Moderne Systemadministration

Modern Art der Systemadministration mit Perl

Über den Autor (1)

- A. Nonymous alias Unknown alias Max Mustermann
  - Unix-Systemadministrator mit Europas größter Websphere Installation mit mehreren tausend J2EE Applikations-Servern
  - Vom Chef gab es leider keine Genehmigung, dass der Mitarbeiter selbst auftreten darf …

Über den Autor (2)

* Alvar C.H. Freude
  * Berät obigen Dienstleister in Sachen Perl
  * Freiberuflicher Trainer, Software-Entwickler, Berater
  * http://www.perl-blog.de/
  * http://alvar.a-blast.org/
  * alvar@a-blast.org

Um was geht es

* Komplexe Infrastruktur mit rund 5000 Application-Servern / Websphere
* Plattform: AIX, Sun Solaris
* Lokale Entwicklungsplattform: Windows XP
* Skripte laufen auf zentralen Konfigurationsdiensten unter Solaris
* Skripte generieren Jobs und Konfigurations-Informationen

Bisherige Lösung

- Alt: Monolithischer Code
  - Hauptanwendung:
    - eine .pl-Datei
    - 9645 Zeilen Code
    - Mit Subs, strict, warnings, einige Module
      - trotzdem schwer wartbar
    - Keine Tests

Nun: modularer Code

* Viele neue Funktionen wurden verlangt, XML usw.
  * Ergebnis:
    * 51 Perl-Module
    * 22883 Zeilen Code inkl. POD
    * 12708 ohne POD
    * 500 generierte Module
    * 6752 Zeilen Test-Code

Vorteile

- Code ist:
  - übersichtlicher
  - besser wartbar
  - testbar
  - erweiterbar
- Niedrigere Fehlerrate
- bessere Handhabung

Nachteile

- Höhere Komplexität:
  - Durch OO, viele Module, viele kleine Dateien
    - Man muss sich auskennen und Doku lesen, um mal eben was zu ändern
      - Trial and Error nicht möglich
        - letztendlich wieder ein Vorteil!

Ergebnis Vor-/Nachteile

- Insgesamt überwiegen die Vorteile eindeutig
  - Nur so sind Erweiterungen einfach implementierbar
  - Bessere Trennung zwischen den einzelnen Bereichen
  - Auf Änderungen kann schneller und sicherer reagiert werden.

Genutzte Umgebung

- Entwicklungssystem Windows XP
  - Strawberryperl
  - Eclipse+Perl-Plugin
  - Subversion
    - Datenbank Original mit Oracle
      - Bei den Tests mit SQLite simuliert
      - oder via JDBC gegen eine Entwicklungsdatenbank

Arbeitsweisen

Genutzte Arbeitsweisen und Module

Module::Starter und *Module::Build*

- Ein Skript nutzt *Module::Starter*, *Module::Starter::Smart* und *Module::Starter::PBP*, um neue Module anzulegen
  - angepasste Templates
    - Speicherung von Name und E-Mail-Adresse vom Autor
- *Module::Build* als Build-System
  - aber nur zum Ausführen der Tests und Coverage-Analyse

Tests und Coverage-Analyse

- Über 1500 einzelne Tests in 35 Test-Skripten
  - Teilweise 100% Testabdeckung
  - Manche Teil-Altlasten keine oder kaum Tests
  - Voll-Altlasten keine Tests
- Vorteil: viele potentielle oder tatsächliche Fehler entdeckt
- Durch fehlende Tests blieb aber auch so mancher Fehler unentdeckt

Was ist hier falsch?

```
my $config_data = $self->dbh->resultset("config_delta")->search(
    {
    stage      => $app->stage   || die "Stagename fehlt\n",
    release    => $app->release || die "Anwendungs-Release fehlt\n",
    region     => $app->region  || die "Region fehlt\n",
    });
```

Das macht der Code:

```
my $config_data = $self->dbh->resultset("config_delta")->search(
    {
    stage => $app->stage || die
                (
                "Stagename fehlt\n",
                "release",
                $app->release || die
                    (
                    "Anwendungs-Release fehlt\n",
                    "region",
                    $app->region  || die "Region fehlt\n"
                    )
                ),
});
```

So funktioniert es wie gewollt

```
my $config_data = $self->dbh->resultset("config_delta")->search(
   {
   stage      => ( $app->stage   || die "Stagename fehlt\n" ),
   release    => ( $app->release || die "Anwendungs-Release fehlt\n" ),
   region     => ( $app->region  || die "Region fehlt\n" ),
   });
```

Vorteile von Tests

* Tests und die Nutzung von *Devel::Cover* helfen, auch subtilere Fehler zu finden
* Tests finden Fehler in Programm-Teilen, die nicht so häufig genutzt werden
* Stellt klar, dass einmal funktionierendes weiterhin funktioniert

