime'      => { 'Uptime' => '1460852'};
        }
```

Listing 6

```
use JMX::Jmx4Perl;
use JMX::Jmx4Perl::Request;
use Data::Dumper;

my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
my $response =
   $j4p->request(new JMX::Jmx4Perl::Request(READ,"java.lang:type=Threading","ThreadCount"));
print Dumper($response);
```

Listing 7

execute($mbean,$operation,@arguments)
JMX Operationen werden mit `execute` ausgeführt. Neben den MBean- und Operationsnamen können ein oder mehrere Argumente übergeben werden, je nachdem welche Argumente die Operation benötigt.

search($mbean_pattern)
Bei `search` können MBean Namen mit einem Muster gesucht werden. Dass Muster kann die Wildcards `*` und `?` enthalten, jedoch sind diese nur an bestimmten Stellen erlaubt.

info()
Mit `info` enthält man eine Kurzbeschreibung der Serverumgebung als String.

list()
`list` liefert Meta-Informationen über alle verfügbaren MBeans. Der Rückgabewert ist ein Hash, der in der Dokumentation vom `JMX::Jmx4Perl` ausführlich beschrieben ist. `list` wird z.B. von den Tools `jmx4perl` und `j4psh` genutzt um das Stöbern in der Liste von vorhandenen MBeans zu erleichtern, um "interessante" MBeans zu finden.

Eine Stufe tiefer ...

Diese High-Level Methoden bieten einen guten Einstieg und sind für viele Anwendungsfälle ausreichend. Um `JMX::Jmx4Perl` voll auszureizen, müssen wir eine Stufe tiefer gehen. Jmx4Perl kommuniziert mit dem Agenten über HTTP GET oder POST Request, und erhält eine HTTP Antwort zurück. Diese werden über die Perl-Module `JMX::Jmx4Perl::Request` und `JMX::Jmx4Perl::Response` gekapselt - siehe Listing 7.

Ein `JMX::Jmx4Perl::Request` hat einen Typ (`READ`) und typspezifische Parameter. Der Konstruktor kann Argumente in verschiedenen Formaten auswerten, mehr dazu in der Dokumentation zu `JMX::Jmx4Perl::Request`. Die Methode `$j4p->request()` nimmt nun solch ein Request-Objekt, wandelt es in einen HTTP Request um, kontaktiert den Agenten und packt die HTTP Antwort in eine `JMX::Jmx4Perl::Response` ein. Die `JMX::Jmx4Perl::Response` beinhaltet neben dem Ergebnis auch weitere Informationen, wie den Request selbst, einen Timestamp und den Ergebnis-Status (Listing 8).

Die `status` Codes sind an HTTP Ergebniscodes orientiert: 200 wird bei einer erfolgreichen Operation zurückgegeben, alles andere sind Error-Codes (z.B. 404 wenn die MBean oder ein Attribut nicht gefunden werden kann).

Neben einzelnen Requests kann die `request` Methode des Clients auch eine Liste von Anfragen verarbeiten. In diesem Fall wird auch ein Liste von `JMX::Jmx4Perl::Response` Objekten zurückgegeben. Hier können auch Anfragen mit verschiedenen Typen gemischt werden (Listing 9).

Das letzte Beispiel (Listing 10) zeigt eine kleine Anwendung, die bei der Fehlersuche einer Java Anwendung hilfreich sein kann. Wenn seltsame Dinge in einer Java Applikation passie-

ren, dann fragen die Java Entwickler meist als erstes nach einem Thread-Dump um die Laufzeitsituation analysieren zu können. Dieser Thread-Dump sollte zeitnah zum Problem geholt werden, auch sind mehrere Thread-Dumps kurz hintereinander sinnvoll um die Dynamik der Applikation zu verstehen. Dieses Beispiel überprüft periodisch, ob es Probleme gibt. In diesem Fall wird ein Problem erkannt, wenn der Hauptspeicher zu mehr als 90% ausgelastet ist oder wenn Threads gefunden werden, die sich gegenseitig in den Schwanz beißen (deadlocks). Ist dies der Fall wird via JMX ein Thread Dump erzeugt und als Perl-Hash in einer Datei abgelegt, die post-mortem analysiert werden kann. Zu beachten ist, dass das Skript nur mit Java 6 aufwärts funktioniert, da erst ab dieser Version die entsprechenden JMX Methoden zur Verfügung stehen. Optimiert werden kann hier noch die initiale Abfrage der KO-Kriterien, die dank Bulk-Request auf einzige HTTP-Abfrage reduziert werden kann. Auch kann man sich zusätzliche oder alternative Kriterien überlegen, wie beispielsweise die Anzahl der Sessions einer Webapplikation oder die Anzahl der offenen Datenbankverbindungen.

Tools, Tools, Tools

Alles schön und gut, aber was sind denn nun die "interessanten" MBeans, die so Java-Server typischerweise im Bauch haben ? Unglücklicherweise kocht hier jeder Hersteller sein etwas anderes Süppchen. Der Standard JSR-77, der sich um eine einheitliche Namensgebung der MBeans kümmert, hat es leider nie richtig zum Durchbruch gebracht und wird jetzt auch für zukünftige Versionen eingestampft. Zudem lässt es JSR-77 auch jedem Server freigestellt, die interessantesten MBeans, die sog. *Statistics Provider* zu implementieren. Die wenigsten tun dies jedoch und wenn dann auch nur für Teilaspekte wie Servletstatistiken. Daher

```
bless({
       'request' => bless( {
                            'attribute' => 'ThreadCount',
                            'mbean' => 'java.lang:type=Threading',
                            'path' => undef,
       'status' => 200,
       'value' => '14'
      }, 'JMX::Jmx4Perl::Response')
```

Listing 8

```
use JMX::Jmx4Perl;
use JMX::Jmx4Perl::Request;
use Data::Dumper;

my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
my @requests = (
                 new JMX::Jmx4Perl::Request(READ,"java.lang:type=Threading","ThreadCount"),
                 new JMX::Jmx4Perl::Request(READ,"java.lang:type=Runtime","Uptime"),
                 new JMX::Jmx4Perl::Request(AGENT_VERSION)
                );
my @responses = $j4p->request(@requests);
print Dumper(\@responses);
```

Listing 9

```
use JMX::Jmx4Perl;
use Data::Dumper;

my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");

while (1) {
    my $memory =
      $j4p->get_attribute("java.lang:type=Memory","HeapMemoryUsage");
    my $dead_locked =
      $j4p->execute("java.lang:type=Threading","findDeadlockedThreads");
    if ($memory->{used} / $memory->{max} > 0.9 || @$dead_locked) {
        my $thread_dump = $j4p->execute("java.lang:type=Threading",
                                        "dumpAllThreads",1,1);
        open(F,">/tmp/thread_dump_".time);
        print F Dumper($thread_dump);
        close F;
    }
    sleep(5 * 60);
}
```

Listing 10

bleibt einem nichts anderes übrig als selbst auf die Suche zu gehen. Jmx4perl bietet hierfür verschiedene Möglichkeiten.

Zunächst einmal kann man sich mit `jmx4perl list` die Namen aller MBeans inklusive ihrer Attribute und Operationen ausgeben lassen - siehe Listing 11.

Mit `jmx4perl attributes` können zusätzlich die Werte aller Attribute abgefragt werden. Aber Vorsicht, dabei können durchaus Datenmengen von 200 MB und größer entstehen.

Komfortabler durchwandert man den JMX Namensraum mit `j4psh` - siehe Listing 12. Das ist eine interaktive, readline-basierte Shell mit Syntax-Highlighting und kontextsensitiver Vervollständigung.

Auch lohnt sich ein Blick in die vordefinierten Nagios-Checks, die sich im jmx4perl Build-Verzeichnis unter `config/` befinden. Für Tomcat zum Beispiel, existieren aktuell 17 vordefinierte Checks in `config/tomcat.cfg`, die Attribute wie die Anzahl der Session und Requests pro Sekunde einer Webapplikation oder auch die Größe der Datenbank Verbindungspools ansprechen (Listing 13).

Der Syntax ist ausführlich in der Manpage zu `check_jmx4perl` beschrieben, das Beispiel hier greift auf MBeans mit dem Muster `*:type=GlobalRequestProcessor,name=$0/requestCount` zu, wobei `$0` durch den Namen des Tomcat Konnektors ersetzt werden muss (z.B. `http-8080`).

Was fehlt ?

In diesem Artikel kann naturgemäss nur ein kleiner Teil der jmx4perl Funktionen ausführlich vorgestellt werden. Darüberhinaus bietet jmx4perl noch weitere Möglichkeiten:

```
$ jmx4perl http://localhost:8080/j4p list | grep -i session
cookies             boolean, "Should we attempt to use cookies for session id communication?"
emptySessionPath    boolean, "The 'empty session path' flag for this Connector"
pathname            java.lang.String, "Path name of the disk file in which active sessions"
activeSessions      int [ro], "Number of active sessions at this moment"
maxActive           int, "Maximum number of active sessions so far"
sessionCounter      int, "Total number of sessions created by this manager"
duplicates          int, "Number of duplicated session ids generated"
distributable       boolean, "The distributable flag for Sessions created by this Manager"
sessionMaxAliveTime int, "Longest time an expired session had been alive"
.....
```

Listing 11

```
$ j4psh http://localhost:8080/j4p
[localhost:8080] : ls
....
java.lang:
  type=Memory
  type=OperatingSystem
  type=Runtime
  type=Threading
  .....
 [localhost:8080] : cd java.lang:type=Memory
 [localhost:8080 java.lang:type=Memory] : ls
 java.lang:type=Memory

 Attributes:
   NonHeapMemoryUsage              CompositeData [ro]
   ObjectPendingFinalizationCount  int           [ro]
   Verbose                         boolean
   HeapMemoryUsage                 CompositeData [ro]

 Operations:
   void gc()
 [localhost:8080 java.lang:type=Memory] : cat HeapMemoryUsage
    {
      committed => 85000192,
      init => 0,
      max => 129957888,
      used => 28258696
    }
```

Listing 12

Feingranulare Sicherheit

Der Agent kann neben der Standard HTTP-Security auch auf MBean-Ebene abgesichert werden, in dem man eine *XML-Policy-Datei* in den WAR-Agenten packt. Dabei kann man den Zugriff auf bestimmte jmx4perl Operationen (read, write, exec,..), auf bestimmte MBeans und/oder bestimmte Attribute oder JMX Operationen einschränken. Auch ist es möglich, den Zugriff nur von bestimmten IP-Addressen oder Subnetzen zu erlauben. Mehr zu den Policy-Dateien findet sich in der man page `JMX::Jmx4Perl::Manual`.

JMX Proxy

Wenn es aus welchen Gründen auch immer nicht möglich ist, einen Agenten auf der Zielplattform zu platzieren, dann kann jmx4perl mit Hilfe des *Proxy-Modes* trotzdem verwendet werden. Dazu wird der Agent auf einem dedizierten, kleinen Servlet-Container (Tomcat oder Jetty) installiert. Dieser kommuniziert mit dem Zielserver über JSR-160, der dafür aber eingerichtet sein muss. Das Einschalten von JSR-160 JMX Konnektoren ist für jeden Applikationsserver-Typ unterschiedlich. Für JBoss und Weblogic finden sich Anleitungen dafür unter *http://labs.consol.de/tags/jsr-160/*. Ein `JMX::Jmx4Perl::Request` bei Verwendung eines Proxies sieht etwas anders aus - siehe Listing 14.

Dabei ist `proxy-host` der Host, auf dem der Agent installiert ist, während `target-host` den abzufragenden Server bezeichnet. Die JMX Service URL enthält darüber hinaus noch das Protokol (RMI) und den Port (1099) mit dem die JMX MBeans exportiert sind.

Severseitige Historie

Manchmal reicht es nicht aus, nur den Wert eines Attributes zu messen, sondern die *Zuwachsrate* ist das Interessante. Dazu kann man natürlich auf der Clientseite sich den Wert speichern und diesen von dem Neuen, der etwas später ausgelesen wird, abziehen. Dazu muss der Wert aber lokal gespeichert werden, was z.B. für das Nagios-Plugin `check_jmx4perl` zusätzlich Aufwand bedeutet. Daher bietet der Agent die Möglichkeit, den Wert eines Attributes bzw. den Rückgabewert einer Operation serverseitig im Speicher zu merken und diesen alten Wert bei dem nächsten Request für das gleiche Attribut oder Operation in der Antwort mit zurückzuliefern. Dieser sog. *History-Mode* muss dediziert eingeschaltet werden. Dazu hat der Agent eine eigene MBean registriert, die man z.B. mit `jmx4perl` ansprechen kann - siehe Listing 15.

In diesem Beispiel wird der History-Mode für das Attribut `ThreadCount` der MBean `java.lang:type=Threading` eingeschaltet. Dabei werden max. 10 historische Werte vorgehalten. Die beiden `[null]` Werte sind optionale Argumente, die hier nicht genutzt werden. Mit dem ersten kann ein Pfad angegeben werden, der zweite bezeichnet eine JSR-160 URL falls der Proxy-Mode verwendet wird. Wird nun dieses Attribut ausgelesen, werden bis zu 10 historische Werte zusammen mit einem Zeitstempel in der Antwort zurückgeliefert (Listing 16).

```
# Requests per minute for a connector
# $0: Connector name
# $1: Critical (default: 1000)
# $2: Warning (default: 900)
<Check tc_connector_requests>
  Use = count_per_minute("requests")
  Label = Connector $0 : $BASE
  Value = *:type=GlobalRequestProcessor,name=$0/requestCount
  Critical = ${1:1000}
  Warning = ${2:900}
</Check>
```

Listing 13

```
my $j4p = new JMX::Jmx4Perl(url => "http://proxy-host:8080/j4p");
my $response = $j4p->request(
    new JMX::Jmx4Perl::Request(
         READ,"java.lang:type=Threading","ThreadCount",
         {
            target => {
                        url => "service:jmx:rmi:///jndi/rmi://target-host:1099/jmxrmi",
                        env => { user => "roland", password => "s3cr3!"}
                      }
    ));
```

Listing 14

Aliase

Wie wir gesehen haben, sind die Namen der MBeans oft recht komplex und es gibt viele davon. Um sich das Leben etwas einfacher zu machen gibt es sogenannte *Aliase*, die Kombinationen von MBean Namen, Attributen oder Operationen und eventuell einem Pfad unter einer festen Konstante zusammenfassen. Dabei funktionieren diese Aliase unabhängig vom konkreten Typ des Applikationsservers. Bisher gibt es Aliase für die MXBeans, die in jeder JVM vorkommen. Alle vorhanden Aliase lassen sich mit `jmx4perl aliases` anzeigen und sind im Modul `JMX::Jmx4Perl::Alias` definiert. Verwendet werden können Aliase in `JMX::Jmx4Perl` als Ersatz für die volle Spezifikation der MBean (Listing 17).

Aliase können übrigens auch vom Nagios Plugin `check_jmx4perl` genutzt werden. Hier bietet es sich jedoch an, den flexibleren und neueren Konfigurationsmechanismus zu nutzen, der auch Parameter (z.B. den Namen einer Webapplikation) für vordefinierte Checks verarbeiten kann.

Wie geht's weiter ?

Unsere beiden Schwestern, die Perl-Bibliothek und die Java-Agentin, vertragen sich bestens und bilden ein starkes Duo. Dennoch ist es Zeit weiterzugehen, und so ist die Java-Agentin ohne Groll ausgezogen und hat sich in Jolokia (*http://www.jolokia.org*) umbenannt. Auch ein Grund für die Trennung ist der neue Bekanntenkreis, den sie sich gesucht hat. Der Ansatz, JMX über JSON und HTTP zugänglich zu machen ist nämlich auch für andere Sprachen interessant. Jolokia bietet zusätzlich eine Java-Client Library, eine JavaScript Library ist in Arbeit, andere (Scala, Groovy, Python) sind in Planung. Jedoch bleibt jmx4perl die gute Schwester, die ab Version 0.80 auf Jolokia angewiesen ist.

Fazit

Auch wenn ein agentenbasierter Ansatz zunächst Mehraufwand bedeutet, ist er für alle ausserhalb der Java-Welt der optimale Weg in die innere Maschinerie der JVM. Die Installation des Java-Agentens bedarf zugegebermassen Überzeugungsarbeit und bedeutet extra Arbeit z.B. beim Upgrade des Agenten auf eine neuere Version, aber die Vorteile sind immens. Und selbst wenn es die Coporate Policy nicht erlaubt, 'Fremdsoftware' auf den heiligen JEE Servern zu installieren oder ein Genehmigungsprozess zum Spiessrutenlauf wird, dann bleibt ja immer noch der Proxy-Mode.

Gerhard ist nun zufrieden. Seine JMX Checks laufen in weniger als zwei Sekunden durch und er überwacht damit über 500 JEE-Server mit mehr als 10 Checks in einer einzigen Nagios Server Installation. Dank des ausgereiften Nagios-Checks `check_jmx4perl` kann er täglich seinen Java-Entwicklern auf die Finger klopfen, aber auch selbst dank `JMX::Jmx4Perl` die tollsten Dinge mit seiner Java-Server Landschaft anstellen. Selbst 'Guerilla-Remoting' über selbstgeschriebene MBeans, die man innerhalb einer Java Applikation von überall heraus registrieren kann, wäre möglich. Aber, psst, erzähl das niemals deinem Java-Architekten.

```
$ jmx4perl http://localhost:8080/j4p exec jmx4perl:type=Config \
    setHistoryEntriesForAttribute \
    java.lang:type=Threading ThreadCount [null] [null] 10
```

Listing 15

```
$ jmx4perl -v http://localhost:8080/j4p \
             read java.lang:type=ThreadingThreadCount
....
{"history":[ {"timestamp":1290495645,"value":"10"},
             {"timestamp":1290495640,"value":"12"} ],
 "timestamp":1290495930,
 "status":200,
 "request":{"mbean":"java.lang:type=Threading",
            "attribute":"ThreadCount","type":"read"},
 "value":"14"}
....
```

Listing 16

```
use JMX::Jmx4Perl;
use JMX::Jmx4Perl::Alias;
my $j4p = new JMX::Jmx4Perl(url => "http://localhost:8080/j4p");
print "Heap-Memory used: ",$j4p->get_attribute(MEMORY_HEAP_USED)," Bytes\n";
```

Listing 17

Renée Bäcker

Perl und SVG

SVG ist vielleicht nicht die Neuigkeit schlechthin, aber es ist immer wieder sehr nützlich und gut.

SVG steht für *Scalable Vector Graphics* und ist eine W3C-Empfehlung. Das Format basiert auf XML und ist eine Beschreibungssprache für 2D-Grafiken, die die Kombination von VektorGrafik Formen, gerasterte Bilder (JPG, PNG, etc.) und Text erlaubt.

Der große Vorteil von SVG gegenüber gerasterten Bildformaten wie JPG oder PNG ist, dass sie - wie der Name schon sagt - skalierbar sind. Egal,wie groß man die Grafik braucht, sie sieht immer gut aus.

Mit EventHandlern, die im SVG auch enthalten sind, kann man Skripte aufrufen und so das SVG dynamisch verändern. Ein Beispiel dazu wird in diesem Artikel noch gezeigt.

Werfen wir einen Blick auf eine einfache SVG-Datei (Listing 1).

Da es sich bei SVG um eine XML-basierte Sprache ist, liegt es nahe, dass man hier auch viel mit Perl machen kann - sowohl SVGs erzeugen als auch parsen und weiterverarbeiten.

Die Daten...

Als Beispiel werde ich hier eine Grafik über fiktive Verkaufszahlen von $foo erstellen und das dann nach und nach ausbauen.

Die Daten liegen in einer CSV-Datei, von der ein Auszug in Listing 2 zu sehen ist.

