Jenseits des Tellerrands, Teil 2: Typisierung in Java und darüber hinaus

Wanderer zwischen den Welten

VON DIERK KÖNIG UND JOHANNES LINK

Schon im letzten Beitrag über „das nächste Java" sind dynamisch typisierte Skriptsprachen als Konkurrenten aufgetaucht. Viele Java-Entwickler kommen mit dynamischer Typisierung in Berührung, ohne dass es ihnen bewusst wäre, bspw. wenn sie in JSPs die JavaServer Pages Standard Tag Library (JSTL) einsetzen. Bei der Diskussion um das richtige Typsystem treffen meist sehr entschiedene Positionen aufeinander – häufig in eher unversöhnlichem Ton. Um jedoch Vor- und Nachteile der unterschiedlichen Typsysteme in der Praxis einschätzen zu können, muss man in beiden Welten zu Hause sein.

Der Begriff „statische Typisierung" wird meist so verstanden, dass zu jeder Referenz im Programmtext einer Sprache ein zugehöriger Typ „statisch" ermittelt werden kann, typischerweise zur Kompilierzeit. Diese Typermittlung kann vom Compiler oder Interpreter dazu genutzt werden, mehr oder weniger starke Restriktionen durchzusetzen, z.B. nur typkonforme Zuweisungen zuzulassen. Je nach Stärke dieser Restriktionen werden Programmiersprachen als „schwach" bis „stark" statisch typisiert bewertet.

Java implementiert vergleichsweise „schwache" Restriktionen auf dem statischen Typsystem. Eine Schwächung ist zum Beispiel das Java-Typecast-Konstrukt, das zur Laufzeit zu einer *ClassCastException* oder einer *NullPointerException* führen kann, d.h., trotz statischer Typermittlung zur Kompilierzeit kann es Typfehler zur Laufzeit geben. Man könnte sogar sagen, dass ein Typecast das statische Typsystem aushebelt. Andere Programmiersprachen, wie z.B. Nice [1], verhindern das völlig und sind deshalb stärker typisiert. Als Extremform statischer Typisierung kann man funktionale Sprachen betrachten (zum Beispiel Haskell [4]) – dort sind Typfehler zur Laufzeit unmöglich. Statische Typisierung bringt u.a. folgende Vorteile:

- *Optimierungen:* Für Referenzen statischen Typs kann zur Kompilierzeit ihr Speicherbedarf ermittelt werden. Damit sind sowohl der Speicherverbrauch als auch der Speicherzugriff optimierbar. Je nach Stärke der Typisierung ist auch der Methoden-Dispatch (welche Implementierung einer Methode in der Klassenhierachie wird tatsächlich aufgerufen?) durch den Compiler und Interpreter in mehr Fällen optimierbar.
- *Compiler-Prüfungen:* Typisierungsfehler können durch einen Compiler gefunden werden. Das sind erfahrungsgemäß Flüchtigkeitsfehler des Programmierers, diese entstehen aber auch häufig durch Änderungen von verwendeten APIs. Das ist sehr nützlich für sprach- bzw. plattformunerfahrene Entwickler, wird aber in seiner Aussagekraft sehr häufig überschätzt; schließlich lässt eine Typprüfung zur Compile-Zeit nur bedingt auf die Typsicherheit zur Laufzeit schließen. Der Mechanismus des Late Binding kann zu Fehlerklassen wie z.B. **Class*Error* und **Method*Error* führen, wenn Kompilierzeit und Laufzeitumgebung dementsprechend verschieden sind.
- *IDE-Support:* IDEs können statische Typinformationen aus dem Programmtext ermitteln und mit dieser Information Mehrwertdienste anbieten: Markierung von Typen oder fehlerhafter Typisierung, automatische Vervollständigungen sowie Verweisverfolgung und Nutzungsanalyse. Darüber hinaus sind einige Refactorings in statisch typisierten Sprachen mit größerer Sicherheit durchführbar als in dynamisch typisierten.
- *Kommunikation:* Die explizite Angabe von Typen in der Methodensignatur erleichtert das Verständnis der Methode und ihrer Parameter, besonders wenn die Parameternamen ungünstig gewählt bzw. nicht verfügbar sind. In nicht typisierten Sprachen mildert man dieses Problem oft dadurch ab, dass der Parametertyp mit in den Namen aufgenommen wird, z.B. *aNameString* statt *String name*.