Nachteile von Tests

* Es ist erstmal nicht eingängig, dass es insgesamt Zeit erspart:
  * Es ist erstmal mühsam
  * Wer dabei ist, Perl zu lernen und sich dann auch noch ins Testing einarbeiten muss . . .
  * Die Vorteile der Testabdeckung sind in der Produktion schwer zu verkaufen

Code-Qualitäts-Tests

* Perl::Critic und Test::Perl::Critic helfen, Codierungs-Regeln einzuhalten
  * Severity 3 ist (mit wenigen Ausnahmen) gut schaffbar
  * In die Tests integriert, kaum weiterer Aufwand
* Bessere Wartbarkeit durch erzwungeneren besseren Code
  * z.B. explizitier Import

Performance-Problem mit Windows

* Dank Virenscannern ist unter Windows der Umgang mit Dateien oft langsam
* *Devel::Cover* ist zusammen mit *Perl::Critic* besonders langsam
  * Kein *Perl::Critic* wenn testcover läuft
  * `if ( $INC{'Devel/Cover.pm'} ) { ... }`

Code-Formatierung

* Einsatz von *Perl::Tidy* zur Code-Formatierung
  * Einheitlicher Stil bei allen Entwicklern
  * automatisiert bessere Übersicht
  * Command-Shift-F ist schnell machbar …

Objektorientierung

* Objektorientierung mit Class::Accessor
  * Macht die Erzeugung von Accessoren zum Kinderspiel
  * Aber keine Inside-Out-Objekte
  * object->method und $object->{method} funktionieren
    * => Aufpassen
* Moose bietet mehr, wäre besser, ist modern

Datenbanken

* Nutzung von *DBIx::Class*
  * Komfortabel
    * Weiterreichen von Datenbankabfragen trivial
      * Wäre mit SQL nur manuell machbar
* Nicht für alles eingesetzt, Altlasten laufen mit bisherigem selbstentwickeltem (Monster-)Wrapper mit knapp 70000 Zeilen Code: jede Spalte ist eine manuell geschriebene Methode …

Logging mit *Log::Log4perl*

* Ersetzt ein altes, manuelles Logging
  * Komfortabler
  * Mehr Funktionalität
  * Einfacher
  * Deutlich flexibler

Dokumentation mit POD

* Umfangreiches POD, u.a. mit API-Dokumentation
  * sehr hilfreich
  * HTML-Variante leicht erstellbar
  * Für Perl-Entwickler alles am gewohnten Ort
* *Pod::ProjectDocs* erzeugt Komplette Dokumentation

Vererbung beim CLI

* *Getopt::Euclid:* Etwas Segen, viel Fluchen
  * Vererbung möglich

```
WAS::CommandLine                # Standard-CLI-Schnittstelle
WAS::CommandLine::JyBox         # Definition für eine Anwendung
WAS::CommandLine::Fonet2PCE     # Definition für eine andere A.
```

  * Damian Conway-Modul!
    * Fehlerhaft, keine Reaktion auf Bugreports/Patches

Die Goldene Kombination

PAR und CPAN::Mini machen die CPAN-Nutzung möglich

Bisher CPAN nicht nutzbar

* Installation durch Firewall nicht möglich
  * Also: Modul manuell runterladen, Abhängigkeit feststellen, diese manuell runterladen, Abhängigkeiten feststellen, …
* Das ist vollkommen unpraktikabel
  * Und oft ein Grund, kein CPAN zu nutzen

Die Lösung: Lokaler Mirror

* Lokaler CPAN-Mirror mit CPAN::Mini
  * via Webserver oder via file:// Protokoll

* Vorsicht mit Viren-Scannern
  * Test-Virus z.B. im ClamAV-Modul
    * Viren-Admin sollte den EICAR-„Virus“ kennen …

Deployment mit PAR

* Installation aller genutzten und eigenen Module auf Zielsystemen kann aufwendig sein
  * Sehr einfache Verteilung mit PAR!
    * Etwas Aufwand bei weiteren Daten-Files
* Weiterer Vorteil: Isolation einzelner Anwendungen
  * Jede kriegt ihr eigenes PAR

Problem Plattformabhängigkeit

- Es lassen sich auch die architekturabhängigen Daten ablegen
  - Pro Plattform Compiler nötig
- In der Praxis Ärger mit Solaris 8 und Solaris 10:
  - Die mit Solaris 10 erstellten Module laufen auch mit Solaris 8
  - Aber: Eine Abhängigkeit benötigte die LibXML2 in anderer Version bei Solaris 8 als bei Solaris 10

Fazit

Der Psyhopath mit der Kettensäge wurde weitgehend abgewehrt, schwebt aber gelegentlich (vor allem bei Altlasten) noch im Hintergrund

Ende.

* Danke.

* Fragen?