```
id,monat,article,anzahl
1,1,Abonnement,13
2,2,Abonnement,1
3,3,Abonnement,35
13,1,Firmenabo,1
14,2,Firmenabo,1
33,9,Einzelheft,31
34,10,Einzelheft,9
```

Listing 2

Zum Auslesen der Datei verwende ich hier das Modul `DBD::CSV`, damit ich das ganz einfach auch an eine "richtige" Datenbank andocken kann. Das Auslesen der Werte wird in Listing 3 gezeigt.

Damit haben wir die Daten schonmal zur Verfügung. Jetzt gibt es verschiedene Wege, daraus SVGs zu machen. In vielen älteren Tutorials, die man im Internet findet, wird der händische Weg gegangen:

```
print '<rect width="...">';
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
          "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg height="100" width="180" xmlns="http://www.w3.org/2000/svg"
          xmlns:svg="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
    <rect height="44" id="rect_1" width="20" x="10" y="46" />
    <text id="sum_1" x="10" y="38">220</text>
    <rect height="46" id="rect_2" width="20" x="40" y="44" />
    <text id="sum_2" x="40" y="36">230</text>
    <rect height="7" id="rect_3" width="20" x="70" y="83" />
    <text id="sum_3" x="70" y="75">37</text>
    <rect height="20" id="rect_4" width="20" x="100" y="70" />
    <text id="sum_4" x="100" y="62">101</text>
</svg>
```

Listing 1

```
sub get_data {
    my $dbh = DBI->connect(
      'DBI:CSV:',
      '',
      '',
    );
    $dbh->{RaiseError} = 1;

    my $sql = q~SELECT article, SUM(anzahl)
                  FROM svgtest
                  GROUP BY article~;

    my $sth = $dbh->prepare( $sql );
    $sth->execute;

    my @data;

    while(
        my @row = $sth->fetchrow_array
    ) {
        push @data, {
            type => $row[0],
            sum  => $row[1],
        };
    }

    return \@data;
}
```

Listing 3

```
# plot bars
my $y       = 5;
my $x       = 10;
my $counter = 1;
my $width   = 20;

my $image_width  = 180;
my $image_height = 100;
my $margin       = 10;

my $svg = SVG->new(
    width  => $image_width,
    height => $image_height,
);

for my $info ( @sorted ) {
    my $height = int ( $info->{sum} / 5 );
    my $real_y = $image_height - $margin
         - $height;

    $svg->rect(
        x      => $x,
        y      => $real_y,
        id     => 'rect_' . $counter++,
        width  => $width,
        height => $height,
    );

    $x += $width + 10;
}

if ( open my $fh, '>', 'basic_chart.svg' ) {
    print $fh $svg->xmlify;
}
```

Listing 4

Aber wer will sich das wirklich antun. Wesentlich besser sind da Templates. Die bieten den ganz klaren Vorteil, nicht bei jeder Änderung an der SVG am Perl-Code arbeiten zu müssen.

Einsatz von Modulen

Als erstes erzeugen wir eine Basisgrafik mit den Daten, die wir aus der CSV-Datei geholt haben. Wir verwenden dazu das Modul `SVG`. Das Modul bietet für die verschiedenen Formen schon Methoden an. In der Grafik sollen einfach ausgefüllte Balken die Verkaufszahlen zeigen (Listing 4).

Bei der Initialisierung des `SVG`-Objekts wird die Breite und Höhe der Grafik angegeben. Danach wird für jede Verkaufsinfo ein Balken gezeichnet. Da bei SVG die Koordinaten *(0,0)* in der linken oberen Ecke annimmt, werden die Balken "nach unten" gezeichnet, wenn man nicht statt `$real_y` einfach nur `$y` als y-Koordinate für den Balken angibt (siehe Abbildung 1).

Normalerweise ist man aber gewöhnt, dass die Balken "nach oben" gehen (siehe Abbildung 2).

Jetzt sagen diese Balken aber noch nicht allzu viel aus, denn man sieht nicht, welcher Balken für was steht. Aus diesem Grund fügen wir noch einen Text hinzu (Listing 5).

Der erste Text wird direkt mit `x`, `y`-Koordinaten platziert. Mit der Funktion `cdata` wird der eigentliche Text angezeigt. Der zweite Text wird um 90° gedreht angezeigt. Hier ist auch zu sehen, dass nicht unbedingt die Methode `cdata` verwendet werden muss, sondern der Text auch mit der Option `-cdata` festgelegt werden kann. Die resultierende Grafik ist in Abbildung 3 zu sehen.

Eine Analoguhr

Das Balkendiagramm ist eine sehr einfache Grafik. Etwas komplexer ist da eine Analoguhr. Damit nicht jeder Stun-

Abbildung 1: B alken Kopf über

Abbildung 2: Chart auf festem Boden

```
$svg->text(
    id => 'sum_' . $counter,
    x  => $x,
    y  => $real_y - 8,
)->cdata( $info->{sum} );

my $ly = $real_y - 29;
my $lx = $x + 14;

my $t = "translate($lx,$ly) rotate(-90)";

$svg->text(
    id        => 'type_' . $counter,
    transform => $t,
    -cdata    => $info->{type},
);
```

Listing 5

```
my $def = $svg->defs(
    id => 'strich_def',
);

$def->line(
    id => 'strich',
    x1 => 0, y1 => 0,
    x2 => 0, y2 => 8,
    'stroke-width' => 2,
    stroke         => 'black',
);

my $g = $def->g(
    id => 's',
);

$g->use(
    -href     => '#strich',
    transform => 'translate(0,-50)',
);
```

Listing 6

```
<defs id="strich_def">
      <line id="strich" stroke="black"
          stroke-width="2" x1="0" x2="0"
          y1="0" y2="8" />
      <g id="s">
           <use transform="translate(0,-50)"
                 xlink:href="#strich" />
      </g>
</defs>
```

Listing 7

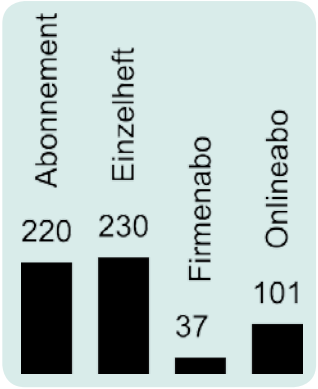

Abbildung 3: Verkaufschart mit Beschriftung

denstrich einzeln gezeichnet werden muss, kann man einen Strich anfangs definieren (Listing 6) und dann immer wieder verwenden.

Das erzeugt eine Definition, wie sie in Listing 7 zu sehen ist.

Innerhalb der `def`-Tags werden die Elemente definiert, die später in der Grafik referenziert werden. Hier ist es eine Definition des Striches. Durch diese Möglichkeit, bestimmte Elemente und Gruppen am Anfang zu definieren und später nur noch darauf zu referenzieren, können SVG-Dateien kleiner gehalten werden. Die Elemente werden nicht automatisch gerendert, erst durch die Referenzierung im SVG-Dokument werden sie dann angezeigt.

Die Referenzierung passiert über das `use`-Tag, das ein `xlink`-Attribut besitzen muss, in dem dann die ID des referenzierten Elements zu finden ist.

Mit dem `g`-Tag werden Elemente gruppiert. Der Vorteil bei Gruppierungen besteht darin, dass eine Transformation der Gruppe eine Transformation aller gruppierten Elemente bedeutet. Man kennt das aus anderen Programmen, in denen mehrere Elemente markiert und dann zusammen verschoben werden können.

Mir persönlich hilft es immer, wenn ich eine Grafik mit Inkscape oder anderen Tools mal zeichne, damit ich eine grobe Vorlage habe und diese dann mit dem Programm "nachbaue" bzw. mir ein Template daraus generiere.

SVGs mit komplexen Formen

Sollen jetzt noch komplexe Formen - z.B. der Umriss von Europa - gezeichnet werden, muss man mit Pfaden arbeiten. In diesem Beispiel wird eine einfache kubische Bezierkurve gezeichnet (Listing 8).

Die gezeigten Beispiele verdeutlichen, dass man mit dem `SVG`-Modul noch viel selbst berechnen muss. Da muss man einiges ausprobieren. Wesentlich bequemer kann das Diagramm für die Verkaufszahlen mit dem Modul `SVG::TT::Graph` erzeugt werden (Listing 9).

```
$svg->path(
  d      =>
    'M 10 100 C 10 50 200 50 200 100',
  id     => 'pline_1',
  style => {
    fill               => '#808000',
    stroke             => '#000000',
    'stroke-width'     => '1px',
    'stroke-linecap'   => 'butt',
    'stroke-linejoin' => 'miter',
    'stroke-opacity'   => 1,
  }
);
```

Listing 8

```
my $graph = SVG::TT::Graph::Pie->new({
   'height' => '500',
   'width'  => '300',
   'fields' => \@fields,
});

$graph->compress(0);

$graph->add_data({
   'data'  => \@sales,
   'title' => 'Sales 2010',
});
```

Listing 9

```
use SVG::Calendar;

# Create an A4 calendar
my $svg = SVG::Calendar->new(
    page => 'A4',
);

# create a calendar for the year 2011
$svg->output_year( 2011, 'foo_magazin' );
```

Listing 10

```
my %dialog = (
    start    => [ qw/software schulung/ ],
    software => [qw/referenzen kompetenz/],
    referenzen => [qw/start/],
);

my $graph = Graph::Easy->new;

for my $page ( keys %dialog ) {
    for my $next ( @{ $dialog{$page} } ) {
        $graph->add_edge( $page, $next );
    }
}

if ( open my $out, '>', 'easy.svg' ) {
    print $out $graph->as_svg({
        standalone => 1,
    });
}
```

Listing 11

Für die Erstellung des Kuchendiagramms wird ein Template für Template-Toolkit herangezogen. Man merkt an diesem Beispiel, dass das Modul einiges an Arbeit erspart und ein sauberes Diagramm erstellt (Abbildung 4). Möchte man mit der Datei weiterarbeiten, muss man die Komprimierung abschalten (`compress(0)`).

CPAN nimmt weitere Arbeit ab

Auf CPAN gibt es noch eine ganze Menge Module, die bei der Arbeit mit SVG-Dokumenten helfen. Möchte man ein Verlaufsdiagramm haben, kann man `SVG::Sparkline` nehmen und `GD`-Kenner werden sich mit `GD::SVG` sehr wohl fühlen.

Für einige Anwendungsbereiche gibt es schon fertige Module auf dem CPAN. In der Ausgabe 4 habe ich schon das Modul `GraphViz::Regex` vorgestellt, mit dem Reguläre Ausdrücke in einer SVG-Datei dargestellt werden können. Ein weiteres Modul ist `SVG::Calendar` (siehe Listing 10).

Mit `Graph::Easy` können Graphen sehr einfach als SVG ausgegeben werden. In Listing 11 ist ein einfaches Beispiel zu sehen, wie man die Dialogstruktur einer Webseite als SVG-Graph ausgeben kann. Man muss sich dafür neben `Graph::Easy` noch das Zusatzmodul `Graph::Easy::As_svg` installieren.

Wenn die Grafik weiterverwendet werden soll, sollte man darauf achten, dass bei der `as_svg`-Methode der Parameter `standalone` gesetzt wird. Das fügt die XML-Deklaration vor der Grafik an - siehe Abbildung 5.

Verarbeiten von SVGs

SVGs müssen ja nicht nur erzeugt werden, hin und wieder müssen die Grafiken auch weiterverarbeitet werden. Grundsätzliche Informationen über die Grafik bekommt man mit dem Modul `Image::Info` (Listing 12).

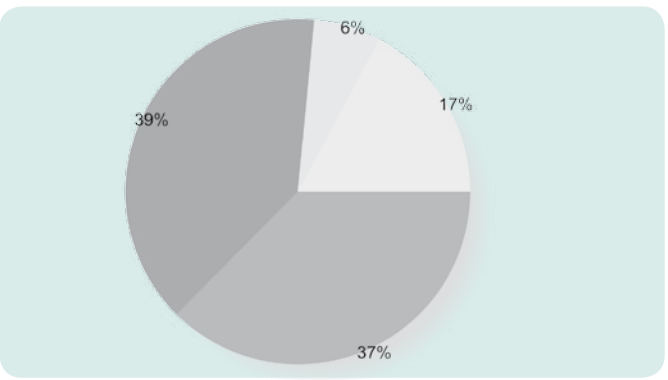

Abbildung 4: Kuchendiagramm

Hierbei ist darauf zu achten, dass die Datei die XML-Deklaration beinhaltet. Wird zum Beispiel das `Graph::Easy`-Beispiel ohne den Parameter `standalone` ausgeführt, kommt `Image::Info` damit nicht zurecht und meint, dass es mit dem Dateiformat nicht umgehen kann.

Aber solche Informationen aus den SVG-Dateien zu holen ist nicht das einzige was man mit SVGs machen kann. Weiter oben wurde gezeigt, wie man ein Kuchendiagramm erstellt. Allerdings reicht mir das nicht. Ich möchte ein paar Knoten so ändern, dass bei einem Klick auf ein Kuchenstück einfach die Prozentzahl neben dem Kuchenstück geändert wird.

Dazu muss das SVG geparst werden mit einem DOM-Objekt als Ergebnis. Für diesen Zweck gibt es `SVG::Parser`. Die `parse`-Funktionen liefern dann ein `SVG`-Objekt zurück (Listing 13).

Jetzt werden die Teile rausgesucht, die die Kuchenstücke darstellen. Durch einen Blick in die Datei, die mit dem Skript weiter oben (Listing 9) generiert wurde, weiß ich, dass es `path`-Elemente sind, deren id mit einem "w" beginnt.

```
my @paths = $dom->getElements( 'path' );
```

Bei diesen Elementen wird jetzt ein `onclick`-Handler eingefügt, wie in Listing 14 zu sehen ist.

Das `setAttributes` fügt neue Attribute hinzu. Die bestehenden Attribute bleiben unangetastet, solange sie nicht explizit angegeben werden. Wird ein `undef` für ein Attribut übergeben, wird dieses gelöscht.

Dann wird noch das JavaScript hinzugefügt (Listing 15). Die Prozentzahlen stehen in einem `text`-Element, dessen ID mit einem "d" beginnt und der Nummer, die auch in der ID des `path`-Elements zu finden ist. Also gehört `<text id="d1">` zu `<path id="w1">`.

Wenn ich jetzt die Grafik anschaue und auf ein Kuchenstück klicke, wird die Prozentzahl geändert.

SVGs in Pixelgrafiken umwandeln

Mit dem Modul `SVG::Rasterize` kann man die Grafiken in andere Formate umwandeln. Für so etwas kann man in der Regel auch das Programm Inkscape verwenden. Dem Autor fehlten aber in allen Tools gewisse Funktionen und hat so `SVG::Rasterize` geschrieben (siehe Listing 16).

```
use Image::Info qw(image_info dim);

my $file = 'easy.svg';
my $info = image_info( $file );

my $title  = $info->{SVG_Title};
my ($w,$h) = dim( $info );

print "$file: $title -> $w x $h\n";
```

Listing 12

```
use SVG::Parser;

my $file = 'pie.svg';

my $parser = SVG::Parser->new;
my $dom    = $parser->parse_file(
    $file
);
```

Listing 13

```
PATH:
for my $path ( @paths ) {
    my $id = $path->getElementID;

    my ($nr) = $id =~ m{ \A w (\d+) }xms;

    next PATH if !$nr;

    $path->setAttributes({
        onclick => "change_value($nr);",
    });
}
```

Listing 14

```
my $script = qq~
function change_values(nr) {
  var textelement =
    document.getElementById( 'd' + nr );
  var percent     = nr * 3;
  textelement.firstChild.nodeValue =
    percent + '%';
}
~;
$dom->script(
    type   => 'text/javascript',
    -cdata => $script,
);
```

Listing 15

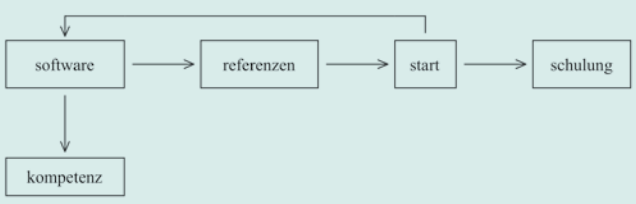

Abbildung 5: easy

```
use SVG::Rasterize;
use SVG::Parser;

my $file = 'easy.svg';

my $svg = do{
    local (@ARGV,$/) = $file;
    <>
};

$svg =~ s{ <!-- .*? --> }{}xmsg;

my $parser = SVG::Parser->new;
my $dom    = $parser->parse( $svg );

my $rasterizer = SVG::Rasterize->new;
$rasterizer->rasterize(
    svg => $dom,
);

$rasterizer->write(
    type      => 'png',
    file_name => 'easy_graph.png',
);
```

Listing 16

Allerdings hatte ich in meinen Tests immer wieder ein paar Schwierigkeiten mit Grafiken, die ich in Inkscape erstellt hatte und mit `SVG::Rasterize` in ein anderes Format umwandeln wollte. In der Regel hat der Autor des Moduls aber prompt auf Anfragen reagiert. Grafiken, die ich in anderen Tools erstellt habe, habe ich nicht getestet.

Mit den oben erzeugten SVG-Dateien funktioniert das Umwandeln problemlos - fast. `SVG::Rasterize` hat Probleme, wenn in den SVG-Dateien Kommentare vorkommen (getestet Version: 0.003004). Aus diesem Grund wurden die Kommentare erst entfernt, die regulären Ausdrücke sind zwar keine absolut sichere Lösung, aber in den meisten Fällen absolut ausreichend.

Interaktive Grafiken

Wie ich eingangs schon erwähnt habe, kann man auch Skripte (z.B. JavaScript) in SVGs einbetten. Das werde ich jetzt dazu nutzen, die Grafik etwas aufzuwerten. Wenn ich auf einen Balken klicke, werden mir die Monatszahlen für den jeweiligen Typ angezeigt.

Es gibt neben `onclick` noch jede Menge weiterer Events bei SVG, mit denen man arbeiten kann. Viele dürften aus dem Webbereich schon bekannt sein, wie z.B. `onclick`, `onmouseover` etc. Aber es gibt auch Events, die SVG-spezifisch sind, wie z.B. `SVGError` oder `SVGZoom`. Eine gute Seite für einen Überblick ist http://www.carto.net/papers/svg/eventhandling/.

```
use DBI;
use HTTP::Daemon;
use HTTP::Status;
use HTTP::Response;
use JSON::Syck;
use SVG::TT::Graph::Pie;

my $d = HTTP::Daemon->new(
    LocalPort => 8088,
) || die;

my %content_type = (
    'Content-Type' => 'application/json',
);

print "<URL:", $d->url, ">\n";
while(my $c = $d->accept) {
    while ( my $r = $c->get_request ) {
        my $path = $r->uri->path;
        if ( $path eq '/chart.svg' ) {
            my $file = create_chart();
            $c->send_file_response( $path );
        }
        elsif ( $path eq '/chart.cgi' ) {
            my $p = $r->url->query;
            my ($nr) = $p =~ m/nr=(\d+)/;

            $c->send_response(
                HTTP::Response->new(
                    200, 'OK',
                    [%content_type],
                    JSON::Syck::Dump(
                        get_details($nr),
                    )
                )
            );
        }
        else {
            $c->send_error(RC_FORBIDDEN)
        }
    }
    $c->close;
    undef($c);
}
```

Listing 17

Weil die Daten nicht direkt mit der Grafik mitgeliefert werden, muss die Grafik mit einem Server sprechen. Dieser Server ist relativ schnell geschrieben (Listing 15). Dazu wird `HTTP::Daemon` verwendet. Für den Datenaustausch verwende ich JSON.

Es werden nur zwei mögliche URIs akzeptiert. Der erste, der die Grundgrafik liefert und der zweite, der die Überschrift für das angeklickte Kuchenstück liefert. In der Funktion `create_chart()` sind die Funktionalitäten aus Kuchendiagramm erzeugen und JavaScript einfügen kombiniert.

Die Grundgrafik ist von oben bekannt, und Abbildung 6 zeigt wie es vor einem Klick aussieht und Abbildung 7 danach.

Perl 6 und SVG

Bis jetzt habe ich nur gezeigt, wie man mit Perl 5 SVGs erzeugen kann. Da aber im Juli letzten Jahres die erste Version von Rakudo Star veröffentlicht wurde, möchte ich hier auch noch zeigen, wie man mit Perl 6 SVGs erstellen kann. Moritz Lenz hat bereits eine SVG-Bibliothek für Perl 6 geschrieben.

Das Beispiel in Listing 19 zeigt, wie die Basisgrafik der Verkaufszahlen mit Perl 6 und dem SVG-Modul erstellt wird. Bevor das laufen kann, muss neben Rakudo Star auch `svg` und `svg-plot` installiert werden. Dazu kann das Tool `proto` verwendet werden (Listing 18).

Die ersten drei Befehle können ausgelassen werden, wenn Rakudo bereits installiert ist. In der aktuellen Version von `proto` taucht `XML::Writer` nicht in der Projektliste (`projects.list`) auf. Dieses muss dann manuell hinzugefügt werden.

