Smells like Java

Softwareentwicklung mit Natural

Stefan Macke, Markus Weßjohann

Die Softwareentwicklung in Natural unterscheidet sich stark von der Softwareentwicklung in Java. Viele bekannte Tools gibt es im Natural-Umfeld nicht und die Vorgehensmodelle sind sehr individuelle Lösungen. In diesem Artikel möchten wir unseren Ansatz für eine moderne – an Java orientierte – Natural-Entwicklung vorstellen und damit einhergehende Probleme erörtern.

Natural, eine Sprache der 4. Generation

Natural ist eine Programmiersprache, die vor über 40 Jahren von der Software AG für den Mainframe entwickelt wurde, inzwischen wurde sie aber auch auf diverse andere Plattformen, wie Linux und Windows, portiert. Ursprünglich diente Natural als Abfragesprache für das Datenbanksystem Adabas aus demselben Hause. Die Verzahnung zwischen Natural und Adabas ist daher sehr hoch, was eine schnelle Entwicklung datenbankgestützter Anwendungen ermöglicht. Die Syntax ähnelt COBOL oder ABAP und der Sprachumfang ist relativ gering und daher schnell erlernbar. Einsatzbereiche sind klassischerweise dialoggetriebene Terminalanwendungen und die Batchverarbeitung von Massendaten, zum Beispiel in Banken und Versicherungen.

Natural ist eine 4GL-Sprache, das heißt, die Anweisungen bilden recht abstrakte Funktionen ab, die auch von einem fachlichen Mitarbeiter verstanden werden könnten. Sie ist, ähnlich wie Java, statisch typisiert und wird zunächst kompiliert und dann interpretiert. Im Gegensatz zu Java wird in Natural jedoch nicht objektorientiert, sondern prozedural entwickelt, und die Laufzeitumgebung liegt auf dem Server.

```
DEFINE DATA
* ein-/ausgehende Parameter
PARAMETER
  01 P-ANZAHL (I4) /* Integer, 4 Bytes
  01 P-PERSONEN (A20/1:10) /* Array (Länge 10) aus Strings (Länge 20)
* externe Variablendefintion
LOCAL USING PERSDEF
* Definition lokaler Variablen
LOCAL
 01 #INDEX (I4)
END-DEFINE
RESET #INDEX
READ PERSONEN /* lesender Datenbankzugriff
 IF #INDEX > 10 OR #INDEX = P-ANZAHL
     ESCAPE BOTTOM /* wie break
 END-IF
 ADD 1 TO #INDEX
 P-PERSONEN(#INDEX) := PERSONEN.NAME
END-READ
END
```

Listing 1: Aufbau eines Natural-Moduls

Der typische Aufbau eines Natural-Moduls ist in Listing 1 zu sehen. Es werden alle genutzten Variablen im Kopf des Moduls definiert und sind damit im gesamten Modul global gültig. Der einzige Einstiegspunkt in die Programmlogik folgt nach der Variablendefinition. Eine Unterteilung in sogenannte *Subroutines* (vergleichbar mit privaten Methoden) ist möglich.

Jedes Modul hat eine Parameterliste, in der alle Ein- und Ausgabewerte definiert sind. Schnittstellen zwischen Modulen werden durch ihre verwendeten Parameter definiert, daher kann man die Variablendefinitionen auch in gemeinsam genutzte sogenannte *Parameter Data Areas* auslagern.

Der Datenbankzugriff mit Natural ist sehr performant und unkompliziert (s. Listing 1), daher wird gerne häufiger auf die Datenbank zugegriffen, anstatt Werte im Speicher zu halten. Da es nicht wie in Java möglich ist, die Daten und zugehörige Funktionen zu kapseln, wird das fachliche Modell häufig in mehreren Modulen redundant implementiert. Eine Abstraktion von der Datenbank ist in vielen Anwendungen nicht umgesetzt und es gibt keine Trennung von Datenbank-, Programm- und Oberflächenlogik.

Für die Benennung von Natural-Modulen stehen maximal 8 Zeichen zur Verfügung. Für Subroutines sind bis zu 32 Zeichen erlaubt. Beide können in sogenannten *Libraries* organisiert werden, die ebenfalls nur 8 Zeichen im Namen und keine weitere Unterorganisation erlauben. Abhängigkeiten zwischen Libraries können eingerichtet werden, indem sogenannte *Steplibs* konfiguriert werden. Jedes Modul aus einer Library kann alle Module aus seinen Steplibs verwenden.

Herausforderungen

Die Softwareentwicklung mit Natural unterscheidet sich von der Softwareentwicklung in Java in einigen zentralen Punkten.