Jenseits des Tellerrands, Teil 2

Softwareentwicklung ist weit mehr als Programmierung und Programmierung ist weit mehr als das bloße Beherrschen von Programmiersprachen und Werkzeugen. Die Artikel dieser Serie möchten wichtige Themen, aktuelle Fragen und auch grundsätzliche Probleme aufgreifen, beleuchten und „die gängigen Antworten" immer wieder in Frage stellen. Subjektivität ist dabei keineswegs verpönt, sondern das notwendige Salz in der technischen Einheitssuppe.

Subversive Techniken

Kann man sich darauf verlassen, dass man die oben genannten Vorzüge der statischen

Typisierung in einem „echten“ Java-Projekt bekommt? Leider nein. In fast allen Java-Projekten, denen wir begegnen, kommen Verfahren und Technologien zum Einsatz, welche die statische Typisierung unterlaufen.

Eines dieser Verfahren wird von Java selbst angeboten: der Reflection-Mechanismus. Überall dort, wo Klassen anhand ihres Namens geladen werden, Instanzen nicht mit *new*, sondern mit *myClass.newInstance()* erzeugt und Methoden dynamisch aufgerufen werden, wird die statische Typisierung außer Kraft gesetzt. Ein charakteristisches Indiz für diese Verfahren und Technologien ist die Verwendung von Klassennamen in Konfigurationsdateien, wie zum Beispiel in der *web.xml*-Konfiguration für Java EE-Anwendungen, Ants *TaskDefs*, *struts-config.xml*, JSFs *faces-config. xml*, Jettys *server.xml*, Eclipses *plugin. xml* u.v.a.m.

Eine andere typbrechende Technik ist die Bytecode-Instrumentierung, wie sie z.B. im Umfeld von AOP, bei JDO, Hibernate, Spring etc. eingesetzt wird. Hierbei werden der Compiler übergangen und direkt ausführbarer – und potenziell typunsicherer – Bytecode erzeugt. Indizien für die Verwendung dieses Verfahrens sind die Bibliotheken *ObjectWeb ASM* oder *cglib* im Klassenpfad.

Persönliche Einschätzung der Autoren

Dierk König: Wenn ich Java programmiere, versuche ich, die statische Typisierung möglichst wenig aufzuweichen. Wenn ich dynamische Sprachkonstrukte und schlanken, ausdrucksstarken Code haben möchte, setze ich Groovy ein. So bekomme ich zusätzlich noch ein Meta Object Protocol, Closures und selbst definierbare Operatoren. Für meine eher seltenen Ausflüge abseits der Java-Plattform überzeugt mich das dynamische Sprachkonzept von Ruby.
Johannes Link: Meine ersten Jahre objektorientierter Programmierung habe ich überwiegend mit Smalltalk verbracht. Da es damals auch in größeren Teams vernachlässigbar wenig klassische Typprobleme gab, bin ich überzeugt, dass statische Typisierung als Fehlervermeidungstechnik für erfahrene Entwickler kaum eine Rolle spielt. Nach vielen Jahren Java wende ich mich seit kurzem wieder verstärkt dynamisch typisierten Sprachen zu und stelle fest, dass der geringere syntaktische Aufwand den Spaß am Programmieren deutlich vergrößert.