```
xml-writer:
    home: github
    owner: masak
```

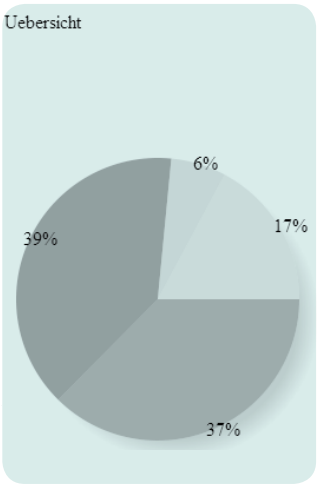

Abbildung 6: Vor dem Klick auf ein Tortenstück...

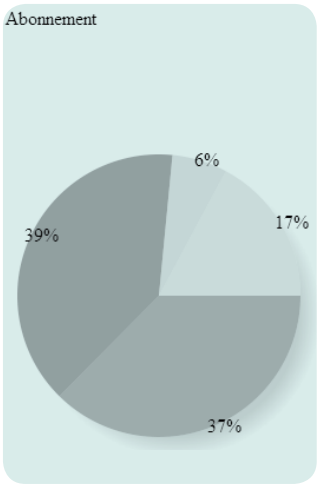

Abbildung 7: ... und danach

Wer Probleme bei der Installation feststellt und eine Meldung wie `Method 'd' not found for invocant of class 'Str'` muss noch die Datei `Installer.pm6` im `proto/lib`-Verzeichnis patchen. Aus

```
if $dir !~~ :d {
```

in Zeile 497, muss

```
if $dir.IO !~~ :d {
```

werden.

Dann kann es losgehen - siehe Listing 19.

Mit SVG kann man noch andere tolle Sachen machen. Wer sich dafür interessiert, wie Sprache entsteht, sollte sich mal *http://svg-whiz.com/svg/linguistics/theCreepyMouth.svg* (Links gibt's auch auf Delicious) anschauen. Ein sehr schönes Beispiel dafür, was man mit SVG alles machen kann.

```
$ git clone https://github.com/masak/proto.git
$ cd proto
$ ./proto.pl install rakudo
$ ./proto.pl install xml-writer
$ ./proto.pl install svg
$ ./proto.pl install svg-plot
```

Listing 18

```
use v6;

use SVG;
use SVG::Plot;

my @csv_lines = lines 'svgtest';
@csv_lines.shift;

my %hash;
for @csv_lines -> $line {
    my @info = $line.split(',');
    %hash{@info[2]} += @info[3];
}

my @labels = %hash.keys;
my @data   = %hash.values;

my $svg = SVG::Plot.new(
    width   => 400,
    height  => 350,
    values  => ([@data]),
    title   => 'Verkaufszahlen',
    legends => ('Data series 1'),
    :@labels,
).plot(:bars);

my $fh = open( 'test.svg', :w );
$fh.say( SVG.serialize($svg) );
$fh.close();
```

Listing 19

Alexander Halle

OTRS-Community-Meetings

2. OTRS-Community-Meeting

OTRS ist eine komplett auf Perl basierende Plattform für die verbreiteten Opensource-Produkte "OTRS Helpdesk" und "OTRS ITSM". Das vergangene Jahr stand vor allem im Zeichen der lange erwarteten Version 3, aber vor und insbesondere hinter den Kulissen wurde auch fleißig an einer verstärkten Förderung der großen Community gearbeitet.

Als eine Maßnahme wird es daher 2011 weitere OTRS-Community-Treffen (OCM) geben. Das OCM 2 wird am 19. und 20. März auf den Chemnitzer Linuxtagen (CLT) stattfinden mit durchgängigem Programm. Neben dem Community-Stand und zwei Vorträgen wird es sowohl einen Admin- und einen Entwickler-Workshop am Samstag geben, sowie ein großes Struktur-Treffen der Community am Sonntag von 11 bis 15 Uhr. Für die Workshops sind eine Anmeldung plus 5 EUR Beitrag erforderlich, nähere Informationen sind auf otrs.org zu finden.

Das OCM 2 ist eine Gemeinschaftsveranstaltung der OTRS-User, der OTRS AG und der cape IT GmbH. Zudem haben bereits die Planungen für das OCM 3 begonnen, das Hinweisen zu Folge auf dem Linuxtag Berlin im Mai stattfinden soll. Man darf gespannt sein.

1. OTRS-Community-Meeting

Am zweiten August 2010 war es also so weit, das Community-Treffen in Frankfurt am Main würde stattfinden, aber es hatten nur sechs Leute zugesagt. Was war geschehen und wieso war das Treffen trotzdem ein großer Erfolg?

Geplant war alles ganz anders. Anfang Juli hatte Shawn Beasley, Community Manager der OTRS AG, zur OTRS-3.0-Beta-Release-Party geladen. Großartig dachte ich, endlich Mal andere User kennen lernen und sich gleichzeitig auf OTRS 3.0 vorbereiten. Mehrere Dutzend Anmeldungen zeigten dann, dass die User das wohl ähnlich sahen. So war ich dann auch mit meiner Enttäuschung nicht allein, als das Treffen am 25. Juli kurzfristig abgesagt werden musste - Shawn war leider ernsthaft erkrankt.

Die zweite Überraschung kam noch am selben Tag. Frank Tailor schlug vor, statt der Release-Party ein Community-Meeting zu veranstalten mit Schwerpunkt OTRS 3.0 und auch für die Kosten war gesorgt. Respekt, das ist Initiative, dachte ich mir und beschloss, ihn zu unterstützen. Es folgten ein paar anstrengende Tage, um das Treffen zu planen. Shawn unterstützte uns vom Krankenbett aus und Martin Edenhofer, Erfinder von OTRS, hatte spontan als Tutor zugesagt und kümmerte sich auch um die Technik. Die Koordination per Wiki klappte ganz gut, allerdings sollte noch eine weitere Überraschung folgen. Als ich am Frankfurter Bahnhof meine Mails checkte, erfuhr ich, dass Frank ausfiel. Er war kurzfristig krank geworden. Das fing ja gut an ...

Ich kam also pünktlich um 11:00 Uhr am Haus der Jugend an und traf kurze Zeit später schon auf Martin. Ein paar Minuten danach kamen auch Torsten Thau und René Böhm von der cape IT GmbH, so dass wir nach einer Vorstellungsrunde direkt loslegen konnten. Auf dem Programm standen die Vorstellung der OTRS-3.0-Beta, eine Live-Demonstration und das ausführliche Testen. Vorher gab es aber noch eine herzliche Begrüßung von Shawn per Video-Botschaft, er war sichtlich enttäuscht, dass er nicht da sein konnte.

Die OTRS-3.0-Vorstellung ging dann auf die Neuerungen aus Code-Sicht ein, im Gegensatz zu den schon bekannten neuen Features. Und zwar wird als Framework jetzt jQuery genutzt statt YUI und das frühere Tabellen-basierte Layout basiert nun auf CSS und XHTML. Teilweise wird schon CSS3 eingesetzt, aber nur für die elegantere Optik. Übrigens gibt es eine Reset-CSS-Datei namens core.default.css.

Dann stellte uns Martin die einzelnen Module vor, die wir dann auch direkt testeten. Ab dem Punkt begann das Treffen dann auch, eine unerwartete Dynamik zu entwickeln. Jeder probierte erst für sich rum und fand auch schnell Bugs, da es sich ja um die Beta 1 handelte. Die Neugier trieb uns dann allerdings ständig zwischen den Notebooks hin und her und der arme Martin kam vor lauter Anschauen, Mitschreiben und Beantworten von Fragen und Ideen kaum mehr mit. Da merkte ich dann auch so richtig, dass ich es mit drei Entwicklern zu tun hatte. Ich glaube nach ein paar Stunden ohne mich, hätten sie mir voller Stolz ein neues Beta-Release präsentiert.

Jedenfalls konnte ich viel über OTRS 3.0 und die zu Grunde liegende Architektur lernen. Drei Stunden und 31 Bugs später, beschlossen wir schließlich ein Mittagessen einzulegen. Die Atmosphäre war zwischenzeitlich sehr entspannt und freundschaftlich geworden und ähnlich lief dann auch das gemeinsame Essen ab. Nur die Restaurant-Wahl war schwierig, in keinem gab es mehr als drei Gäste. Das Essen war eher durchschnittlich, beim nächsten Treffen werden wir da gründlicher planen.

Zurück im Besprechungsraum, übrigens ein kleines Haus im Hof, da das Skyline-Deck so kurzfristig nicht mehr zu bekommen war, kam dann auch schließlich Holger Vier dazu. Nach einer neuen Vorstellungsrunde war damit die freie Diskussionsrunde eröffnet. Holger ist Geschäftsführer der büKOM Systemhaus GmbH, so dass nun schon zwei User anwesend waren ;) Ich selbst arbeite in der IT-Abteilung der radprax Gesellschaft für Medizinische Versorgungszentren mbH.

Die Dynamik und gute Atmosphäre setzten sich unvermindert fort. Unser erstes Thema war die Zukunft von OTRS unter freier Lizenz. Martin konnte uns da wie erwartet beruhigen, OTRS wird auch weiterhin vollständig Open-Source bleiben, ein Open-Core-Modell oder ähnliches ist nicht geplant. Hauptsächlich diskutierten wir aber die Zusammenarbeit und Kommunikation und wie man diese weiter verbessern kann. Zwei konkrete Ziele kamen dabei heraus : Zum einen, die User bei Code-Beiträgen zu fördern, damit diese zunehmen, zum anderen die Bündelung von Ressourcen, um Wünsche gemeinsam schneller zu erreichen.

Daher beschlossen wir ein weiteres OTRS-Community-Treffen anzustoßen. Bei der bloßen Idee ist es dann auch nicht geblieben, die Ankündigung zum zweiten Community-Treffen findet sich in der vorliegenden Ausgabe des Perl-Magazins. Tatsächlich ist sogar bereits ein drittes Treffen in Gespräch, so dass es bei der Planung schon anstrengend wurde “OTRS-Community-Treffen“ zu schreiben. Es kam, wie es kommen musste, die Dinger heißen jetzt OCM ;)

Gegen 16 Uhr beschlossen wir schließlich, das Treffen noch entspannter in einem Cafe fortzusetzen. Das funktionierte dann auch so gut, dass sich unsere Wege erst um 19 Uhr trennten. Für das OCM 1 danke ich abschließend insbesondere Frank Tailor, ohne den alles anders verlaufen wäre und natürlich Martin Edenhofer und Shawn Beasley für ihre tatkräftige Unterstützung.

Viacheslav Tykhanovskyi

Mit WebSockets und Perl in HTML5 eintauchen

HTML5

HTML5 ist ein neuer HTML Standard, welches sich das W3C 2007 angenommen hat. Der erste öffentliche Arbeitsentwurf wurde 2008 veröffentlicht. Die Arbeit ist noch im Gange und wird den Empfehlungsstatus wahrscheinlich 2012 erreichen. Aber viele Bereiche sind schon stabil und haben benutzbare Implementationen in verschiedenen Browsern.

Zwischen vielen neuen Dingen bringt es WebSockets. WebSockets sind persistente bidirektionale Vollduplex TCP-Verbindungen zwischen einem Server und einem Browser. Im Vergleich zu anderen Technologien, die heute verwendet werden - wie z.B. langes polling, comet etc - benötigt es nur eine Verbindung. Es werden nur Nutzdaten (keine HTTP-Header nachdem die Verbindung aufgebaut wurde) und eine extrem einfache asynchrone API auf der Client-Seite. Nimm diesen JavaScript-Code:

```
var ws =
    new WebSocket('ws://localhost:3000');

ws.onopen = function() {
    alert('Connection is established!');
};

ws.onclose = function() {
    alert('Connection is closed');
};

ws.onmessage = function(e) {
    var message = e.data;
    alert('Got new message: ' + message);
};

ws.send('Hello, world!');
```

Hier registrieren wir Callbacks (anonyme Funktionen), die bei bestimmten Ereignissen aufgerufen werden, wenn diese eintreten. So wird `onopen` aufgerufen, wenn eine Verbindung mit dem WebSocket Server aufgebaut wird. `onclose` wird verwendet, wenn diese Verbindung geschlossen wird und `onmessage`, wenn eine Nachricht empfangen wird. Es gibt auch ein `onerror` Callback, aber das wird aus Gründen der Einfachheit hier nicht gezeigt.

Jedes Mal wenn Du eine Nachricht senden willst, musst Du die `send` Funktion aufrufen. Wie Du sehen kannst, musst Du Dich auf der Client-Seite nicht um Handshakes, Frames, Puffer und ähnlichem kümmern.

Werfen wir einen Blick auf einen tieferen Level

WebSocket Verbindungen beginnen mit einem Handshake zwischen einem Server und einem Browser mit einem HTTP/1.1 `Upgrade` Header. Hier ein Beispiel was hinter den Kulissen passiert:

```
> Browser request
GET / HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
Sec-WebSocket-Key1: \
    18x 6]8vM;54 *(5:  {   U1]8  z [  8
Sec-WebSocket-Key2: \
    1_ tx7X d  <  nw  334J702) 7]o}` 0

Tm[K T2u

< Server response
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Location: ws://example.com/

fQJ,fN/4F4!~K~MH
```

In der letzten WebSocket Spezifikation Entwurf #76 (das jetzt Version 00 ist *http://tools.ietf.org/id/draft-ietf-hybi-thewebsocketprotocol-00*) wurde der Handshake geändert, indem aus Sicherheitsgründen `Sec-*` Felder hinzugefügt wurden. Es gibt aber noch Browser, die den Entwurf #75

unterstützen (z.B. einige Android Mobilbrowser) und die aktuellen Implementationen sollten das im Hinterkopf behalten. Zusätzlich wird sich das Protokoll erneut ändern (Version 02 *http://tools.ietf.org/id/draft-ietf-hybi-thewebsocketprotocol-02* hat schon eine viel kompliziertere Framestruktur, obwohl aktuelle Browser das nicht unterstützen). Aber die Client-API sollte so bleiben wie sie ist.

Andere HTTP Header, die mit übergeben werden, werden meistens ignoriert. Aber es gibt einen, der wirklich wichtig ist - und das ist `Cookie`. Durch die Verwendung von Cookies werden wichtige Aspekte wie Authentifizierung, User Tracking und Sessions möglich.

Nach dem Handshake - wenn er erfolgreich war - können Server und Browser Daten hin und her schicken wenn sie einen einfachen Frame verwenden (wird wahrscheinlich noch geändert):

```
\x00...DATA...\xff
```

Die Daten selbst sollten in UTF-8 kodiert sein. Die Spezifikation beschreibt auch die Wege, wie Binärdaten übertragen werden können, aber das ist aktuell nicht empfohlen und ich werde das hier auch weglassen.

Die WebSocket Verbindung kann durch eine Standard TLS/SSL-Verschlüsselung abgesichert werden. Der Client wird das berücksichtigen, wenn er über `wss://` anstatt `ws://` die Verbindung aufbaut.

Die Verwendung von TLS/SSL wird nicht nur empfohlen, wenn der Traffic verschlüsselt werden soll, sondern auch wenn Du an Proxy Servern und Firewalls vorbei musst, die sonst WebSocket Verbindungen einfach blockieren oder nicht verstehen.

Browser Kompatibilität

Da sich die WebSocket Spezifikation ändert, ist es sinnvoll zu wissen, welcher Browser sie unterstützt. Ich habe eine Tabelle mit den Clients zusammengestellt, die ich überprüft habe:

```
Draft #75
---------
Chrome 5
Safari 5.0

Draft #76 (version 00)
---------
Firefox 4b
Safari  5.0.2
Chrome  6
Opera   10.70

iOS     4.2
```

Achtung: Safari unterstützt leider keine TLS/SSL WebSocket Verschlüsselung. Hoffentlich wird das möglichst schnell geändert.

Achtung: Internet Explorer 9 wird wahrscheinlich WebSockets unterstützen.

Das sind ganz neue Versionen, die heute noch nicht weit verbreitet sind. Bedeutet das also, dass wir zurzeit noch keine WebSockets verwenden können?

Nicht-WebSocket Browser Support

Selbst wenn die meisten Browser WebSockets unterstützen werden, wird es immer solche geben, die nicht aktualisiert wurden oder die sie nie unterstützen werden. Für solche Fälle wurden einige wirklich weise Techniken implementiert. Ich werde über die zwei größten Lösungen schreiben. Vielleicht gibt es andere, aber das wäre eine weitere Studie.

Flash Fallback

Dieser Workaround benutzt ein Flash Fallback, der eine persistente Verbindung aufbaut und das ganze Parsen des Handshakes übernimmt. Wenn ein Browser also keine WebSockets unterstützt benutzt es nativ Flash.

Es gibt zwei Anforderungen die erfüllt sein müssen, damit das funktioniert. Als erstes müssen Script-Header in der Seite eingebunden werden, die Flash und den passenden JavaScript Code laden (siehe Listing 1).

Die zweite Anforderung ist ein Flash Socket Policy Server, der auf Port 843 laufen muss. Alles was dieser Server tun muss, ist diese Nachricht bei jedem Request zu senden - Listing 2.

```
<script type="text/javascript">
    // Only load the flash fallback when needed
    if (!('WebSocket' in window)) {
        document.write([
            '<scr'+'ipt type="text/javascript" '
                + 'src="web-socket-js/swfobject.js"></scr'+'ipt>',
            '<scr'+'ipt type="text/javascript" '
                + 'src="web-socket-js/FABridge.js"></scr'+'ipt>',
            '<scr'+'ipt type="text/javascript" '
                + 'src="web-socket-js/web_socket.js"></scr'+'ipt>'
        ].join(''));
    }
</script>
<script type="text/javascript">
    if (WebSocket.__initialize) {
        // Set URL of your WebSocketMain.swf here:
        WebSocket.__swfLocation = 'web-socket-js/WebSocketMain.swf';
    }

    // Normal WebSocket initialization
</script>
```

Listing 1

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
    "/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
<site-control permitted-cross-domain-policies="master-only"/>
<allow-access-from domain="*" to-ports="*" secure="false"/>
</cross-domain-policy>
```

Listing 2

Es gibt ein paar Schwierigkeiten für User, die hinter einem Proxy sitzen. Aber diese können durch das manuelle übergeben von Proxy Parametern an den WebSocket Konstruktor gelöst werden oder indem man den User diese selbst festlegen lässt.

Ich selbst habe diesen Workaround benutzt und er funktioniert gut. Aber vergiss nicht, den Flash Policy Server zu starten oder Du wirst eine harte Zeit haben wenn Du das Problem in Deinem Code suchst.

Socket.IO

Diese Bibliothek ist eine vereinheitlichte Klasse für viele Techniken persistenter HTTP-Verbindungen, die nicht nur WebSockets beinhaltet, sondern auch Flash Sockets, XHR long polling, iframes und anderes.

Es rät automatisch an Hand Deines Browsers, welcher Transportmechanismus gewählt werden muss. Die API ist nah genug an der von WebSockets.

```
socket = new io.Socket('localhost');
socket.connect();
socket.on('connect', function(){
    // connected
});
socket.on('message', function(data){
    // data here
});
socket.send('some data');
```

Um diese Bibliothek nutzen zu können, muss nur eine einzige Zeile in Deiner HTML Quelldatei eingefügt werden:

```
<script src=
 "http://cdn.socket.io/stable/socket.io.js">
</script>
```

Die Bibliothek löst das Proxy-Problem, in dem es einfach zu einem anderen Transportmechanismus wechselt.

Da es eine weit entwickelte Bibliothek ist, benötigt sie einen smarteren Server, der auch alle diese Transportmechanismen unterstützt. Du musst nicht nur WebSockets unterstützen, sondern auch HTTP/1.1 mit all seiner Komplexität.

Es gibt ein Socket.IO für Perl, das auf *Mojolicious* implementiert ist: *https://github.com/xantus/ignite* (long polling und WebSockets).

Probleme mit WebSockets

Es gibt viele Gründe, warum das WebSocket Protokoll wahrscheinlich in der Zukunft wieder geändert wird. Diese Probleme sind durch den Willen, WebSockets einfach implementieren und benutzen zu können, begründet.

Unterstützung von Reverse Proxy

Der Entwurf #76 führte eine Challenge ein, die als Body an den WebSocket Client gesendet wird. Das Problem ist, dass es keinen `Content-Length` Header hat und Proxies es als eine nächste Antwort betrachten oder einfach ohne eine Fehlermeldung an den User hängen bleibt. Einige Lösungen wurden vorgeschlagen, wie dieses Problem zu lösen ist - einschließlich eines `Content-Length: 8` oder `Connection: close` Headers, Verwendung der `POST` Methode und anderem.

Framing

Das `\x00..\xff` Framing wurde nicht als beste Lösung betrachtet, weil die Verwendung von nur einem Byte als Nachrichtentrenner genug sein sollte (die Versionen 01, 02 führen ein weiterentwickeltes Framing ein). Es ist auch nicht klar, ob es Begrenzungen der Framegrößen geben sollte um z.B. DDoS-Attacken zu verhindern.