Server versus lokal

Ein wichtiger Unterschied ist, dass die Natural-Programme in derjenigen Natural-Umgebung erstellt und übersetzt werden müssen, in der sie später auch ausgeführt werden, also auf dem Server. Auf dem Mainframe werden dabei sowohl der Quellcode als auch die Kompilate in einer Datenbank abgelegt. Eine lokale Entwicklung mit Quellcode auf dem Client des Entwicklers ist somit nicht möglich.

Außerdem arbeiten alle Entwickler gleichzeitig auf einer einzigen gemeinsamen Quellcode-Basis. Damit es dabei nicht zu Konflikten kommt, könnte jeder Entwickler in seiner eigenen Library entwickeln und die gemeinsame Code-Basis aus einer zentralen Steplib nutzen, ähnlich zum Classpath, in dem die JARs des Entwicklers vor denen der gesamten Anwendung liegen. Es werden also erst die Module aus der eigenen Library angezogen, bevor ein Modul aus der gemeinsamen Code-Basis verwendet wird. Bei diesem Verfahren kommt es durch

die Arbeit mehrerer Entwickler am selben Modul häufig zu Änderungskonflikten oder Problemen mit Modulabhängigkeiten. Diese müssen meist organisatorisch gelöst werden, da eine vollständige technische Unterstützung fehlt.

Durch die Remote-Entwicklung wird eine Versionsverwaltung erschwert, da der Quellcode nicht lokal gespeichert wird und die üblichen Versionsverwaltungswerkzeuge keinen Zugriff auf ihn haben. In der älteren Entwicklungsumgebung *Natural Studio* ist eine Versionsverwaltung mit CVS oder SVN möglich, aber immer nur optional und mit Zusatzaufwand für den Entwickler verbunden. Einige Entwickler programmieren sogar direkt in der Terminaloberfläche ohne Unterstützung durch eine IDE. Die Code-Generierung aus anderen Programmiersprachen ist ebenfalls schwierig, da die Natural-Module nicht lokal abgelegt werden können, sondern zum Server übertragen und der Natural-Umgebung bekannt gemacht werden müssen.

Tools

Der Compile-Vorgang für Natural besteht aus mehreren Schritten. Zunächst müssen die vorhandenen Module mittels des Kommandozeilenwerkzeugs `ftouch` der Laufzeitumgebung bekannt gemacht werden, wobei dies für aktuell in der Laufzeitumgebung erstellte Module entfallen kann. Danach wird in der Natural-Session der Befehl `CATALL` ausgeführt, der alle Module für eine einzelne Library kompiliert. Dabei werden Library-übergreifende Modulabhängigkeiten nicht berücksichtigt und die Reihenfolge der kompilierten Libraries muss daher manuell durch den Entwickler vorgegeben werden.

Als Entwicklungsumgebung für Natural kommen drei Varianten in Frage: der Terminaleditor, Natural Studio und eine Eclipse-basierte IDE. Die ersten beiden Editoren bieten nur rudimentäre Unterstützung bei der Entwicklung, wie Syntax-Highlighting oder einen Formatter. Eine Tool-Unterstützung für Refactorings, wie man sie aus dem Java-Umfeld kennt, ist in keiner Entwicklungsumgebung vorhanden.

Weiterführende Entwicklungstools wie zur statischen Codeanalyse, Testabdeckung (Code Coverage) oder eine Integration in einen automatischen Build-Prozess sind entweder nicht vorhanden oder werden nur von sehr wenigen Unternehmen kostenpflichtig angeboten. Eine Ausnahme stellt das Komponententest-Framework NatUnit dar, das von den Autoren bereits vor längerer Zeit als Open Source veröffentlicht und von einigen anderen Unternehmen adaptiert wurde.

Auch für das Deployment von Modulen in andere Umgebungen gibt es kein einheitliches Verfahren. In einigen Unternehmen werden Kompilate manuell per Copy/Paste auf die Produktion übertragen. Andere Unternehmen haben hierfür ihre individuellen Verfahren und Werkzeuge entwickelt, die sich nicht ohne Weiteres auf andere Unternehmen übertragen lassen.

Community

Die Anwender- und Entwicklergemeinde ist im Netz kaum präsent beziehungsweise überwiegend in geschlossenen Foren aktiv. Die Dokumentation der Programmiersprache Natural ist nicht frei verfügbar, sondern nur für registrierte Kunden zugänglich. Kurse oder Vorträge sind fast ausschließlich nur beim Hersteller belegbar. Die wenigen Treffen der User-Groups werden größtenteils von der Software AG organisiert und betreut. Bücher und andere Informationsquellen sind kaum verfügbar und schwierig zu finden, und auch die Veröffentlichung von selbstentwickelten Natural-Werkzeugen ist in der Community nicht verbreitet. Die Gründe dafür sind, dass zentrale Austauschplattformen fehlen oder vorhandene nicht genutzt werden und die Lösungen sehr unternehmensspezifisch sind.