Nimmt man es ganz genau, dann ist jede Verwendung von expliziten Typecasts verdächtig. Schließlich kann man die statische Typisierung in Java ad absurdum führen, indem man in allen Methodensignaturen und Feldern ausschließlich den statischen Typ *Object* verwendet. Wenn man auf Haarspaltereien verzichtet, muss man solchen Code als dynamisch typisiert bezeichnen. Solcherart „dynamisch typisiertes Java“ kann selbst in großen Projekten durchaus seine Berechtigung haben, z.B. zur Vermeidung von Kompilierzeit-Abhängigkeiten oder zur Realisierung möglichst generischer Bauteile. Auch die Java-Collection-Klassen bedienen sich dieses Prinzips – zumindest vor der Zeit generischer Typen in Java 5.

Brauchen wir die statische Typisierung?

Wenn nun die statische Typisierung in so vielen populären Anwendungsfällen umgangen wird, dann geschieht das wahrscheinlich nicht zufällig, sondern aus einer Notwendigkeit heraus. Diese Notwendigkeiten können unterschiedlicher Natur sein, z.B. die Konfigurierbarkeit/Erweiterbarkeit, ohne das System neu kompilieren zu müssen oder eine bessere Abstraktionsfähigkeit und Ausdruckskraft von „Meta“-Code. In diesen Fällen ist statische Typisierung augenscheinlich nicht nur überflüssig, sondern sogar hinderlich.

Ist das Umgehen der statischen Typisierung zwangsläufig gefährlich? Die obigen Beispiele sprechen wohl eher dagegen, schließlich sind auf diese Art viele funktionsreiche und stabile Systeme entstanden. Wichtig scheint uns allerdings, dass man sich besser bewusst macht, wo man das Sicherheitsnetz des Compilers durch andere Verfahren ersetzen muss.

Dynamische Typisierung

Die Sprachen mit dynamischer Typisierung können Restriktionen, die auf Typinformationen beruhen, erst zur Laufzeit durchsetzen. Als Ersatz für die automatischen Prüfungen des Compilers kommen Prüfungen durch automatisierte Tests in Betracht. Nicht umsonst entstand die erste Implementierung einer xUnit-Testsuite für das dynamische typisierte Smalltalk. Das Erstellen von Unit-Tests wird bei dynamischer Typisierung einfacher, da man auf viele Hilfskonstrukte, beispielsweise komplizierte Mock-Objekte, oftmals verzichten kann. Aus dem gleichen Grund wird der Code schlanker und leichter wiederverwendbar.

Dynamisch typisierte Sprachen können durch reduzierte statische Abhängigkeiten sehr viel einfacher inkrementell kompiliert werden; oftmals so schnell, dass die Notwendigkeit, Neukompilierungen zu vermeiden, vollständig entfällt. Darauf können Konfigurationen in der Programmiersprache selbst erfolgen, was zur Homogenität des Systems beiträgt. Außerdem werden durch diesen Geschwindigkeitsvorteil Sprachen möglich, die zwar auf schnellem, kompiliertem Code arbeiten, aber so dynamisch änderbar sind wie interpretierte Sprachen. Die kompaktere Syntax dynamischer Programmiersprachen erleichtert weiterhin die Erstellung „interner domänenspezifischer Sprachen (DSLs)“, wie Martin Fowler [3] ausführt.

Unentschieden bleibt die Frage, ob Refactorings in dynamisch typisierten Sprachen leichter oder schwerer sind. Einerseits erfordern Refactorings in statisch typisierten Sprachen viel weitreichendere Codeanpassungen; das macht sie schwieriger. Andererseits wird diese Arbeit von den Entwicklungsumgebungen sehr gut unterstützt. Dass Refactoring-Tools auch für dynamisch typisierte Sprachen machbar sind, zeigt der Vater aller Refactoring-Browser, der für Smalltalk entwickelt wurde. Seine Verwendung setzt eine aussagekräftige Testsuite voraus. Das sollte allerdings auch bei Refactorings für statisch typisierte Sprachen eine Bedingung sein.