Andere Probleme

Andere Probleme: Metaframes, die für zusätzliche Einstellungen benutzt werden können, wie das Setzen von Cookies nach dem Handshake; der benötigte Level der HTTP-Kompatibilität; festlegen, wie wichtig die Keep Alive Timeouts sind; etc.

WebSockets im wirklichen Leben einsetzen

Was brauchen wir, um mit WebSockets zu beginnen? Natürlich brauchen wir einen geeigneten Webbrowser. Aber das bedeutet keine zusätzliche Arbeit. Die wirkliche Arbeitet ist der WebSocket Server.

WebSocket Server Anforderungen

Die offensichtlichen Anforderungen sind:

- Er muss wissen, wie WebSocket Handshake und Frames geparst werden
- Er muss asynchron und nicht-blockierend sein
- Er muss in der Lage sein, viele persistente Verbindungen zu halten

Das Parsen von Handshake und Frames ist nicht so schwierig wie das Parsen von z.B. HTTP/1.1 Nachrichten, aber es erfordert das Lesen von einigen Spezifikationen, einschließlich der Tatsache, dass sie sich sehr bald wieder ändern.

Der Server muss wegen der asynchronen Natur der WebSockets asynchron sein. Du kannst jederzeit Nachrichten erhalten und jederzeit Nachrichten versenden - auch zur gleichen Zeit. Da Threads die Wurzel allen Übels sind, werden wir nicht-blockierende Sockets für das Lesen und Schreiben benutzen. Der Unterschied zwischen asynchron und nicht-blockierend kann mit einem Ausdruck erklärt werden: "Der Nicht-blockierende Modus teilt uns mit, wann wir mit einer Aktion beginnen müssen; der asynchrone Modus sagt uns, wann die Aktion beendet ist."

Die dritte Anforderung ist nicht wirklich eine Anforderung wenn Du WebSockets nur zum Testen benutzt, aber es ist gut, das im Hinterkopf zu behalten.

Perl Optionen

Es gibt Perl Webframeworks, die WebSockets "out of the box" unterstützen, wie z.B. *Mojolicious*, das zusätzlich einen WebSocket Client hat. Einiges hat sich in `Dancer` und `Tatsumaki` getan. Standalone WebSocket Server gibt es auf Basis von verschiedenen Event-Loops, z.B. `Net::Async::WebSocket`, `AnyEvent::HTTP::Server` und auch `Plack`-Middleware (*https://github.com/motemen/Plack-Middleware-WebSocket*).

Mehr Infos darüber, wie ein WebSocket Server mit Perl umgesetzt werden kann, findest Du in folgendem Artikel: *http://showmetheco.de/articles/2010/11/timtow-to-build-a-websocket-server-in-perl*.

Die Beispiele, die später in diesem Artikel gezeigt werden, verwenden meinen eigenen WebSocket Server ReAnimator und die Event-Loop EventReactor, aber die können leicht auf eine andere Event-Loop portiert werden. Es gibt dort nichts Besonderes bezüglich des Parsens von WebSocket Handshakes (oder durch das Verwenden von *Protocol::WebSocket*) und nicht-blockierenden Sachen in Perl.

Wenn Du überlegst, welchen Server Du verwendet, prüfe, ob er die Entwürfe #75 und #76 (Version 00) unterstützt, TLS/SSL unterstützt und Cookies versteht. Wenn Du Dich dann immer noch nicht entscheiden kannst, nimm einen, der eine asynchrone DNS-Auflösung hat.

WebSocket Demo-Anwendungen

Konfiguration und Installation

Die Beispiele sind online auf meiner GitHub-Seite verfügbar und benötigen für gewöhnlich keine Installation von Modulen Dritter. Die einzige Ausnahme ist *JSON*, aber ich bin mir ziemlich sicher, dass es schon auf der Maschine von jedem Perl-Entwickler installiert ist.

Einfacher Echo-Server

Um einmal anzufangen und das Gefühl zu bekommen, dass etwas tatsächlich funktioniert, probieren wir einen einfachen Chatserver aus, der mit ReAnimator ausgeliefert wird. Wenn Du das git Repository klonst, findest Du ihn im `examples/` Verzeichnis (Du findest mehr Beispiele von WebSocket Echo-Servern in der *Protocol::WebSocket* Distribution oder auf GitHub unter *http://github.com/vti/protocol-websocket* im *examples/* Verzeichnis).

```
$ git clone \
  http://github.com/vti/reanimator.git
$ cd reanimator/
```

Starte nun den Chatserver (standardmäßig wird er auf 0.0.0.0:3000 lauschen):

```
$ perl examples/echo
```

Öffne die statische `index.html` Seite, die im `examples/public/` Verzeichnis liegt, in Deinem Browser.

Wenn Du eine `Disconnected` Nachricht bekommst, bedeutet das, dass entweder Dein Webbrowser keine WebSockets unterstützt oder es gibt ein Problem mit dem Server. Den Server kann man debuggen, in dem man die Umgebungsvariable `EVENT_REACTOR_DEBUG=1` setzt.

Beim Erfolg, bekommst Du ein `input` Feld und einen `submit` Button. Wenn Du etwas abschickst, solltest Du es wieder zurückbekommen. Es ist kein Fallbackmechanismus eingebaut, also benutze bitte einen Browser, der WebSockets nativ unterstützt.

Jetzt kommt eine Erklärung, wie alles funktioniert.

Was ist drin?

Im Inneren haben wir eine statische `index.html` Seite, die mit Hilfe von JavaScript mit dem WebSockets Server über die vorhin beschriebene API verbindet. Und dort ist ein `echo` Perlskript mit nur wenigen Zeilen Code:

```
use ReAnimator;

my $server = ReAnimator->new;

ReAnimator->new(
    on_accept => sub {
        my ($self, $client) = @_;

        $client->on_message(
            sub {
                my ($client, $message) = @_;

                $client->
                     send_message($message);
            }
        );
    }
)->listen->start;
```

Dieser Code sollte Dich an den JavaScript Code erinnern. Hier haben wir ein `on_accept` Hook, der gestartet wird, wenn eine neue Verbindung akzeptiert wird, der `on_message` Hook wird ausgeführt wenn eine neue Nachricht (bereits richtig geparst und nach UTF-8 dekodiert) eintrifft. Eine `send_message` Methode wird benutzt, um eine Nachricht an den Browser zu schicken. In diesem einfachen Beispiel senden wir die Daten einfach ohne eine Änderung zurück.

Um einige Konfusionen zu beseitigen, der `on_accept` Hook wird nicht aufgerufen, wenn eine normale Socket-Verbindung akzeptiert wird, sondern nach dem WebSocket Handshake. So kannst Du sicher sein, dass alles für die bidirektionale Übertragung vorbereitet ist. Ich habe es nicht `on_connect` genannt, weil der Leser dann denken könnte, dass sich der Server mit dem Browser verbindet, was gerade das Gegenteil ist.

Die echo-Anwendung kann natürlich nicht die ganzen Möglichkeiten von WebSockets zeigen, lass uns jetzt also zu ein paar wirklichen Anwendungen kommen.

Remote Shell

Natürlich ist der beste Weg, einen Server zu administrieren, über `ssh`. Du bekommst ein Terminal mit allen Features und Du bemerkst nicht einmal, dass Du remote arbeitest. Aber manchmal hast Du entweder keinen `ssh` Client auf Deiner Maschine installiert oder es gibt einen `ssh`-Zugang zu Deinem Server.

Die offensichtliche Lösung ist, einen Browser zu benutzen, weil er überall installiert ist und Du kannst Dich immer zu allen Servern verbinden. Das Problem ist, dass für ein Realtime Terminal in einem Browser einige fortgeschrittene Techniken benutzt werden müssen. Immer wenn Du einen Buchstaben eintippst, sollte er umgehend zum Server geschickt werden. Immer wenn der Server etwas zurückschickt, sollte es sofort angezeigt werden. Und das ist asynchron. HTML5 WebSockets passen hier perfekt.

Das Beispiel, das ich zeigen werde, ist ein wirkliches Realtime Terminal. Der Unterschied zu den meisten Lösungen ist, dass der Browser und ein "thin layer" zwischen Dir und der Remote Shell auf dem Server sind. Es versteht Farben, Terminalgrößen, funktioniert natürlich mit lang laufenden oder interaktiven Anwendungen. Genau so wie Du es von einem normalen `ssh` Client erwarten würdest.

Auf dem Server

Um diese Art von Interaktivität zu implementieren, müssen wir auf dem Server ein Pseudoterminal erzeugen, dass ein wirkliches Terminal emuliert und dabei besondere Terminalzeichen wie Farben parst und Kommandos wie `CTRL-C`, `CTRL-D` und so weiter versteht.

Für das Pseudoterminal benutzen wir *IO::Pty*, für die Terminalemulation und das Handling der Zeichen `Term::VT102`.

Die folgenden Schritte müssen unternommen werden, um ein Terminal zu erzeugen:

- Ein Kindprozess forken
- Erzeuge ein neues pty-Objekt und mache es zum slave eines normalen tty
- Öffne `STDIN`, `STDOUT` und `STDERR` neu, so dass sie auf unser pty-Objekt zeigen
- Führe ein Kommando wie `/bin/sh` aus, um eine Shell im neuen Terminal zu starten
- Registriere Callbacks, die ausgeführt werden wenn neue Daten eintreffen

Das ist alles auf der Serverseite. Der Einfachheit halber habe ich alles in eine Klasse gesteckt und es `Terminal.pm` genannt. Du findest den Quellcode auf GitHub unter *https://github.com/vti/showmetheshell/blob/master/lib/Terminal.pm*. Jetzt registrieren wir nur Deskriptoren in unserer Event-Loop. Mit ReAnimator könnte das so aussehen (vereinfacht), wie in Listing 3 dargestellt.

```
my $server = ReAnimator->new;

ReAnimator->new(
    on_accept => sub {
        my ($self, $client) = @_;

        my $terminal = Terminal->new(
            cmd            => '/bin/sh',
            on_row_changed => sub {
                my ($self, $row, $text) = @_;

                $client->send_message(
                    JSON->new->encode(
                        {type => 'row', row => $row, text => $text}
                    )
                );
            }
        );

        $self->event_reactor->add_atom($terminal);
        $terminal->start;
        $client->on_message(
            sub {
                my ($client, $message) = @_;

                ...
                # Send a pressed key to the terminal
                $terminal->key($code);
            }
        );
    }
)->listen->start;
```

Listing 3

Das ist alles. Immer wenn irgendetwas im Terminal passiert, wird es zum Browser geschickt und wenn etwas im Browser eingegeben wird, wird es zum Terminal geschickt.

Die Clientseite

Das größte Problem auf der Clientseite ist, alle Tastatureingaben zu bekommen. Das wird noch schwieriger wenn andere Browser ins Spiel kommen. Aber mit einem Modernen JS-Framework (z.B. `JQuery`) kann das größtenteils vereinfacht werden. Eine andere Lösung wäre, eine virtuelle Tastatur zu erzeugen.

Da der Server mitteilt, welche Zeile geändert wurde, gibt es keine Probleme mit der Anzeige per `JavaScript`. Ich habe 24 Zeilen mit `span` und eindeutigen IDs erzeugt und benutze diese Funktion:

```
function(n, data) {
    var row = $('#row' + n);
    row.html(data);
};
```

Wirklich, das ist alles was man braucht. Die WebSocket-Interaktion bleibt gleich.

Wichtige Verbesserungen

Eine offensichtliche Verbesserung ist natürlich, SSL zu benutzen. Du möchtest nicht eine `ssh` Session in einer ungeschützten Verbindung laufen lassen. Und da WebSockets SSL unterstützen, ist es einfach das Ersetzen von `ws://` durch `wss://` (natürlich muss der Server auch SSL unterstützen).

Demo und Sourcecode

Der komplette Sourcecode des Beispiels ist unter *http://github.com/vti/showmetheshell* verfügbar. Das Demovideo ist unter *http://vimeo.com/13681309* zu finden.

VNC

Wenn Du Dich jemals auf einen anderen PC verbinden und dabei den Desktop sehen wolltest, sowie Deine Tastatur und Maus benutzen wolltest (nicht via `ssh`), weißt Du vermutlich, was VNC ist. VNC ist ein wirklicher "thin client", der das RFB (Remote FrameBuffer) Protokoll verwendet.

Es ist eine Client-Server-Architektur. Alle Pixelberechnungen für das Display werden auf dem Server gemacht und der Client fragt einfach nur Änderungen am Framebuffer nach. Die Daten zwischen Client und Server werden mit verschiedenen Encodings ausgetauscht um den Traffic zu minimieren.

Es gibt viele VNC Server und Clients, einige Betriebssystem haben einen VNC Server schon standardmäßig dabei. So zum Beispiel Mac OS mit seinen Einstellungen zum Screensharing. Die meisten Linux-Distributionen bieten verschiedene Alternativen, die mit einem Kommando installiert werden können. Es gibt auch Open Source Varianten für Windows-Benutzer.

Aber was machst Du, wenn Du nur einen Browser und Internetverbindung hast? Es gibt Java Applets, die einen VNC-Server mitliefern, so dass Du auf die Remotemaschine zugreifen kannst. Aber das erfordert Java-Unterstützung, was manchmal ein Problem sein kann. Wir wollen aber eine native Lösung. Und diese Lösung sind WebSockets. Sie sind ideal für persistente Verbindungen, sie können Daten ohne Overhead austauschen und sie machen Spaß und sind einfach zu benutzen.

Um das RFB-Protokoll verwenden zu können, müssen wir es verstehen und in einem Browser anzeigen. Während RFB mit Pixeln arbeitet, können wir das HTML5 `canvas` verwenden, das eine extrem einfache API hat um Bilder zu zeichnen, Rechtecke zu kopieren und mehr. Schau Dir das folgende Beispiel an:

```
var canvas =
    document.getElementById('canvas');
var context = canvas.getContext('2d');

// Draw image
context.drawImage(img, x, y);

// Copy rectangle
var img =
   context.getImageData(x, y, width, height);
context.putImageData(img, x, y);
```

Wir können RFB-Nachrichten an den Browser weiterleiten ohne Veränderungen vorzunehmen, aber dann müssen wir die Nachrichten mit JavaScript parsen - was keine gute Idee ist, nicht nur weil das recht lange dauern kann, sondern weil es auch nicht einfach ist in JavaScript mit Binärdaten zu arbeiten. Wir können RFB-Nachrichten serverseitig parsen, in eine Art Text umwandeln und an den Browser ausliefern.

Die ganzen Benutzereingaben wie Tastendruck oder Mausbewegungen können ohne Probleme mit JavaScript gecaptured werden.

Bei Perl und CPAN gibt es nur `Net::VNC`, das mit VNC funktioniert, aber das captured nur Screenshots und ist für ande-

```
ReAnimator->new(
    on_accept => sub {
        my ($self, $client) = @_;

        my $vnc   = Protocol::RFB::Client->new(password => $VNC_PASSWORD);
        my $slave = $self->event_reactor->connect(
            address    => $VNC_ADDRESS,
            port       => $VNC_PORT,
            on_connect => sub {
                my $slave = shift;

                $slave->on_read(
                    sub {
                        my $slave = shift;
                        my $chunk = shift;
                        $vnc->parse($chunk);
                    }
                );
                $slave->on_disconnect(sub { $self->drop($client) });
            }
        );

        $vnc->on_handshake(
            sub {
                my $vnc = shift;
                $client->send_message(
                    JSON->new->encode({
                            type   => 's',
                            name   => $vnc->server_name,
                            width  => $WIDTH || $vnc->width,
                            height => $HEIGHT || $vnc->height
                    })
                );
            }
        );

        $vnc->on_framebuffer_update(
            sub {
                my ($vnc, $message) = @_;
                foreach my $rectangle (@{$message->rectangles}) {
                    $client->send_message(
                        JSON->new->encode( {type => 'fu', rectangle => $rectangle} )
                    );
                }
            }
        );

        $vnc->on_write(
            sub {
                my ($vnc, $chunk) = @_;
                $slave->write($chunk);
            }
        );

        $client->on_message(
            sub {
                my ($client, $message) = @_;

                my $json = JSON->new;
                eval { $message = $json->decode($message); };
                return if !$message || $@;

                if ($message->{type} eq 'fuq') {
                    ...
                }

                ...
            }
        );
    }
)->listen->start;
```

Listing 4

re Dinge nicht nützlich. Deshalb musst ich das Modul `Protocol::RFB` schreiben, das die meisten RFB-Nachrichten versteht, mit Tastatur und Maus umgehen kann und Hooks anbietet, die in asynchronen Anwendungen verwendet werden können.

Jetzt muss der WebSocket Server aber nicht nur auf Verbindungen vom Browser lauschen, sondern auch auf den VNC-Server auf einer anderen Adresse. Er braucht einen einfachen Mechanismus, um Nachrichten vom Client an den Server weiterzuleiten und umgekehrt.

Mit ReAnimator und EventReactor ist das ziemlich einfach (Code ist vereinfacht) - siehe Listing 4.

Hier laufen WebSocket und VNC-Verbindungen in derselben Event-Loop, sie haben nur ein unterschiedliches Level an Abstraktion. Während die WebSocket-Verbindung mit Textnachrichten arbeitet, arbeitet die VNC-Verbindung mit Binärdaten.

Die tatsächliche Geschwindigkeit des Renderings ist akzeptabel. Der Flaschenhals ist offensichtlich Perl selbst. Das könnte mit einer XS-Version optimiert werden, aber das ist mehr als genug für die Demonstrationszwecke.

Demo und Sourcecode

Der komplette Sourcecode dieses Beispiels ist unter http://github.com/vti/showmethedesktop verfügbar. Um die Anwendung laufen zu lassen, brauchst Du eine zweite Maschine mit einem VNC-Server. Das könnte z.B. eine Linux-Distributionen in einer Virtuellen Maschine sein. Das Demovideo ist unter http://vimeo.com/16459612 zu finden.

Fazit

HTML5 WebSockets sind eine mächtige Möglichkeit, um Realtime Anwendungen zu schreiben, die eine persistente Verbindung und bidirektionale Kommunikation erfordern. Die Zukunft des Webs ist da und Perl ist bereit, die Herausforderung anzunehmen.

Renée Bäcker

Neuerungen in Perl 5.14

Perl 5.14 ist zwar noch nicht erschienen, aber im April wird es soweit sein. Seit Perl 5.12.0 veröffentlicht wurde, waren die Perl 5 Porters und andere Entwickler sehr fleißig und haben einige Neuerungen in Perl 5.14 untergebracht. Die hier gezeigten Änderungen spiegeln den Stand von Perl 5.13.8 (Dezember 2010) wider und bringen Lust auf mehr.

Unicode-Unterstützung

Unicode ist eigentlich immer ein Thema - auch eines, das immer wichtiger wird. Seit einiger Zeit arbeitet vor allem Karl Williamson an der verbesserten Unicode-Unterstützung in Perl. Zum einen wurde auf die Version 6.0 von Unicode aktualisiert. Auch bei den Regulären Ausdrücken hat sich in Bezug auf Unicode einiges getan.

Mit der neuen Unicode-Version sind 2.088 neue Zeichen aufgenommen werden. Im Gegensatz zu früheren Perl-Versionen kennt `charnames` jetzt auch alle Zeichen aus der Unicode-Datenbank.

Unicode-Zeichen können einfach mittels `\N{name}` verwendet werden:

```
my $string = "I \N{HEAVY BLACK HEART} Perl";
```

An diesen `\N{...}`-Werten gab es auch Änderungen: So ist `\N{BELL}` jetzt als `deprecated` markiert, weil Unicode diesen Namen für ein anderes Zeichen verwendet. Außerdem können auch bestimmte Abkürzungen statt der langen Namen verwendet werden:

```
my $string = "\N{NBSP}";
# ist das gleiche wie
my $string = "\N{NO-BREAK SPACE}";
```

Für Reguläre Ausdrücke gibt es drei neue Flags: `u`, `d` und `l`.