Daher kämpft die Natural-Community schon seit Längerem mit dem Problem, Entwicklernachwuchs zu finden. Dadurch werden moderne Entwicklungsprinzipien und Vorgehensweisen wie Test-Driven Development oder Pair Programming nur äußerst langsam von der Entwicklergemeinde angenommen.

Ein moderner Entwicklungsprozess für Natural

Trotz der beschriebenen Herausforderungen bei der Natural-Entwicklung haben wir versucht, uns mit der Entwicklungsumgebung und dem Entwicklungsprozess an die im Java-Umfeld etablierten Methoden anzunähern. Dazu war es zunächst notwendig, jedem Entwickler seine individuelle Laufzeitumgebung (den sogenannten `FUser`) zur Verfügung zu stellen, in der er ohne Beeinträchtigung der Kollegen ent-

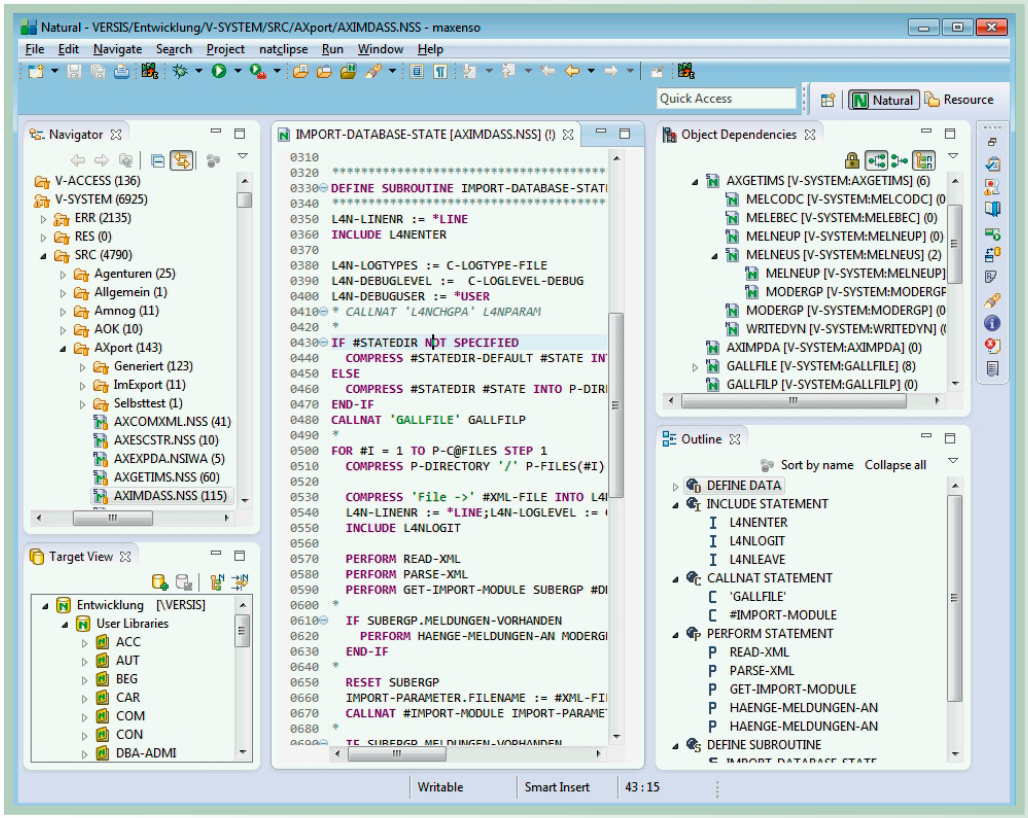

Abb. 1: natclipse

wickeln kann. In dieser Laufzeitumgebung ist der gesamte Quellcode enthalten.

natclipse

Im Anschluss wurde dann die Entwicklungsumgebung modernisiert, indem auf die Eclipse-basierte IDE natclipse des Herstellers innoWake umgestellt wurde. Die Alternative der Software AG, NaturalONE, hat sich nach längerer Testphase für uns als nicht praktikabel erwiesen. Der zentrale Unterschied zum vorherigen Prozess ist, dass der gesamte Quellcode nun lokal auf dem Client des Entwicklers verfügbar ist und nur noch Änderungen sowie deren Abhängigkeiten auf den Server übertragen werden. Dadurch wird der Einsatz von Entwicklungswerkzeugen ermöglicht, die Zugriff auf den Quellcode benötigen. Das wichtigste Werkzeug ist sicherlich die Versionsverwaltung, in unserem Fall mit Git. Das Vorgehen bei der Entwicklung hat sich dadurch stark verändert, da nun nicht mehr der Quellcode auf dem Server führend ist, sondern derjenige in der Versionsverwaltung.