Gemischte Modelle

JavaServer Pages (JSP) haben in diesem Zusammenhang eine interessante Eigenschaft. Wenn man sie vorkompiliert, d.h., den zugehörigen Servlet-Code erzeugen und kompilieren lässt, dann erhält man die Eigenschaften der statischen Typisierung. Wenn man sie lediglich als Ressource der Webapplikation behandelt, werden wie bei der dynamischen Typisierung mögliche Typfehler erst zu Laufzeit gefunden. (Elemente der JSTL sind übrigens immer typunsicher, da diese sich des Java-Reflection-Mechanismus bedienen.)

Auch andere Sprachen wie zum Beispiel Groovy [2] haben diese Eigenschaft. Man kann sie entweder vorkompilieren oder „interpretieren“ (genauer: beim ersten Zugriff kompilieren). Im Fall der Vorkompilierung werden statische Typfehler schon zu diesem Zeitpunkt erkannt. Dabei eröffnet Groovy auch noch einen zweiten Freiheitsgrad: Der Programmierer kann entscheiden, wo er statisch typisieren und wo er den Typ undefiniert lassen möchte. Im letzteren Fall wird implizit der Typ *Object* verwendet. Auf diese Weise kann man sich als Programmierer langsam in die Welt der dynamischen Typisierung hineintasten. Hier ein kleines Beispiel zum Vergleich zwischen Java und Groovy:

```
// Java (pre Java 5):
Map ages = new HashMap();
ages.put("Johannes", new Integer(36));
ages.put("Dierk",  new Integer(37));
int johannesAge = ((Integer)ages.get("Johannes")).
                                        intValue();

// Groovy:
Map ages = ['Johannes':36, 'Dierk':37]
int johannessAge = ages['Johannes']
```

Was die Map-Einträge angeht, ist Java „dynamisch“ typisiert und der Cast könnte (theoretisch) fehlschlagen. Dass solche Verwendungen tatsächlich fehlschlagen, ist sehr selten. Im Groovy-Code sind *ages* und *johannesAge* statisch typisiert (genau wie im Java-Code) und die Werte in der Map sind vom statischen Typ *Object* und dynamischen Typ *Integer*, d.h., beide haben hier die gleiche Typsicherheit zur Compile-Zeit. Groovy ist jedoch deutlich ausdrucksstärker und könnte mit voller dynamischer Typisierung noch knapper formuliert werden:

```
// Real Groovy
ages = ['Johannes':36, 'Dierk':37]
johannesAge = ages.Johannes
```

In diesem Fall sind alle Referenzen vom statischen Typ *Object*, die dynamischen Typen bleiben gleich.

Fazit

Die Sicherheit, die Java durch seine statische Typisierung vermeintlich mit sich bringt, hat in realen Projekten doch erhebliche Lücken. Die statischen Prüfungen des Compilers können die dynamischen Prüfungen einer automatisierten Testsuite nicht ersetzen. Umsicht ist angebracht beim Einsatz von Verfahren und Technologien, die eine Aushöhlung der statischen Typisierung verdecken – andernfalls fällt man leicht auf die trügerische Sicherheit des Compilers herein.

Der offene, klare und automatisiert getestete Einsatz von dynamischen Sprachen ist eine wirkungsvolle Alternative, um in Projekten schnell vorwärts zu kommen. Dabei muss nicht einmal auf die Mächtigkeit der Java-Plattform verzichtet werden: Groovy, Jython und andere machen's möglich.

Dierk König ist Softwareentwickler und Coach bei Canoo in Basel. Ende 2005 erscheint sein Buch „Groovy in Action“. Feedback ist willkommen unter dierk.koenig@canoo.com.
Johannes Link ist Softwareentwickler und Projektleiter bei der andrena objects ag in Karlsruhe. Auf zahlreiches Feedback und Anregungen zur Serie freut er sich unter johannes.link@andrena.de.

Links & Literatur

[1] nice.sourceforge.net/safety.html
[2] groovy.codehaus.org
[3] martinfowler.com/articles/languageWorkbench.html
[4] www.haskell.org