Wird das Flag `l` verwendet, wird für die Regulären Ausdrücke die `locale`-Einstellung berücksichtigt. Bei dem folgenden Programm wird im ersten Fall nichts ausgegeben, weil bei `en_US.UTF-8` das "ä" nicht als Buchstabe vorkommt. Nachdem das locale auf `de_DE.ISO-8859-1` geändert wird, wird ausgegeben, dass der Reguläre Ausdruck matcht. Im ISO-8859-1 Zeichensatz ist das "ä" als Buchstabe enthalten.

```
use charnames qw(:full);
use POSIX qw(locale_h);

setlocale( LC_CTYPE, 'en_US.UTF-8' );

my $text = "ä";

{
    use locale;
    print $text =~ /(?l)\w/;
}

setlocale( LC_CTYPE, 'de_DE.ISO-8859-1' );

{
    use locale;
    print $text =~ /(?l)\w/;
}
```

Mit dem Flag `u` matcht z.B. `\w` alle Unicode-Zeichen, die als "Letter" bekannt sind.

```
my $text = "ä";
print $text =~ /(?u)\w/;
```

Möchte man das `u`-Flag für alle Regulären Ausdrücke in einem Scope einschalten, kann man das auch mit `use feature 'unicode_strings'` machen.

```
use feature 'unicode_strings';

my $text = "ä";
print $text =~ /\w/;
```

Das Flag `d` erzwingt das "alte" Verhalten von Perl - wie es vor Perl 5.14 war. Was `\w` bedeutet, hängt hier also vom String ab, der durchsucht werden soll. Also auch in solchen Fällen, in denen `use feature 'unicode_strings'` aktiviert ist.

```
use feature 'unicode_strings';

my $text = "ä";
print $text =~ /\w/;
print $text =~ /(?d)\w/;
```

Bei diesem Beispiel matcht der erste Reguläre Ausdruck, da durch das 'unicode_strings' die neue Unicode-Semantik für Reguläre Ausdrücke aktiviert wird. Der zweite Ausdruck matcht aber nicht, weil durch das `d`-Flag das alte Verhalten wieder eingeschaltet wird.

Zu beachten ist hierbei, dass die Regex-Engine **immer** die Unicode-Semantik nimmt, sobald für den String das UTF-8-Flag angeschaltet ist. Dann hat auch das `l`-Flag keine Auswirkung.

Karl Williamson war so fleißig, dass die Menge der Änderungen hier nicht reinpasst. Insgesamt wurde die Unicode-Unterstützung aber stark verbessert. Als weiterführende Literatur empfehle ich diese `perldoc`s:

- perlunitut
- perlunifaq
- perlunicode
- perluniintro
- Encode
- charnames

Moritz Lenz hatte für die 5. Ausgabe von $foo einen ausführlichen Artikel zu Charsets und Unicode geschrieben.

Reguläre Ausdrücke

Bei den Regulären Ausdrücken gibt es noch weitere Änderungen:

Wer mit stringifizierten Regulären Ausdrücken arbeitet, muss mit Perl 5.14 ein paar Änderungen vornehmen. Bisher war es so, dass ein stringifizierter RegEx mit `(?...)` begann. Das musste durch die Erweiterungen, die im Zuge der Entwicklungen hinzugekommen sind, angepasst werden. Jetzt beginnt der String mit `(?^...)`.

```
my $regex = qr/[0-9]/;
print "$regex";
```

Unter Perl 5.12 wird `(?-xism:[0-9])` ausgegeben, unter Perl 5.14 ist die Ausgabe `(?^:[0-9]`.

Die Modifikatoren der Regulären Ausdrücke können jetzt auch über `user re ...` für einen Scope "global" aktiviert werden, ohne sie bei jedem Regulären Ausdruck hinschreiben zu müssen.

```
my $regex = qr{ \A \d+ };
print "1: $regex\n";

{
    use re '/xms';
    my $regex = qr{ \A \d+ };
    print "2: $regex\n";
}
```

Liefert als Ausgabe:

```
1: (?^: \A \d+)
2: (?^xms: \A \d+)
```

Änderungen bei Operatoren und Funktionen

Ein sehr nützliches Feature ist die Erweiterung der Operatoren `s///` und `y///`. Bisher hatten diese Operatoren die als Rückgabewert entweder Erfolg/Misserfolg (1/`undef`) bzw. die Anzahl der Ersetzungen. Durch den neuen Modifikator `r` wird jetzt der veränderte Wert zurückgeliefert und der Variableninhalt bleibt erhalten. Bisher musste man Code so schreiben:

```
(my $neu = $alt) =~ s/alt/neu/;
my @neue_werte =
   map{ s/alt/neu/; $_ }@alte_werte;
```

Unter Perl 5.14 kann man das besser schreiben als:

```
my $neu = $alt =~ s/alt/neu/r;
my @neue_werte =
   map{ s/alt/neu/r }@alte_werte;
```

Funktionen, die mit Arrays und Hashes arbeiten können mit der neuen Perl-Version auch mit den entsprechenden Referenzen umgehen. Sollen Werte zu einer Arrayreferenz hinzugefügt werden, muss beim `push` die Arrayreferenz nicht mehr dereferenziert werden. Es funktioniert einfach

```
push $arrayref, $neuer_wert;
```

Die weiteren Funktionen, die jetzt auch Referenzen akzeptieren sind `push`, `shift`, `unshift`, `pop`, `splice`, `keys`, `values` und `each`.

Sonstiges

Unter Linux kann der Spezialvariablen `$0` (Name des Programms) ein neuer Wert zugewiesen werden. Dieser Wert wird dann auch bei Programmen wie `top` oder `ps` angezeigt.

```
$0 = 'mein perl programm';
```

In Perl kann man das unäre – auch auf Strings anwenden. Bisher war es so, dass bei einem "positiven" String ("ein test") einfach ein "-" vorangestellt wurde ("-ein test") und bei "negativen" Strings ("-ein test") wurde aus dem "-" ein "+" ("+ein test"). Das wurde auch gemacht, wenn der String eine Zahl war (also z.B. "1"). Bei Strings aus Zahlen hat sich das jetzt geändert und das Verhalten wurde so angepasst, dass es gleich zu dem Verhalten von normalen Zahlen ist:

```
my $var = - 'ein test';  # -ein test
my $var = - '-ein test'; # +ein test
my $var = - '1';         # -1

my $var = - '-1';        # alt: +1, neu: 1
```

Verwendet man mehrere `package`s in einer Datei muss man mit dem Scope aufpassen. Definiert man in einem `package` eine Variable, ist diese auch in den anderen `package`s in der Datei sichtbar. Möchte man das beschränken, musste man bisher

```
{
    package Foo;

    # weiterer Code
}
```

schreiben. Um den Code lesbarer zu halten, wurde hier die Möglichkeit geschaffen, das `package` vor dem Block zu definieren:

```
package Foo;
{
    # weiterer Code
}
```

Prototypen sind in Perl ein eigenes Thema. Bei Perl-Einsteigern sieht man häufiger die Verwendung von Prototypen, weil sie versuchen ihr Wissen von Methodensignaturen auf Perl zu übertragen. Dabei sind Prototypen in Perl etwas ganz anderes.

Mit Perl 5.14 gibt es ein paar Neuerungen in Bezug auf Prototypen. Methoden mit den Prototypen `(*)`, `(;$)` oder `(;*)` werden jetzt mit einer höheren Priorität versehen.

```
my $nr = 5;
sub foo(;$) { $_[0] * 2 };
print foo $nr < 3;
```

Wird jetzt als `print foo($nr) < 3` geparst.

Möchte man Funktionen schreiben, die sowohl Arrays/Hashes bzw. Referenzen auf diese Strukturen erlauben, kann man den `+`-Prototypen verwenden. Ohne diese Änderungen, müsste man sich vorher entscheiden, wie der Parameter aussehen muss.

```
use Data::Dumper;

sub test(\@)  { print Dumper \@_ }
sub test2(+@) { print Dumper \@_ }

my @array = ( 1 .. 3 );

test( @array );
test2( @array );
test2( \@array );
```

Bei der Subroutine `test` muss ein Array übergeben werden. Versucht man eine Arrayreferenz zu übergeben, bekommt man die Fehlermeldung

```
Type of arg 1 to main::test must be array
```

Der Subroutine `test2` hingegen kann man sowohl ein Array als auch eine Arrayreferenz übergeben und es kommt jeweils als Arrayreferenz in der Subroutine an.

Manchmal sieht man so etwas:

```
for my $var qw(1 2 3) {
    # ...
}
```

In solchen Fällen wurde das qw// ohne runde Klammern außenherum als `deprecated` markiert. Man muss den Code jetzt so schreiben:

```
for my $var (qw(1 2 3)) {
    # ...
}
```

Weitere Änderungen im Schnelldurchlauf: `given` hat jetzt einen Rückgabewert (man muss aber `do` verwenden); Am Errorhandling wurde einiges gemacht; Stashes sind immer definiert; Bei getieten Variablen wird bei `local` das `tie` aufgehoben.

In Perl gibt es auch Module, die als "deprecated" markiert sind, teilweise schon seit einigen Jahren. Ein paar dieser Module wurden jetzt aus dem Core entfernt. Damit Programmierer diese weiterhin nutzen können, müssen die Module vom CPAN installiert werden. Module, die entfernt wurden:

- Switch
- Class::ISA
- Pod::Plainer

Neben diesen Änderungen wurden noch sehr viele Module aktualisiert, deren Änderungen hier zu weit führen würden. Auch am Core wurden noch weitere Änderungen durchgeführt, die ich hier nicht alle aufzeigen kann. Viele haben keine Änderungen für den Programmierer zur Folge, sondern sind Optimierungen in Bezug auf Geschwindigkeit oder Speicherverbrauch.

Herbert Breunung

WxPerl Tutorial - Teil 6: Einfache App

Seit Jahrzehnten verstehen Entwickler unter Applikationen Programme für Endanwender mit graphischer Oberfläche. Diese Folge widmet sich den Teilen, die bisher dazu fehlten: Menü, Werkzeugleiste, Statuszeilen sowie Tastatur und Mauseingaben. Auch zeitgesteuerte Ereignisse werden vorgestellt. Einige andere wichtige Zutaten wie Splashscreen, Icon und Titelleiste wurden bereits in Folge 5 erläutert, Reiterleisten eine davor.

Die offizielle WxPerl-Wiki ist nach *http://wxperl.info/* umgezogen.

Eine Statuszeile

Dabei beginnen wir denkbar einfach, indem wir folgende Zeile in das bereits mehrmals gelistete "Hallo Erde"-Beispiel einfügen:

```
my $bar = $frame->CreateStatusBar(2);
```

Das ist alle Male kürzer als die folgenden 3 bedeutungsgleichen Zeilen:

```
my $bar = Wx::StatusBar->new($frame, -1);
$bar->SetFieldsCount(2);
$frame->SetStatusBar($bar);
```

Es erzeugt eine sofort sichtbare Statuszeile mit 2 gleichgroßen Feldern. Für mehr Individualität sorgt:

```
$bar->SetStatusWidths(200,-1);
```

Das erste, linke Feld wird 200 Pixel breit, das zweite bekommt den Rest. Die Größe des Fensters spielt keine Rolle bei dieser linksbündigen Anzeige. Wer proportionale Breiten haben möchte, muss sich aus einem `Panel`, `ToolBar` oder ähnlichem etwas selber bauen. Da Statuszeilen nur Textnachrichten anzeigen und keine Bilder oder andere Widget beinhalten können, (Tk ist an dieser Stelle wesentlich mächtiger) erfordern auch solche Wünsche eine Sonderanfertigung. Die braucht man schließlich auch bei `MiniFrame`'s oder `Dialog`en, da Statuszeilen, wie auch Menüzeilen, eigentlich nur für einen `Frame` gedacht sind, der darauf optimiert ist, das Hauptfenster einer App zu spielen. Einziger Gestaltungsspielraum ist der Stil der Zellen der einzeln wie die Breite vergeben wird.

```
$bar->SetStatusStyles
(wxSB_NORMAL, wxSB_FLAT, wxSB_RAISED);

$bar->SetStatusText($nachricht, $nr);
```

Auch das versenden der Nachrichten an die Statusfelder ist denkbar einfach. Ich empfehle nur dringend, das von einer zentralen Routine aus zu tun.

```
sub statusmeldung {
    my $nachricht = shift;
    $bar->SetStatusText($nachricht, 1)
          if length $nachricht > 2;
}
```

So lässt sich prüfen, ob die Nachricht ein gefordertes Format hat und sie landet auch nicht aus Versehen im falschen Feld. Das erste, linke Feld mit der Nummer 0 würde ich hier den Kurzhilfemeldungen des später hinzukommenden Menüs überlassen. Das sind die bekannten, schnell wechselnden Statusnachrichten, die beim Blättern im Menü erscheinen. Sie erscheinen standardmäßig im Feld 0, was sich aber auch mit

```
$bar->SetStatusBarPane($nr);
```

ändern lässt. Eine -1 an der Stelle stellt die Hilfetexte im Status ab. Vielleicht mag man die Kurzhilfen woanders, z.B. in einer selbst gestalteten Textblase auftauchen lassen oder man könnte sie auch nur für kurze Zeit in einem Statusfeld einblenden, was auch nicht dem Standardverhalten entspricht. Jedes Feld besitzt nämlich einen Stack für Textnachrichten. Wenn die neue Nachricht mit:

```
$bar->PushStatusText($nachricht, $nr);
```

ankommt, kann hinterher

```
$bar->PopStatusText($nr);
```

den vorherigen Zustand wiederherstellen. Bleibt nur das Problem der Zeitsteuerung.

Timer Events

Zeitsteuerung geschieht in Wx mit einem Event wie in Folge 2 vorgestellt.

```
Wx::Event::EVT_TIMER( $frame,$TimerID,sub {
    $bar->PopStatusText($nr);
} );
```

Damit wird quasi das Ohr geschaffen, welches danach lauscht, ob hier ein Wecker klingelt. Der erste Parameter ist das Widget der Objekthierarchie, welches das Ohr angepflanzt bekommt, also den Auftrag erhält, auf Events dieser Art zu reagieren, wenn ihm der Wx-schedular Arbeitszeit zuweist. Es muss eine von `Wx::EvtHandler` abgeleitete Klasse sein. Denn auch bei der Erzeugung des Timers wird ein Widget als erster Parameter (meist der Frame) angegeben, der bestimmt wo der Wecker klingelt. Wie bei bei jedem anderen Event so gilt auch hier: hat dieses Widget keinen EventHandler für dieses Ereignis, wird es zu den Eltern deligiert u.s.w. Der zweite Parameter ist eine Timer ID, die konsistent mit anderen Timern, aber ansonst beliebig verteilt werden kann, da diese ID einen eigenen Namensraum haben, der sich nicht mit Widget- oder Menüeintrag-ID überschneidet. Der dritte Parameter ist der Callback, die auszuführende Codereferenz. Wer den Aufwand gering halten möchte und sich keine *TimerID* ausdenken mag, kann bei der Erzeugung statt:

```
my $timer = Wx::Timer->new($frame,$TimerID);
my $timer = Wx::Timer->new($frame,-1);
```

die zweite Variante nehmen und die garantiert unbenutzte ID dann später mit

```
$timer->GetId
```

abfragen. Nützlich ist auch die Methode `IsRunning`, aber vorher muss nur noch der Wecker gestellt werden:

```
$timer->Start(
   $millisekunden, $einmal_auslösen );
```

Hat der zweite Parameter hier keinen positiven Wert wird der Alarm periodisch, ab sofort, alle `$millisekunden` ausgelöst. Für ein einmaliges Signal setzen wir `$einmal_auslösen` auf 1, was aber nicht bemerkt wird, da `$bar->PopStatusText($nr);` bei einem leeren Textstack nichts tut und die letzte sichtbare Nachricht belässt. Die expliziten Konstanten sind hierfür `wxTIMER_CONTINUOUS` und `wxTIMER_ONE_SHOT`, aber 1 und 0 sind hier deutlich genug. Wenn der Timer periodisch arbeitet, kann er natürlich mit `$timer->Stop;` beruhigt werden. Ein neuerlicher Startist jederzeit möglich. Mit einfachem `$timer->Start;` wird periodisch mit dem letztbenutzten Zeitwert gefeuert

Werkzeugleiste

Da Statuszeilen in Wx sehr beschränkt sind, könnte man anstatt ihrer wirklich Werkzeugleisten nehmen, da sie sich wie Widgets mit Sizern (siehe Folge 3) überall platzieren lassen, auch am untere Ende eines Frames. Und sie können viele Widgets wie Texte, Bilder, Regler und Comboboxen (Auswahl mit ausklappbarem Menü) in sich aufnehmen.

Standardgemäß sind es aber Knopfleisten, gleich unter der Menüleiste. Diese Knöpfe sind jedoch keine `Wx::Button` oder gar `BitmapButton` sondern von `ToolBarToolBase` abgeleitete Klassen, auf die man nur per `Wx::ToolBar` zugreift.

Ähnlich der Statuszeile, kommt der Standard per:

```
my $bar = $frame->CreateToolBar();
```

Die so erhaltene Werkzeugleiste erwartet 16x16 Pixel große Icons, ist waagerecht und behält ihre Position ohne Sizer. Um diese Eigenschaften etwas zu ändern, gibt es Stilkonstanten (mit *TB* für Toolbar) für den ersten Parameter. Der zweite wäre die Widget-ID. Interessant ist zum Beispiel `wxTB_VERTICAL` (für senkrechte Werkzeugleisten) als Gegenstück zum eigentlich überflüssigen `wxTB_HORIZONTAL`. Ein `wxTB_BOTTOM` würde (als Gegenstück zum ebenso seltenem `&Wx::wxTB_TOP` - gibt es auch als `wxTB_LEFT` und `wxTB_RIGHT`) die Leiste an den unteren Fensterrand, aber über die Statuszeile legen. Da ein `Frame` jedoch nur eine einzige "offizielle" Werkzeugleiste hat, kann man auf dem Weg nicht eine Werkzeugleiste und noch eine reichhaltige Statuszeile bekommen.

Werkzeugleiste mit Textknöpfen

Beim Aufbau einer App hat nicht jeder sofort die Icons für die `ToolBar` zu Hand, die er benötigt. Ist auch nicht notwendig. Entweder kann man auf die Standardicons des Betriebsystems mit `Wx::ArtProvider` zurückgreifen (siehe Folge 4), oder man setzt Worte an an Stelle der Bilder. Dazu wird die Leiste vorbereitet:

```
my $bar = $frame->CreateToolBar(
    &Wx::wxTB_TEXT|&Wx::wxTB_NOICONS
);
```

Nun kann sie mit Knöpfen gefüllt werden.

```
my $tool = $bar->AddTool(
    $tool_id, 'Close',
    &Wx::wxNullBitmap,
    &Wx::wxITEM_NORMAL, 'Tip'
);
```

Die `$tool_id` bewegt sich im gleichen Namensraum mit den Menüeinträgen und Widget-ID. Also wieder am besten mit -1 generieren lassen oder sich aus dem bereits vergebenen und dem vordefiniertem Bereich (`wxID_LOWEST` .. `wxID_HIGHEST`) raushalten. "Close" ist im Beispiel der Schriftzug (Label) and "Tip" erscheint nur dann, wenn der Mauscursor einige Zeit still über dem Knopf liegt. Die `wxNullBitmap` ist lediglich der Platzhalter für das nichtvorhandene Bild und `wxITEM_NORMAL` eigentlich überflüssig, wenn es keinen Tooltip gibt. Es gibt auch die Form

```
my $tool = $bar->AddTool(
    $tool_id, 'Close',
    &Wx::wxNullBitmap,
    &Wx::wxNullBitmap,
    &Wx::wxITEM_NORMAL,
    'Tip', 'Hilfe'
);
```

die 2 Funktionalitäten mehr bietet und bei allen Knopftypen (nächster Absatz) funktioniert Die Kurzhilfe ("Hilfe") erscheint wie besprochen im *StatusBarPane*. Sind alle Knöpfe erzeugt, folgt ein:

```
$bar->Realize();
```

das sich auch nach jedem Hinzufügen eines Werkzeugs zur Laufzeit empfiehlt. Mit dieser letzten Zeile verknüpfen wir den Knopf mit einem Funktionsaufruf.

```
Wx::Event::EVT_TOOL (
    $frame, $tool->GetId,
    sub{ $frame->Close; }
);
```

Kippschalter erzeugt `AddCheckTool`. Hier gilt nur die Form mit 2 `wxNullBitmap`-Platzhaltern. Das gilt auch für `AddRadioTool`, mit dem man Kippschaltern erzeugt, von denen nur einer "eingedrückt" sein kann (eine Form von Radio-Button). Möchte man mehrere solcher Gruppen, so reicht es sie bei der Erzeugung mit `AddSeparator` zu trennen, was einen Querstrich oder zusätzlichen Platz einfügt (je nach Betriebssystem). Weitere Widgets fügt hinzu:

```
$bar->AddControl($widget);
```

Zu allen Add...-Methoden gibt es natürlich ein Pendant mit Insert..., dessen erster Parameter die Einfügeposition ist.

Werkzeugleiste mit Bildern

Um die Leiste zu bebildern, fallen als erstes die beiden Stilkonstanten weg. Als Folge dessen werden keine Label mehr angezeigt, müssen aber weiterhin angegeben werden (" reicht), da Parameter positional sind. Die leeren Knöpfe sind aber immer noch am ToolTip und Hilfe unterscheidbar, was zu internen Testzwecken reichen kann.

Im nächsten Schritt fügen wir `Wx::InitAllImageHandlers();` zu, sonst kommt es zu wenig beliebten Warnfenstern und weiterhin leeren Knöpfen.

Die `wxNullBitmap` mit einer kontrastreichen Pixmap auszutauschen ist auch sehr einfach, da der Konstruktor dieser Klasse nur einen Dateinamen braucht und als zweiten Parameter auch ein `wxBITMAP_TYPE_ANY` akzeptiert um den Bildtyp selbst herauszufinden. `Wx::Bitmap` geht sogar soweit, dass es im Falle einer JPG- oder PNG-Datei `Wx::Image` bemüht, die das Bild lädt, konvertiert und an `Wx::Bitmap` übergibt. Bitmaps sind zweidimensionale Matrizen (Tabellen) aus Pixelfarben, was auf die Formate **.bmp*, **.xpm*, **.xbm*, **.pict* und **.gif* zutrifft. `wxBITMAP_TYPE_XPM` bestehen sogar aus reinem C-code, wodurch man sie theoretisch auch mit einem Texteditor "malen" kann. **.png*, **.jpg*, **.pcx*, **.pnm* und **.tif* speichern die Farbinformationen kompakter und **.jpg* auch mit mehr oder weniger sichtbaren Verlusten.

Für diese Formate hat Wx die Klasse `Wx::Image`, die auch ein kleines GD ist, eine Art Erste-Hilfe-Kasten für einfache Bildbearbeitung. Das wird im nächsten Absatz wichtig, wenn ich zeige, wie die `Toolbar` große Icons bekommt, da nur `Wx::Image` die Icons skalieren (strecken) kann.

Auch wenn das Laden der Datei simpel ist, so kapsel ich es zumindest in eine eigene Routine.

```
sub bitmap {
    require File::Spec;
    my $file = File::Spec->catfile
                        ('icons/dir', shift);
    Wx::Bitmap->new(
        $file, &Wx::wxBITMAP_TYPE_ANY
    ) if -e $file;
}
```

Das lässt die übernächste Zeile kürzer und robuster werden. Und der Umstand, dass das Iconsverzeichnis nur an einem Ort im Code steht, macht ihn wesentlich wartbarer. In den Leitlinien für guten Stil heisst das DRY (don't repeat yourself - aber auch kurz: trocken - als Gegenteil von (über-)flüssig). Auch die Umstellung der Leiste auf große Icon's (kleine mittendrin sehen nicht gut aus) bedarf so nur einer weiteren Zeile.

```
my $bmp = Wx::Bitmap->new(...);
Wx::Bitmap->new(
    Wx::Image->new($bmp)->Scale(32, 32)
);
```

Das Ergebnis dieser Aktion ist leider optisch suboptimal, da die Pixel nur größer werden. Geschickter wäre es, größere Icons zu verwenden. Entweder zwischen 2 verschiedenen Größen wählen, oder nur Größere nehmen und dann optional runterskalieren. Bei gemischten Beständen hilft ein `$bmp->GetHeight` oder `GetWidth` die Größe des geladenen Bitmaps zu erkennen. Wurde es schon eingefügt, hilft `$bar->GetToolBitmapSize($id)` dessen Ergebnis ein Size-Objekt ist, das auch `GetHeight` und `GetWidth` versteht.

Das Erstellen der Knöpfe geht in jedem Fall:

```
my $tool = $bar->AddTool(
    -1, '', bitmap('close.xpm'),
    &Wx::wxNullBitmap,
    &Wx::wxITEM_NORMAL,
    'Tip', 'Hilfe'
);
```

Bei den *Check-* und *RadioTool*'s ändert sich nichts. Auch mit Icon reagieren die Knöpfe graphisch genau wie vorher. Mit `ToggleTool($tool_id, $bool)` kann auch der Pragrammierer die Kippschalter (simuliert) drücken. `$bool` muss einfach positiv sein, oder negativ (0 oder '') um den Knopf wieder auszubeulen. Ähnlich funktioniert auch die Methode `EnableTool` um Knöpfe zu deaktivieren. Jedoch Vorsicht ! Das Eingrauen funktioniert nur, wenn brav `wxNullBitmap` als zweites Icon des Tools gewählt wurde. Denn sonst wird einfach das andere Icon eingefügt und der Knopf funktioniert trotzdem nicht, was Benutzer verwirren kann, wenn die Graphik nicht eindeutig ist.

Aber besonders bei `RadioTool`'s bietet es sich an, mehrere Knöpfe mit einem Callback zu verknüpfen. Auch wenn Wx die ID's fortlaufend vergiebt, lohnt es sich hier vielleicht das selbst zu tun.

```
Wx::Event::EVT_TOOL_RANGE
 ($frame, 2001, 2003, sub {
    my ($bar, $event) = @_;
 });
```

Um herauszubekommen welcher Knopf zwischen 2001 und 2003 gedrückt wurde, genügt ein `$event->GetId`. Wem das nichts sagt, kann mit `$bar->GetToolShortHelp($id)` den ToolTip oder mit `GetToolLongHelp` den Hilfetext des Tool's anfordern.

```
$bar->SetStatusText(
  $toolbar->GetToolShortHelp(
     $event->GetId), 1);
```

Beide Ereignisse reagieren nur aus das Drücken der linken Maustaste. Soll es mal mit rechten Dingen zugehen, nimmt man `EVT_TOOL_RCLICKED` und `EVT_TOOL_RCLICKED_RANGE`.

Ich hätte auch gern noch diese Knöpfe mit dem kleinen schwarzen Dreieck behandelt, die nach unten ein Menü öffnen. Dazu reicht beim `AddTool` statt `wxITEM_NORMAL` `wxITEM_DROPDOWN` zu sagen und das Ereignis mit `EVT_TOOL_DROPDOWN` abzufangen. Wie Menüs und Kontextmenüs gebaut werden beschreiben die nächsten Absätze. Allerdings hat hier Wx derzeit einen Bug weswegen ich kein Beispiel zeigen kann.

Menüs im Überblick

Viel vom soeben beschriebenen lässt sich auch auf Menüs übertragen. Der Aufbau ähnelt sich sehr, nur dass hier Menüs weitere Menüs als Items (Menüpunkt) haben können. `EVT_MENU` ist gar nur ein Synonym für `EVT_TOOL`. So können sich auch ein *Item* und ein *Tool* die selbe ID teilen um den gleichen Aufruf zu bewirken. Viele der bekannten Applikationsfunktionen haben dazu auch schon eine vordefinierte ID wie zum Beispiel: `wxID_OPEN`, `wxID_Paste` oder `wxID_ABOUT`.

Die vollständige Liste steht in der WxPerlTafel (*http://wiki.perl-community.de/Wissensbasis/WxPerlTafel*) oder im Modul `Wx::Wx_Exp.pm`. Oder in der offiziellen Dokumentation unter *http://docs.wxwidgets.org/trunk/page_stockitems.html*, wo auch jedes Icon gezeigt wird, das mit einer solchen ID in Verbindung steht und bei Menüerzeugung automatisch über den ArtProvider bezogen wird.

Auch hier gibt es `wxITEM_NORMAL`, `wxITEM_CHECK`, `wxITEM_RADIO` und `wxITEM_SEPARATOR`, wobei `ToggleTool` hier sinnigerweise `Check` heißt. Anstatt `Add...` wird nun `Append...` geschrieben und es gibt auch ein `Prepend...`. Wirklich neu hinzu kommt nur die äusserst selten verwendete Methode `$menu->Break`, die das untere Ende eines Menüs markiert und rechts daneben fortfahren lässt.

Doch die Grundidee ist simpel. Es gibt `Wx::MenuItem` die zusammen `Wx::Menu` bilden, die an einer `Wx::MenuBar` aufgereiht werden. Nur ein `$frame->CreateMenuBar();` gibt es nicht. Es heißt:

```
my $bar = Wx::MenuBar->new();
$bar->Append($edit_menu, 'Bearbeiten');
$bar->Insert(0, $file_menu, 'Datei');
$frame->SetMenuBar($bar);
```

Dabei ist nicht wichtig ob die Menüleiste eben frisch erzeugt wurde oder schon vollständig ist. Hier sind Änderungen sofort sichtbar, auch wenn es ein `$bar->Refresh` gibt. Auch Menüs brauchen kein `Realize`, da sie mit jedem Aufruf einen neuer Renderbefehl bekommen. Jedoch empfehle ich hier nicht den bequemen Weg zu gehen und mit der Appendmethode der `Menu`-Klasse oder Ähnlichem das Item generieren zu lassen. In kleinen Programmen spart das einige Zeilen Code, aber bei Sonderwünschen wie eigenen Icons oder farbigem Text (nur unter Windows) merkt man das es hier feine Unterschiede gibt. Diese Dinge lassen sich nicht während der Erzeugung angeben und auch nicht nachträglich ändern. Dazu erzeuge ich ein `Wx::MenuItem`-Objekt und füge dieses nur mit Append in das Menü. Bei anspruchsvollen Lösungen sollte man sich diese überflüssige Arbeit eh automatisieren. Entweder mit Eric Wilhelm's `WxPerl::MenuMaker`, der auch Werkzeugleisten aus Datensätzen erstellt, oder mit etwas wie Kephra's `Kephra::Menu`. (siehe *http://kephra.svn.sourceforge.net/viewvc/kephra/dev/base/lib/Kephra/Menu.pm?revision=663&view=markup*)

Aufbauhilfen

Eric hat neben WxPerl-Dokumentation in der Wiki einige nützliche Module geschaffen.

`WxPerl::MenuMaker` wandelt verschachtelte Strukturen in fertige Menüs oder Menüleisten um.

```
{
    name      => 'file_open'
    icon      => 'kephra.ico',
    tooltip   => 'Öffnen',
    longhelp  => 'Was ich schon ...'
    associate => 'file_open',
    menu      => [ .... ]
},
```

Die kann ein Eintrag aus einer Liste sein, in der wiederum Menülisten enthalten sein können. So etwas lagert man in Konfigurationsdateien aus die mit einem YAML-Befehl oder ähnlichem in einer Variable landen. Und nach einem:

```
my $mm = WxPerl::MenuMaker->new(
    handler => $frame,
    nomethod =>
      sub {warn "$_[1] cannot '$_[0]()'"},
);
$mm->create_menubar(\@menu);
```

steht das Hauptmenü.

`WxPerl::MenuMap` wurde geschaffen um Menüstrukturen, egal wie erzeugt, einfacher anzufassen. Die werden in einen Hash gemappt wobei 'Datei_Öffen' der Eintrag Öffnen im Menü mit dem Label Öffnen wäre (englisch 'file_open'). Nach einem

```
my $menumap =
  WxPerl::MenuMap->new($menu_object);
```

Ist `$menumap->file_open` eine Referenz auf das gewünschte Item-Objekt.

Sehr nützlich ist auch `wxPerl::Constructors`, das die Erzeugung der wichtigsten Widgets mit benannten Parametern erlaubt.

```
wxPerl::Frame->new(
  $parent,
  $title,
  id       => -1,
  position => Wx::wxDefaultPosition(),
  size     => Wx::wxDefaultSize(),
  style    => Wx::wxDEFAULT_FRAME_STYLE(),
  name     => wxFrameNameStr,
);
```

Und schließlich `wxPerl::Styles` und `WxPerl::ShortCuts` helfen beim Umgang mit den vielen und langen Wx-Konstanten.

Tastatureingaben

Aber Lehrjahre sind keine Herrenjahre, einmal gehen wir den langen Weg. Vor allem um die Tastatursteuerung eines einfachen Programmes zu zeigen.

```
my $open = Wx::MenuItem->new(
    $menu, $ID++, "Label\tCtrl+L",'Hilfe'
);
# ginge auch mit Ctrl-L
$open->SetBitmap( bitmap('file-edit') );
$menu->Append($open);
```

Beim Parsen des Labels sieht sich Wx berechtigt daraus gleich einen Key-Event zu stricken, was in den meisten Fällen sehr nützlich ist. Aber er versteht nur die englischen Tastenkürzel und was ist, wenn innerhalb verschiedener Widgets eine Tastenkombination verschiedene Belegungen haben soll, aber trotzdem alle diese Funktionen im Menü verzeichnet sind und dort auch nachschlagbar sein sollen.

Obwohl ich weiß, dass die Information zum Key-Binding in einem Objekt namens `Wx::AcceleratorTable` landet, gelang es mir mangels API bisher nicht, diese zu löschen. Deswegen füge ich am Ende des Labels kaum sichtbare Zeichen wie ´ oder ` ein, um den Parser zu stören. Sicher ein Hack, aber wirksam. Um alternativ eine Tastatursteuerung aufzubauen sollte man das Ereignis `EVT_KEY_DOWN` abfangen. Es gibt zwar auch `EVT_KEY_UP`, aber das wäre wohl ungewohnt.

```
Wx::Event::EVT_KEY_DOWN( $frame, sub {
    my ($widget, $event) = @_;
    my $key = $event->GetKeyCode;
    next_page()
        if $key eq &Wx::WXK_TAB
            and $event->ShiftDown
            and $event->AltDown;
    my $chr = chr $key;
    $frame->Close
        if $chr eq 'C'
            and $event->AltDown;
    $event->Skip;
} );
```

Die erhaltenen Tastencodes entsprechen den ASCII-Codes der Grossbuchstaben. In allen anderen Fällen sollte man die Tastaturkonstanten mit langem Namen verwenden, die ebenfalls in der WxPerlTafel gelistet sind. Für die 4 wichtigen Umschalttasten (die im Beispiel verwendeten und `MetaDown` - auch an Mac-user wird gedacht) gibt es gesonderte Methoden, was allemal einfacher ist als die Werte aus der von `GetModifiers` gelieferten Bitmaske zu popeln.

Mauseingaben

Die Namen der Mausevents folgen dem gleichen `..._UP`- und `.._DOWN`-Muster für einfache Klickbewegung und `..._DCLICK` für Doppelklicks. Jedoch fehlt mir dort das Wort "Mouse". Stattdessen besteht der Name neben dem konventionellen `EVT_...` und lediglich aus der Maustaste: `_LEFT_`, `_MIDDLE_` und `_RIGHT_`, sowie `..._UP-` oder `.._DOWN`. `EVT_MOUSEWHEEL` reagiert auf das Mausrad und auch einfache Mausbewegungen über einem Widget lassen ich mit `EVT_ENTER_WINDOW`, `EVT_LEAVE_WINDOW` und `EVT_MOTION` belauschen. Wobei letzteres wohlbegründet sein will, weil es 100 mal je Sekunde ausgelöst wird, sofern sich der Cursor über einem Widget bewegt. Wer wissen will ob der Benutzter irgendwas mit der Maus macht fragt `EVT_MOUSE_EVENTS`.

Da sich alle Mouseevents immer auf ein Widget beziehen, sind die Mauskoordinaten, die das Event kennt, immer relativ auf die linke, obere Ecke des Widgets bezogen. Das ist aber meist was man braucht und auch um Kontextmenüs zu öffnen perfekt. Kontextmenüs sind normale Menüs und es lassen sich auch Teile des Hauptmenüs dazu wiederverwenden, sofern man harmlose GTK-Warnungen ignorieren kann.

```
Wx::Event::EVT_RIGHT_DOWN($widget, sub {
    my ($widget, $event) = @_;
    $widget->PopupMenu(
        $menu, $event->GetX, $event->GetY
    );
});
```

So lassen sich auch an die anfängliche Statuszeile Menüs knüpfen um die immer beliebter werdenden Statuszeilen zu schaffen, an denen man das Angezeigte auch ändern kann. Letzte Hürde dazu wäre nur noch herauszufinden auf welche geklickt wurde. Die Breiten hat man selbst bestimmt, aber die Grenzen lassen sich auch mit `$bar->GetFieldRect($index)->GetTopLeft->y` abfragen.

Vorschau

Dies sollte für dieses mal reichen, aber zu einer guten App gehört noch einiges mehr. Die Zwischenablage, Drag'n Drop, Mauscursor, Logging und vielleicht Drucken sollen aber Thema der nächsten Folge sein.

Renée Bäcker

Moose Tutorial - Teil 3: Vererbung

Nachdem in den letzten beiden Ausgaben die Themen "Attribute" und "Methoden" behandelt wurden, ist in dieser Ausgabe das Thema "Vererbung" an der Reihe. In der nächsten Ausgabe werden dann "Rollen" besprochen. Einige Programmierer bevorzugen "Rollen" gegenüber der "Vererbung". Beides hat seine Daseinsberechtigung.

Vererbung im Allgemeinen

Aus Wikipedia:
"Die Vererbung (engl. Inheritance) ist eines der grundlegenden Konzepte der Objektorientierung und hat große Bedeutung in der Softwareentwicklung. Die Vererbung dient dazu, aufbauend auf existierenden Klassen neue zu schaffen, wobei die Beziehung zwischen ursprünglicher und neuer Klasse dauerhaft ist. Eine neue Klasse kann dabei eine Erweiterung oder eine Einschränkung der ursprünglichen Klasse sein. Neben diesem konstruktiven Aspekt dient Vererbung auch der Dokumentation von Ähnlichkeiten zwischen Klassen, was insbesondere in den frühen Phasen des Softwareentwurfs von Bedeutung ist. Auf der Vererbung basierende Klassenhierarchien spiegeln strukturelle und verhaltensbezogene Ähnlichkeiten der Klassen wider."

In Perl 5 gibt es mehrere Wege, wie Klassen von einer anderen Klasse erben können. Es gibt die Pragmas `base` und `parent` und man kann direkt `@ISA` verändern.

```
use base 'Basisklasse';
# oder
use parent 'Basisklasse';
```

`parent` ist erst seit Perl 5.10.1 im Core enthalten, ist aber über CPAN auch für frühere Perl-Versionen verfügbar.

Die beiden ersten Möglichkeiten sind der letzten Möglichkeit vorzuziehen. Aber das ist nicht Thema dieses Tutorials.

In diesem Teil des Tutorials bleiben wir als Beispiel bei den Zeitschriften aus der letzten Ausgabe. Als oberste Klasse benutzen wir die Klasse "Publikation". Da es aber verschiedene Arten von Publikationen gibt, werden in der Basisklasse nur die Dinge definiert, die bei allen Publikationsformen gleich sind.

```
package Publikation;

use Moose;

has published => ( is => 'ro' );
has pages     => ( is => 'ro' );
has title     => ( is => 'ro' );
has available => ( is => 'ro' );

has format    => (
    is      => 'ro',
    default => 'A4',
);

sub info {
    my ($self) = @_;

    return sprintf "%s has %s pages",
        $self->title, $self->pages;
}

sub is_a {
    my ($self) = @_;

    return sprintf "%s isa %s",
        $self->title, ref( $self );
}

no Moose;

1;
```

Publikationsformen sind z.B. "Buch", "Zeitschrift" und "Paper". Damit haben wir auch schon drei Klassen, die von der Klasse "Publikation" erben. Die Vererbung stellt in der Regel eine "ist-ein"-Beziehung dar: Das Buch ist eine Publikation; die Zeitschrift ist eine Publikation.

Vererbung in Moose

Vererbung ist ein wesentlicher Aspekt in der Objektorientierung. Und so wird das auch von `Moose` unterstützt. Auf die verschiedenen Möglichkeiten, die `Moose` bei der Vererbung bietet, wird in den nächsten Abschnitten eingegangen.

In `Moose` wird für die Vererbung das Schlüsselwort `extends` verwendet. Das drückt schon aus, dass die abgeleitete Klasse die Basisklasse erweitert.

Wie oben erwähnt, ist die Zeitschrift eine besondere Form der Publikation. Sie ist also eine Subklasse.

```
package Zeitschrift;

use Moose;

extends 'Publikation';

sub print_type {
    my ($self) = @_;

    print $self->is_a;
}

no Moose;

1;
```

Ein Objekt der Klasse Zeitschrift hat jetzt alle Attribute und alle Methoden von Publikation zur Verfügung.

```
use Zeitschrift;

my $obj = Zeitschrift->new(
    published => '01.02.2011',
    pages     => 64,
    title     => '$foo - Perl-Magazin',
);

$obj->print_type;
```

Mehrfachvererbung

Jetzt stehen wir vor der Situation, dass eine Zeitschrift in verschiedenen Formaten veröffentlicht werden kann: Im PDF-Format, als epub, usw. Die Methoden zur Umwandlung in PDF werden in der Klasse "PDF" definiert. Jetzt schaffen wir also eine Klasse "PDFZeitschrift", die "ist eine" Zeitschrift und "ist ein" PDF. Also muss die neue Klasse von zwei Basisklassen erben.

Dafür wird die so genannte Mehrfachvererbung eingesetzt. Perl kann von Haus aus schon mit Mehrfachvererbung umgehen. Sprachen wie z.B. Java erlauben nur die Einfachvererbung.

Bei Moose sieht es dann so aus:

```
package PDFZeitschrift;

use Moose;

extends 'Zeitschrift', 'PDF';

no Moose;

1;
```

Mehr ist nicht erforderlich und schon kann man eine PDF-Zeitschrift erstellen:

```
use PDFZeitschrift;

my $pdf = PDFZeitschrift->new(
       title => '$foo - PDF',
);

$pdf->print_type;
print "\n";
$pdf->as_pdf;
```

Die aktuelle Klassenhierarchie ist in Abbildung 1 zu sehen.

Bei der Mehrfachvererbung ist dabei auf folgendes zu achten: Man darf nur einmal `extends` benutzen. Soll die Klasse von mehreren Basisklassen erben, so sind alle Basisklassen in einem `extends` aufzuführen:

```
extends 'Zeitschrift', 'PDF';
```

Macht man das nicht, so erbt die neue Klasse nur von den Basisklassen, die im letzten `extends` aufgeführt sind.

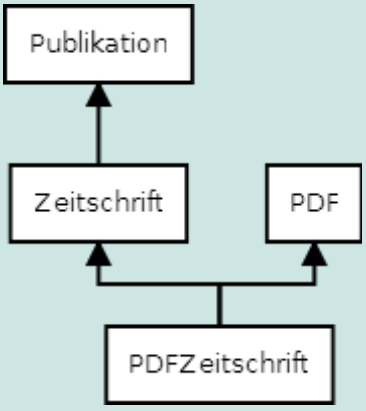

Abbildung 1: Klassenhierarchie

Ein

```
extends 'Zeitschrift';
extends 'PDF';
```

führt zu **keinem** Fehler. Moose akzeptiert das, aber bei der Ausführung des Testprogramms von oben bringt dann den Fehler

```
Can't locate object method "new" \
     via package "PDFZeitschrift" at
```

Die Klasse "PDFZeitschrift" kennt jetzt nur die Methode aus der Klasse "PDF".

Die Mehrfachvererbung hat aber einen großen Nachteil: Stellen mehrere Klassen Methoden mit dem gleichen Namen zur Verfügung, so findet Perl immer nur eine Methode - ohne Warn- oder Fehlermeldungen.

Bei der Suche nach der Methode geht Perl nach der Tiefensuche und "von links nach rechts" vor. Bezogen auf die Klassenhierarchie aus Abbildung 1 sucht Perl erst in der Klasse "PDFZeitschrift", dann "Zeitschrift", "Publikation" und zum Schluss erst in "PDF".

Hier kommt es also auf die Reihenfolge der Basisklassen beim `extends` an. Für das Beispiel gehen wir davon aus, dass die Klassen "PDF" und "HTML" jeweils eine Methode "output" definieren, die jeweils ihren Klassennamen ausgeben (der vollständige Code ist bei den Codebeispielen auf der Webseite vorhanden).

```
package MyTest;

use Moose;
extends 'PDF', 'HTML';
no Moose;

package main;

my $obj = MyTest->new;
$obj->output;
```

In diesem Fall ist die Ausgabe "PDF". Dreht man die Reihenfolge beim `extends` um, so ist die Ausgabe "HTML".

Attribute erweitern

Subklassen sind eine Spezialisierung der Basisklasse. Das kann dazu führen, dass es zwar Attribute mit dem gleichen Namen in den beiden Klassen gibt, die aber Einschränkungen in der Subklasse haben sollen. Mit Moose ist es sehr einfach, solche Attribute zu "erweitern". Im ersten Teil des Moose-Tutorials wurde ja schon auf Attribute im Detail eingegangen.

Nehmen wir also an, dass die Basisklasse folgendes Attribut definiert:

```
has format => (
    is      => 'ro',
    default => 'A4',
    isa     => 'Format',
);
```

Eine Minizeitschrift ist aber üblicherweise im A5-Format und damit würde das geerbete Attribut so nicht stimmen. Mit Moose kann man solche Attribute aber überschreiben:

```
has '+format' => (
    default => 'A5',
    isa     => 'MiniFormat',
);
```

Durch das '+' vor dem Attributnamen macht kenntlich, dass hier ein Attribut einer Basisklasse erweitert bzw. geändert werden soll. Alle Attributeigenschaften, die hier nicht angepasst werden, werden aus der Basisklasse übernommen.

Methoden überschreiben

Bei der Vererbung werden Methoden mitvererbt. Dadurch kennt das FooMagazin-Objekt die Methode `publish`, obwohl diese nur in der Superklasse definiert wird. Manchmal muss man eine Methode in der Subklasse aber überschreiben.

Natürlich kann man einfach

```
sub publish {
    my $self = shift;
    print "in " . __PACKAGE__ . "\n";
}
```

schreiben. Dann müsste man aber

```
$self->SUPER::publish( @_ );
```

schreiben, um die `publish`-Methode der Superklasse aufzurufen. `Moose` bietet hierfür etwas, das viel "netter" aussieht: `override` und `super`.

Mit `override` kann man `Moose` mitteilen, dass eine Methode überschrieben werden soll. Benutzt man diese Möglichkeit, kann man `super()` nutzen, um die Methode der Superklasse aufzurufen. Das würde dann wie folgt aussehen:

```
override 'publish' => sub {
    my $self = shift;
    print "in " . __PACKAGE__ . "\n";
    super();
}
```

Fallen bei der Mehrfachvererbung

Die Mehrfachvererbung ist aber auch Quelle einer ganz gemeinen Falle, auf die Curtis 'Ovid' Poe in seinem use.perl.org-Journal hinweist: Je nach Reihenfolge der Superklassen im `@ISA`-Array sind manche Methoden nie erreichbar. Z.B. bei diesem Code:

```
 {
     package Platypus;
     our @ISA = qw<SpareParts Duck>;
     package Duck;
     our @ISA = 'SpareParts';
     sub quack {}
     package SpareParts;
     our @ISA = 'Animal';
     sub quack {}
     package Animal;
 }
```

ergibt sich dieser Vererbungsbaum

```
        Animal
          |
     SpareParts
      |     |
      |   Duck
      |     |
     Platypus
```

Erweiterte Perl 5 OO

In den folgenden Absätzen werden einige erweiterte Ideen und Ansätze zur Objektorientierten Programmierung in Perl und Allgemein aufgezeigt.

Komposition ist besser als Vererbung

In Abschnitten weiter vorne habe ich ein paar Probleme mit der Vererbung - und speziell der Mehrfachvererbung - gezeigt. Viele dieser Probleme kann man durch Komposition statt Vererbung lösen.

Komposition ist das Zusammenfügen verschiedener Verhalten und Teile. Die Komposition entspricht einer "hat ein"-Beziehung - Die Anwendung "hat einen" Logger.

Für die Komposition mit Moose gibt es die Rollen, die ich in der nächsten Ausgabe ausführlich vorstellen werde.

Das Prinzip der Alleinigen Verantwortlichkeit

Es gibt ein weiteres Prinzip, das beim Objektorientierten Design berücksichtigt werden sollte, um guten wartbaren Code zu schreiben: Das Prinzip der alleinigen Verantwortlichkeit.

Man sollte einer Klasse immer nur eine Aufgabe geben. So sollte die Klasse Zeitschrift nur für die Zeitschriften-spezifischen Aufgaben verantwortlich sein und keine Verantwortung für Artikel etc. übernehmen. Dafür gibt es dann wieder eigene Klassen.

Wird diese Aufteilung konsequent durchgehalten, ist es immer sehr einfach zu sagen, welcher Teil der Anwendung für ein bestimmtes Verhalten verantwortlich ist.

DRY

Viele Fehler werden durch Copy'n'Paste vervielfacht. Aus diesem Grund, sollte Code nicht kopiert werden (DRY steht für "Don't repeat yourself"), sondern in kleine wieder verwendbare Abschnitte eingeteilt werden. Bei ähnlicher Funktionalität kann man versuchen, die Funktion so zu abstrahieren, dass eine gemeinsame Codebasis entsteht, die die ursprünglichen Funktionen verwenden.

Thomas Kappler

Rezension "Effective Perl Programming"

Effective Perl Programming
Joseph N. Hall, Joshua A. McAdams und brian d foy
Addison-Wesley
ISBN: 978-0-3214-9694-2
April 2010

Über die Frage, wie Perl "korrekt" oder "wartbar" oder einfach "besser" zu programmieren sei, wurde schon viel geschrieben. Conway's Perl Best Practices wurde trotz einiger umstrittener Ratschläge zum Standardwerk, und wir haben unzählige Artikel auf Blogs und Perlmonks. Braucht es da noch ein Buch über "Wege, besseres und idiomatischeres Perl zu schreiben"?

Das ist der Untertitel der zweiten Auflage von "Effective Perl Programming" von Joseph N. Hall, Joshua A. McAdams und brian d foy, erschienen April 2010 im Addison Wesley Verlag. Eine deutsche Übersetzung ist zur Zeit nicht angekündigt. Die erste Ausgabe, von Joseph N. Hall allein geschrieben, ist von 1996 und dürfte daher dem einen oder anderen bekannt sein. Natürlich hat sich in der Perl-Welt seitdem viel getan, und die zweite Ausgabe ist daher eine komplette Überarbeitung mit verdoppeltem Umfang von 470 Seiten.

Nicht geändert hat sich der Aufbau des Buches. An Stelle längerer Kapitel, die ein Thema von der Einführung bis zu fortgeschrittenen Inhalten schrittweise erläutern, setzen die Autoren hier Perlkenntnisse voraus und können sich daher auf konkrete Tipps beschränken. Diese haben die Form von etwa fünfseitigen Artikeln, in denen Information und Ratschläge zu einem bestimmten Thema konzentriert sind, beispielsweise "Typeglobs und die Symboltabelle" oder "Benutze Perl::Critic". Dabei werden alle Tipps von recht ausführlicher Hintergruninformation untermauert.

Diese Artikel sind auf 13 Kategorien verteilt: Elementares Perl, Idiomatisches Perl, Reguläre Ausdrücke, Subroutinen, Dateien und Filehandles, Zeiger (References), CPAN, Unicode, Distributionen, Testen, Warnungen, Datenbanken, und Verschiedenes. Da dürfte für alle was dabei sein, wobei man sieht, dass sich das Buch fast ganz auf die Sprache selbst beschränkt. Ausnahmen sind lediglich wichtige Infrastruktur wie CPAN und Datenbanken; zu Web- oder Grafikprogrammierung oder anderen Anwendungsgebieten findet man hier nichts. Die 13 Kategorien unterscheiden sich teils stark in ihrer Länge, was ungefähr ihrer Komplexität und ihrer Bedeutung für den Perlprogrammierer entspricht. So werden idiomatisches Perl, reguläre Ausdrücke und Testen mit jeweils etwa 50 Seiten bedacht, während Zeiger, Warnungen und Datenbanken mit etwa 20 auskommen müssen.

Die einzelnen Artikel sind sehr gut strukturiert. Längere Artikel sind in Unterkapitel aufgeteilt. Nummer 51 über die Testoperatoren für Dateien beginnt beispielsweise mit einer Einführung, die `stat` und einige der `-X`-Operatoren vorstellt und auf das relevante Perldoc hinweist. Es folgen Unterkapitel über die virtuelle Filehandle `_`, die überflüssige `stat`-Aufrufe vermeidet, und die neuen "stacked file tests" der Form `-r -w $file` in Perl 5.10. Drei "Things to remember", eine prägnante Wiederholung der wichtigsten Punkte, schließen jedes Kapitel.

Ich habe in jedem Kapitel interessante Tipps und mir noch unbekanntes Detailwissen über Perl gefunden. Selbst das einführende "The Basics of Perl" bietet noch Neues, zwar wohl nicht für Perlgurus, aber auch nicht nur für Anfänger. Es erhöht den Wert des Buches hier ungemein, dass sich die Autoren nicht scheuen, in die Tiefe zu gehen und auch vor den Feinheiten von `perlop` und Ähnlichem nicht Halt machen. So stellen die Autoren in "Kenne den Unterschied zwischen Arrays und Listen" richtigerweise fest, dass selbst viele

Perlbücher eben diesen Unterschied ignorieren. Der Artikel erklärt dann genau, dass das Komma ein Operator ist und dieser im Skalarkontext das rechteste Element zurückgibt, während es bei Listen die Anzahl der Elemente ist. Beispiele zeigen, dass `my @array = qw(eins zwei); my $count = @array;` daher nicht dasselbe wie `my $rightmost = qw(eins zwei);` ist. Selbst die Warnung, dass der naheliegende Test `my $scalar = (1, 2, 3);` hier in die Irre führt, fehlt nicht. Dann kommt Zuweisung (`=`) im Listenkontext dran, inklusive dem wohl recht unbekannten Detail, dass der Zuweisungsoperator hier auch einen Rückgabewert hat, nämlich die Anzahl der Elemente auf der rechten Seite. Dies ermöglicht den "Goatse-Operator" `=()=`. Möchte man wissen, wie viele Elemente ein split produziert, wenn man die Elemente selbst aber nicht braucht, kann man `my $count =()= split /:/, $line;` schreiben.

Auf diese Weise kann auch ein Perl-erfahrener Leser zu vielen Themen, mit denen er vertraut ist, noch etwas Neues erfahren. Auch Dinge, die man nicht jeden Tag braucht, von Prototypen über die Feinheiten von Unicode zu pack und unpack kommen dran.

Zwei besonders starke Kapitel sind Reguläre Ausdrücke und Testen. Bei den bekannten RegExes sparen sich die Autoren eine Einführung und kommen direkt zu fortgeschrittenen Themen, wobei es vor allem um Performance geht. Viele wichtige Tipps sind hier konzentriert zusammengefasst und erklärt, beispielsweise Non-capturing Groups zu verwenden, die langsamen Matchvariablen wie `$&` zu vermeiden, Backtracking zu umgehen, und die RegExes vorzukompilieren. Schliesslich kommt auch Benchmarking dran, illustriert mit einem schönen Beispiel aus dem Logfile-Parsen.

Das Kapitel über Testen stellt nicht nur praktische Werkzeuge wie `prove` vor, das mir bisher entgangen war, sondern geht auch auf Entwurfsprinzipien ein, die nicht unbedingt Perl-spezifisch sind. So lernt man, wie Dependency Injection und Mocking für wart- und testbaren Code und für fokussierte Tests sorgen können. Nützlich fand ich auch die Artikel über "Modulinos" - schreibe auch kleinere Skripte als Module, so dass man leicht ein paar Tests schreiben kann - und über das Testen von Datenbank-Code mit SQLite.

Unverständlich ist, dass zwei wichtige und grundlegende Themen fehlen: Objektorientierung und Parsen. Bei 500 Seiten ist es zwar nötig, irgendwo eine Grenze zu ziehen. Doch es ist heute kaum mehr möglich, ohne objektorientiertes Perl über die Runden zu kommen, ob man es mag oder nicht. Gerade hier, wo es durch Perls sehr offenes und etwas schräges Objektsystem unzählige Möglichkeiten gibt, sich selbst in den Fuß zu schießen, wären einige Tipps und Hintergründe hilfreich gewesen. Auch gibt es wohl wenige Perl-Programmierer, die nicht immer wieder mit XML, CSV oder irgendwelchen Log- und Konfigurationsdateien zu tun haben. Außer einer kurzen Erwähnung von Text::CSV ist dazu aber nichts enthalten.

Einzelne Ratschläge muten etwas merkwürdig an, aber das ist bei 120 Artikeln wohl kaum zu vermeiden. So beginnt etwa der Artikel "Vermeide übertriebene Interpunktion" (punctuation) mit dem Hinweis, dass man bei Subroutinen-Aufrufen die Klammern weglassen kann, muss dann aber drei Ausnahmen dieser Regel auflisten und darauf hinweisen, dass man die Subs ja mit `use subs` vordeklarieren könne. Wie "Effective" und wartbar das dann ist, ist zweifelhaft.

Für erfahrene Perlentwickler ist das eben genannte Problem natürlich keines, sie kennen die Feinheiten von Subroutinenaufrufen. Dies führt zur Frage, an wen sich das Buch denn richtet. Anfänger können es auch nicht sein, da es keine Einführung in Perl beinhaltet. Und das ist gut so, da es um Stolperfallen und praktische Tipps gehen soll.

Hat das Buch also überhaupt eine passende Zielgruppe? Ich denke ja, und zwar genau die fortgeschrittenen Perl-Programmierer, die irgendwo zwischen den Anfängern und den Gurus sind. Ich schätze mich selbst so ein, und ich habe aus diesem Buch viel Nützliches gelernt, und dies recht einfach und flüssig, ohne woanders Nachschlagen zu müssen. Wer Perl flüssig schreibt und liest, aber hier und da ins Zweifeln kommt oder in Code von erfahreneren Entwicklern ab und zu Rätselhaftes findet, ist hier richtig.

Bleibt noch die Frage, wie sich Effective Perl Programming, nennen wir es kurz EPP, vom recht ähnlichen und fast gleich langen Klassiker Perl Best Practices (PBP) von Damien Conway unterscheidet. Die erste Hälfte von Conway's Buch beschäftigt sich mit den Grundlagen guten Perl-Codes, von Formatierung und Namensgebung über die ganzen Kontrollstrukturen bis zur Syntax von Subroutinenaufrufen. Diese Themen haben keine Entsprechung in EPP, da dieses sich an fortgeschrittenere Programmierer richtet. Die späteren Kapitel in PBP überschneiden sich teilweise mit EPP, wo es zum

Beispiel um Fehlerbehandlung, Module, und Testen geht. In der Behandlung der Themen unterscheiden sich die Bücher aber deutlich. PBP gibt klare Empfehlungen und hält die Begründungen eher kurz. Dadurch kann es ein breiteres Themenspektrum als EPP bearbeiten, andererseits muss man woanders nachschlagen, wenn man mehr wissen will. EPP hingegen geht bei jedem Punkt mehr in die Tiefe und scheut auch vor selten benutzten Sprachelementen nicht zurück. Diese Unterschiede ergeben sich aus der Zielgruppe der Bücher: PBP zielt eher auf Anfänger als EPP.

Das Format und die Druckqualität des Buches sind sehr gut. Der Aufbau aus kurzen, in sich abgeschlossenen Artikeln eignet sich wunderbar dazu, das Buch immer mal wieder kurz in die Hand zu nehmen. Der Preis ist mit 23 Euro (Amazon.de, 2010-12-11) für ein technisches Buch sehr gemäßigt.

Insgesamt kann ich Effective Perl Programming der oben beschriebenen Zielgruppe sehr empfehlen. Da es keine Einführungs- und Übersichtskapitel hat und thematische Löcher vor allem bei der Objektorientierung aufweist, kann es zwar nicht als alleinige Perl-Referenz dienen, doch das war auch nicht das Ziel des Buches. Wer seine Kompetenz in der eigentlichen Sprache Perl verbessern will, ist damit gut beraten.

Herbert Breunung

Rezension
Bücher zur IT - Geschichte

Gottfried Wolmeringer
Coding for Fun
"IT-Geschichte zum Nachprogrammieren"
Galileo Computing - 2008
ISBN 978-3-8362-1116-1
deutsch, 573 S., Klappbroschur, mit DVD

H.R. Wieland
Computergeschichte(n)-nicht nur für Geeks
"Von Antikythera zur Cloud"
Galileo Computing - 2011
ISBN 978-3-8362-1527-5
deutsch, 605 S., Klappbroschur, mit DVD

Bereits im $foo-Heft Nummer 1 stellte Renée Bäcker ein wichtiges Buch vor (Perl Testing) und einige weitere folgten. Doch ab jetzt soll jede Ausgabe mindestens eine Rezension enthalten. Gastschreiber sind auch hierzu immer willkommen. Dabei muss nicht jeder beschriebene Einband Perl im Titel tragen, denn vielerorts lässt sich stetig dazulernen.

Für den Neuanfang wählte ich 2 recht ähnliche Bücher aus dem Galileo Verlag zum Thema Allgemeinwissen. Also was Programmierer unter Allgemeinwissen verstehen natürlich. "Was konnten die ersten Chips?", "Steht DOS nicht in Wirklichkeit für 'Dirty Operating System'?" oder "Wie funktioniert ein neuronales Netz?" Mir gefällt, dass die Bücher den Anspruch haben Freude zu bereiten, was sie auch auf leicht unterschiedliche Art tun. "Computergeschichte(n)-nicht nur für Geeks" ist eher ein Buch zum Schmökern, wohingegen "Coding for Fun" tatsächlich sich eher an Menschen richtet, die Spaß am Programmieren haben. Die Titel sind also treffend gewählt, aber textlich gibt es schon Überschneidungen.

Coding for Fun war das erste Buch des sonst ernsteren Fachverlages in der Richtung und das Thema ist definitiv zu groß, um zwischen 2 Buchdeckel gepackt zu werden. Auch sind Programmierer die fließend zwischen 8 und mehr Sprachen wechseln nicht zu häufig. Also spezialisierte man den praktischen Teil und brachte "Coding for Fun C++", "Coding for Fun C#", "Coding for Fun Python", alles recht verschiedenartig Titel, die den verschiedenartigen Kulturen der Sprachen Rechnung tragen.

Und nun brachte man ein Buch zu dem Thema, dass sich weniger auf die technischen Details stützt, weniger Längen hat, etwas geschliffener und inhaltlich wesentlich runder ist. Vor allem hat es es mehr Bilder wie vom ersten aufgeklebten Computerbug (es war tatsächlich ein Käfer) oder historischen Computerspielen. Ganze Abschnitte wie etwa zu den antiken und historischen Rechenmaschinen finden sich nicht im früheren Buch. Beiden Werken liegt aber eine CD mit viel freier Software bei, die dazu einlädt, die gelesene Geschichte am eigenen Rechner zu simulieren und zu erweitern.

Wer selbst einmal nachvollzog wie eine Turing-Maschine arbeitet oder wie die ersten Computer programmiert, ähm gesteckt wurden, bekommt eine realistischere Perspektive woraus sich die heutigen Chips entwickelt haben und was wirklich in ihnen vorgeht. Welche Menschen und Geschichten sich damit verbunden haben ist mehr noch im späteren Band das Hauptthema.

Welche Rolle spielte Seymour Cray oder Nikolaus Wirth? Wie programmiert man ein Fraktal? Was gab es vor World of Warcraft? All diesen Fragen gehen die Bücher nicht auf den Grund (geht auch gar nicht auf knapp 600 Seiten), aber man bekommt einen guten Überblick. Mir persönlich fehlte da BeOS und auch Perl sowie Larry Wall wird lediglich die Existenz bescheinigt, aber dennoch halte ich es für ein gelungenes Sprungbrett ins Thema Computergeschichte.

Thomas Fahle

HowTo

Parallel::Iterator - Mehrere Tasks parallel ausführen

`Parallel::Iterator` von Andy Armstrong erlaubt die gleichzeitige (parallele) Ausführung mehrerer Tasks auf einem Rechner mit (vorzugsweise) mehreren CPUs.

Dabei kümmert sich Parallel::Iterator um korrekte Interprozeß-Kommunikation, Forking und Verteilung auf mehrere CPUs, so daß man sich voll und ganz auf die eigentliche Aufgabe konzentrieren kann.

Anwendungsgebiete

Da der Forking-Prozess einen nicht zu unterschätzenden Overhead mit sich zieht, bietet sich die Verwendung von `Parallel::Iterator` vor allem bei folgenden Aufgaben an:

- Anwendungen, die oft oder lange auf IO/Netzwerk warten
- CPU-intensive Kalkulationen, die sich (einfach) auf mehrere CPUs verteilen lassen

Iteratoren, Tasks und Worker

`Parallel::Iterator` verwendet Iteratoren, Worker und Tasks. Ein Worker ist eine Subroutine, welche die eigentliche Arbeit ausführt - Tasks sind Listen der Parameter, die an den Worker übergeben werden - Iteratoren gehen durch die Taskliste und parallelisieren den Worker.

Neben der Möglichkeit eigene Iteratoren zu definieren, bietet `Parallel::Iterator` bereits zwei fertige Iteratoren - `iterate_as_array` und `iterate_as_hash` - an, die eine Referenz auf ein Unterprogramm (Worker) und eine Referenz auf die Liste der Tasks entgegen nehmen.

In dem folgenden einfachen und nicht wirlich praxisrelevantem Beispiel wird eine Liste von Quadratwurzeln aus einer Liste von Zahlen parallel ermittelt.

iterate_as_array

Der Worker `square_root` erhält als ersten Parameter den Index aus `@numbers` (Tasks) und als weiteren Parameter die zu bearbeitende Zahl.

Um die Ergebnisse im `@output` korrekt anzuordnen, müssen Index **und** Ergebnis aus `square_root` zurückgegeben werden.

Falls die Reihenfolge der Ergebnisse keine Rolle spielt, kann der Index in der Rückgabeliste entfallen.

```
#!/usr/bin/perl
use warnings;
use strict;

use Parallel::Iterator qw(iterate_as_array);

# Tasks
       # Indizes: 0  1  2  3  4  5  6
my @numbers = qw/ 1  4  9  16 25 36 49 /;

my @output = iterate_as_array(
    \&square_root,
    \@numbers,
);

print join("\t", @output), "\n";

# Worker
sub square_root {
   my ($index, $number) =  @_;
   my $sr = sqrt($number);

   # return index and value
   return ($index,$sr);
}
```

iteratate_as_hash

Das Unterprogramm `square_root` (worker) erhält als ersten Parameter den Key aus `%numbers` (Tasks) und als weiteren Parameter die zu bearbeitende Zahl.

Um die Ergebnisse im `%output` korrekt zuzuordnen, müssen Schlüssel **und** Ergebnis aus `square_root` zurückgegeben werden.

```
#!/usr/bin/perl
use warnings;
use strict;

use Parallel::Iterator qw(iterate_as_hash);

# Tasks
my %numbers = (
   1  => 1,
   4  => 4,
   9  => 9,
   16 => 16,
   25 => 25,
   36 => 36,
   49 => 49,
);

my %output = iterate_as_hash(
    \&square_root,
    \%numbers,
);

my @numbers = sort { $a <=> $b }keys %output

foreach my $number ( @numbers ) {
    print "The square root of $number is "
          . "$output{$number}\n";
}

# Worker
sub square_root {
   my ( $key, $number ) = @_;
   my $sr = sqrt($number);

   # return key and value
   return ( $key, $sr );
}
```

Optionen und Optimierungen

Die beiden dargestellten Iteratoren werden mit sinvollen Performance-Vorgabewerten ausgeliefert. Individuelles Tuning der Iteratoren ist durch Optionen, deren Erläuterung den Rahmen dieser Einführung deutlich sprengen würde, möglich.

Beispiel Link-Checker

Eine typische Anwendung, die kaum Prozessorzeit erfordert, aber oft lange auf IO wartet, ist das Holen von Webseiten.

In diesem einfachen Beispiel wird der HTTP-Statuscode verschiedener Websites ermittelt.

Über die Option `workers` wird die Anzahl der parallelen Tasks begrenzt.

```
#!/usr/bin/perl
use strict;
use warnings;

use LWP::UserAgent;
use Parallel::Iterator qw/iterate_as_array/;

#  a list of pages to fetch
my @urls = qw(
  http://www.perl-howto.de
  http://www.perl.org
  http://www.yahoo.de
  http://www.google.de
  http://www.tagesschau.de
  http://www.zdf.de
);

my $ua = LWP::UserAgent->new();

# this worker fetches a page and returns
# the HTTP status code
my $worker = sub {
    my $index    = shift;
    my $url      = shift;
    my $response = $ua->get($url);
    return ( $index, $response->code() );
};

# Number of parallel tasks
my %options = ();
$options{workers} = 2;

# Fetch pages in parallel
my @status_codes = iterate_as_array(
\%options, $worker, \@urls );

# Display results
my %codes = ();

# Hash slice
@codes{@urls} = @status_codes;

# output results
my $format = "%-40s %s\n";
printf( "$format", 'URL', 'Status' );
foreach my $url ( sort keys %codes ) {
    printf( "$format", $url, $codes{$url} );
}
__END__
```

TPF News

Was die TPF 2010 so gemacht hat

Der Schatzmeister der Perl Foundation (TPF), Dan Wright, hat einen Einblick in die (finanziellen) Aktivitäten gegeben.

Er hat dabei drei große Bereiche unterschieden: Events, Marketing und Entwicklung.

Bei den Events hat TPF vier Konferenzen unterstützt. Von der Abwicklung der Teilnahmegebühren bis zu Versicherungen. Auch das "Send-a-newbie"-Programm der Enlightened Perl Organization wurde unterstützt.

Für Marketing auf den verschiedenen Events hat TPF 1.000 USD ausgegeben. Weitere 1.800 USD wurden für Markenanträge in Kanada, Europa und Japan ausgegeben.

Im Bereich Entwicklung hat TPF das meiste Geld ausgegeben: 1.500 USD kostet die Mitgliedschaft im Unicode Konsortium.

Für die Weiterentwicklung von Perl 6 hat die TPF drei Grants vergeben, die mit insgesamt 14.000 USD zu Buche schlagen.

Der größte Anteil an den Ausgaben dürften aber die 25.800 USD für Dave Mitchell sein, die er für seinen "Fixing Perl 5 Core Bugs"-Grant bekommen hat.

Insgesamt 6.000 USD würden für die Standard-Grants ausgegeben. 8 solcher Grants wurden 2010 beendet und ausbezahlt.

Grants für das 4. Quartal 2010

Für das 4. Quartal 2010 gab es zwei Grantanträge bei der Perl Foundation.

Der Antrag von Jonathan Leto zum Thema "Improve Parrot Embed/Extend Subsystem" wurde angenommen. Abgelehnt hingegen wurde der Antrag von Naim Shafiev mit dem Thema "AnyEvent::HTTPBenchmark".

Parrot Embed API

Jonathan Leto arbeitet daran, die Dokumentation und die Tests der Parrot Embed API zu verbessern. Zu den ersten Ergebnissen der Arbeiten gehören zusätzliche Dokumentation der Parrot Funktionen und Typsignaturen. Die Arbeiten sind auf GitHub unter *https://github.com/parrot/parrot/commits/leto%2Fembed_grant* zu finden.

Grant-Abschluss: Embedding Perl into C++ Applications

Leon Timmermans hat seinen Grant abgeschlossen. Sein Grant umfasste die Dokumentation der API, Portierung auf Perl 5.10 und Unterstützung für Reguläre Ausdrücke. Während des Grants ist Timmermanns auf ein großes Problem gestoßen: Portabilität auf Windows. Hier fehlt es an Unterstützung durch Tools.

Aus verschiedenen Gründen hatte sich der Grant immer wieder verzögert, jetzt hat Leon aber den Abschluss geschafft.

Grant-Abschluss: Mojolicious-Dokumentation

Sebastian Riedel hat seinen Grant der Perl Foundation abgeschlossen. Aus gesundheitlichen Gründen hatte sich der Abschluss verzögert. Riedel hat jedes Modul in dem Projekt ausführlich dokumentiert und ein Tutorial für den schnellen Start wurde geschrieben. Das, jetzt gut dokumentierte, Modul ist auf CPAN unter *http://search.cpan.org/dist/Mojolicious* zu finden.

In der Zwischenzeit erreichte Mojolicious die Version 1.0

Grant-Abschluss: Dokumentation zu Perlbal

José und Bruno haben ihren Grant abgeschlossen. Folgende Themen wurde von den beiden dokumentiert:

- Installation
- Configuring Perlbal as a Reverse Proxy
- Configuring Perlbal as a Web Server
- Configuring Perlbal as a Load Balancer
- Configuring Perlbal as a Selector
- Managing and configuring Perlbal on-the-fly
- Writing Plugins
- Perlbal's Architecture at a glance (Perlbal::Manual::Internals)
- Perlbal's Logging mechanism
- Perlbal's Debugging system
- Perlbal's High/Low Priority Queueing system
- Perlbal's fail-over mechanism

Die Dokumentation ist auf GitHub zu finden: *https://github.com/cog/perlbaldoc/*

Perl 6 Tablets

Im August 2010 hat die Perl Foundation einen Grant an Herbert Breunung vergeben. Er wird an den Perl 6 Tablets arbeiten. Das ist eine Referenz für Perl 6, die welche die Vorteile von Hypertext konsequent nutzt.

Breunung hat Tablet Nr. 2 (Designprinzipien - jetzt Nr.1) abgeschlossen und er wird in den nächsten Wochen an Nr. 1 und Nr. 3 parallel arbeiten. Derzeit befinden sich die Themen Literale, Variablen und Operatoren im Aufbau.

Manual für Spieleentwicklung mit SDL

Diesmal gibt es nur eine kurze Meldung zu dem Grant: Seit dem letzten Update wurden SDLx::Widget 0.07 und SDL-2.525_3 veröffentlicht. Außerdem hat Kartik Thakore weiter an der Dokumentation für Spiele-Programmierung mit SDL gearbeitet. Kartik hat mittlerweile 5 von 11 Kapiteln abgeschlossen. Die anderen 6 Kapitel werden in Kürze veröffentlicht.
Der Fortschritt ist unter diesen Adressen zu beobachten:

- *http://sdlperl.ath.cx/releases/SDL_Manual.pdf*
- *http://sdlperl.ath.cx/releases/SDL_Manual.html*
- *http://github.com/PerlGameDev/SDL_Manual*

Fixing Perl5 Core Bugs

Die Perl Foundation hat den Grant "Fixing Perl5 Core Bugs" von Dave Mitchell um weitere 400 Stunden verlängert, der jetzt somit insgesamt 900 Stunden umfasst. Die Verlängerung hat einen Wert von 20.000 USD, die durch eine Spende von Booking.com ermöglicht wurde.

Bisher hat Mitchell 520 Stunden an dem Grant gearbeitet und dabei 127 Core bugs geschlossen.

Dave Mitchell schickt wöchentliche und monatliche Berichte an die Perl 5 Porters.

Im November hat Dave Mitchell hauptsächlich an einem Bug gearbeitet, um einen Segfault zu verhindern wenn z.B. ein Signalhandler eine Datei schließt während die Datei gelesen wird.

Insgesamt hat er knapp über 54 Stunden im November an seinem Grant gearbeitet.

CPAN News XVII

Data::Format::Pretty::Console

Wie gibt man Daten am besten auf der Konsole so aus, dass man auch gut erkennen kann welche Informationen vorhanden sind? Mit `Data::Format::Pretty::Console` gibt es ein Modul, das mit verschiedenen Datenstrukturen umgehen kann: Einfache Werte, Listen oder auch komplexe Datenstrukturen. Aktuell gibt es leider keine Optionen, mit der man das Verhalten des Moduls (z.B. Reihenfolge von Tabellenspalten) beeinflussen kann.

```
use Data::Format::Pretty::Console
    qw(format_pretty);
my $result = {
  verkauf => [
      {
           Monat => 'Januar',
           'verk. Einzelheft' => 12,
           'verk. Abos' => 2,
      },
      {
           Monat => 'Februar',
           'verk. Einzelheft' => 33,
           'verk. Abos' => 6,
      },
  ],
};
print format_pretty($result);

perl@ubuntu:~/foo_17$ perl pretty_print.pl
verkauf:
.-----------------------------------------.
| Monat   | verk. Abos | verk. Einzelheft |
+---------+------------+------------------+
| Januar  |          2 |               12 |
| Februar |          6 |               33 |
'---------+------------+------------------'
```

Text::SpamAssassin

SpamAssassin ist vielen vermutlich in Bezug auf Spamerkennung bei Emails bekannt. `Text::SpamAssassin` ist entwickelt worden, um Spam in normalen Texten zu entdecken. Das kann z.B. für Wikis ganz interessant sein.

```
use Text::SpamAssassin;

my $sa = Text::SpamAssassin->new(
    sa_options => {
       userprefs_filename =>
           'comment_spam_prefs.cf',
    },
);

$sa->set_text($content);

my $result = $sa->analyze;
print "result: $result->{verdict}\n";
```

DBIx::NoSQL

Mit `DBIx::NoSQL` bekommt man ein NoSQL-Frontend für relationale Datenbanken. Das Modul arbeitet dafür mit einer speziellen Tabelle in der Datenbank und speichert dort die Daten im JSON-Format. Zur Zeit wird SQLite als Datenbanksystem genommen.

```
use DBIx::NoSQL;

my $store =
   DBIx::NoSQL->connect( 'store.sqlite' );

$store->set('Ausgabe'=>'Fruehjahr 2011'=> {
    name => 'Frühjahr 2011',
    published => '01.02.2011',
} );

$store->exists('Ausgabe'=>'Fruehjahr 2011');
# 1

my $foo =
  $store->get('Ausgabe'=>'Fruehjahr 2011');
```

App::highlight

Dieses Modul liefert ein kleines Programm, das ähnlich wie `grep` ist. Nur filtert es nicht die Treffer, sondern hebt diese besonders vor:

```
$ cat words.txt | highlight ba
foo
<<ba>>r
<<ba>>z
qux
```

String::Palindrome

Mit `String::Palindrome` kann man prüfen, ob ein String ein Palindrom ist, d.h. rückwärts genauso geschrieben wird wie vorwärts. Außerdem bietet es die Möglichkeit, Arrays daraufhin zu prüfen, ob das Array in umgekehrter Reihenfolge genauso ist wie in normaler Reihenfolge.

```
use String::Palindrome qw(is_palindrome);

my $string = 'lagerregal';
is_palindrome( $string ); # 1

my @array = (1,2,2,1);
is_palindrome( @array ); # 1
```

Number::Tolerant

Normalerweise möchte man bei einem numerischen Vergleich wirklich auf Gleichheit prüfen, wer aber etwas toleranter sein will, kann mit `Number::Tolerant` arbeiten. Es gibt Fälle, bei denen man keine exakten Zahlen bekommt, die man aber trotzdem akzeptieren will.

```
use Number::Tolerant;

my $range  = tolerance(10 => to => 12);
my $random = 10 + rand(2);

die "I shouldn't die"
  unless $random == $range;

print "This line will always print.\n";
```

Termine

Februar 2011

01. Treffen Frankfurt.pm
Treffen Vienna.pm
03. Treffen Dresden.pm
05.-06. FOSDEM
07. Treffen Hamburg.pm
08. Treffen Stuttgart.pm
12. Linuxinformationstag Oldenburg
14. Treffen Ruhr.pm
Treffen Hannover.pm
16. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
22. Treffen Bielefeld.pm
23. Treffen Berlin.pm

März 2011

01. Treffen Frankfurt.pm
Treffen Vienna.pm
03. Treffen Dresden.pm
07. Treffen Hamburg.pm
08. Treffen Stuttgart.pm
14. Treffen Ruhr.pm
Treffen Hannover.pm
16. Treffen München.pm
Treffen Darmstadt.pm
19.-20. Chemnitzer Linux-Tage
21. Treffen Erlangen.pm
22.-25. GUUG Frühjahrsfachgespräch
29. Treffen Bielefeld.pm
30. Treffen Berlin.pm

April 2011

04. Treffen Hamburg.pm
05. Treffen Frankfurt.pm
Treffen Vienna.pm
07. Treffen Dresden.pm
09. Grazer Linuxtag
11. Treffen Ruhr.pm
Treffen Hannover.pm
12. Treffen Stuttgart.pm
18. Treffen Erlangen.pm
20. Treffen Darmstadt.pm
27. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.perl-community.de

Perl-Community.de ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perlmongers.de/
http://www.pm.org/

Das Online-Zuhause der Perl-Mongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Außerdem kann man auch neue Gruppen gründen und bekommt Hilfe...

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern.

http://www.perl-foundation.org

Die Perl-Foundation nimmt eine führende Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundationen geleistet. Jedes Jahr werden Studenten beim "Google Summer of Code" betreut.

http://www.Perl.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

einraumwerk

Die Individual-Programmierung für OTRS-Anwender

Leistungen

Spezialisierung auf die individuelle Entwicklung von Modulen für OTRS.

Langjährige Erfahrung mit OTRS und der Programmierung von speziellen Anforderungen ermöglicht eine effiziente und kostensparende Arbeitsweise.

Mit der Bewältigung großer Aufgaben ebenso vertraut wie mit der sorgfältigen Abwicklung kleinerer Problemlösungen.

Bewährte Vorgehensweise:

* Erstellung technischer und funktionaler Spezifikationen
* Auf Basis der Spezifikation die Sicherstellung hoher Qualität
* Alternative Lösungsszenarien werden wenn möglich aufgezeigt
* Erstellung des verbindlichen Zeitplans und die dazugehörige Kostenschätzung

Leistungen:

* Entwicklung neuer Module und Funktionen für OTRS
* Anpassung vorhandener Module bei Versions-Wechsel

Kontakt:
Uwe Dieckmann
ud@einraumwerk.de

Chemnitzer
Linux-Tage

"Freiheit leben"

Ein Wochenende voll
interessanter Themen
rund um Linux

19. und 20. März 2011

Im Neuen Hörsaalgebäude der
Technischen Universität Chemnitz

Infos unter http://chemnitzer.linux-tage.de