Die Produktivität der Entwickler konnte durch die Umstellung auf natclipse gesteigert werden, da zusätzlich zu den aus Natural Studio bekannten Features wie Syntax-Highlighting und Code-Formatierer nun zeitsparende Funktionen wie Autovervollständigung (Code Completion) und Code-Templates verfügbar sind. Außerdem bietet die IDE deutlich bessere Möglichkeiten in der Code-Navigation und der Verwaltung von Abhängigkeiten, die in einem separaten View für jedes Modul sichtbar sind (s. Abb. 1).

Unsere größte Library enthält fast 5.000 Module. In anderen Unternehmen sind jedoch mehr als 10.000 Module keine Seltenheit. Zur besseren Übersicht über den Quellcode ist es nun möglich, die Libraries mit Unterverzeichnissen zu strukturieren, was das Auffinden von Modulen schon deutlich erleichtert (s. Abb. 1). Eine noch größere Verbesserung bietet jedoch der Einsatz des Eclipse-Plug-ins MyLyn, mit dem es möglich ist, nur die Quellcode-Dateien anzuzeigen, die zur Bearbeitung der aktuellen Aufgabe benötigt werden. Durch die einfache Integration in Ticketsysteme – in unserem Fall in Redmine – können die individualisierten Ansichten auch an andere Entwickler weitergegeben werden. Außerdem haben wir durch einen Commit-Hook in Git sichergestellt, dass jeder Commit einen Bezug zu einem Redmine-Ticket hat, und somit lückenlos bis auf Ebene einzelner Code-Zeilen nachvollziehbar ist, wer wann welche Änderung gemacht hat und auf welcher fachlichen Anforderung sie basiert.

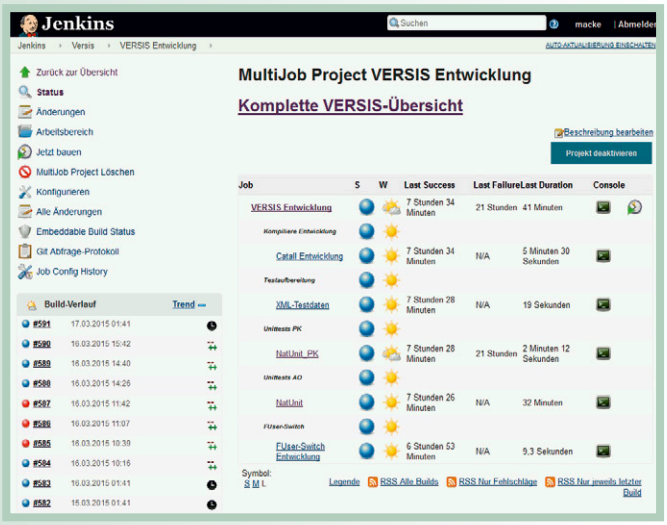

Abb. 2: Jenkins, Build

Build-Server

Der nächste Schritt nach der Umstellung auf die lokale Entwicklung war die Automatisierung des Build-Prozesses mit Einbindung eines Build-Servers – in unserem Fall Jenkins. Hierfür war es zunächst notwendig, den Compile-Prozess für Natural zu automatisieren. Dafür war eine umfangreiche Programmierung notwendig, die als Ergebnis eine Java-Applikation hatte, die den kompletten Compile-Prozess für eine beliebige Zielumgebung durchführt (s. Abb. 2).

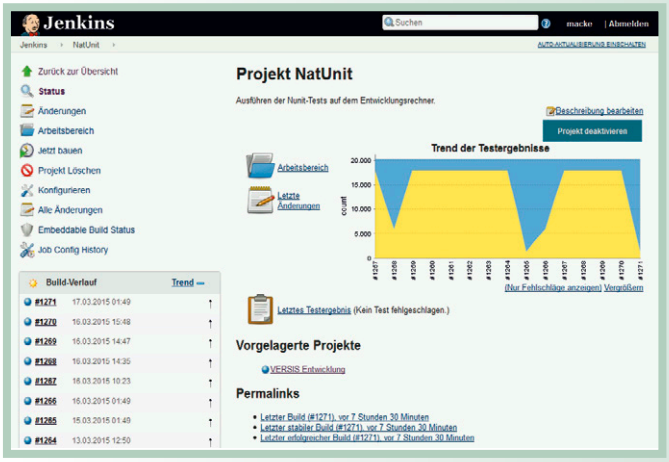

Abb. 3: Jenkins, Tests

Es wird zunächst der `CATALL` in einer separaten Build-Umgebung für alle Libraries durchgeführt, danach werden die Komponententests vorbereitet und ausgeführt und erst bei erfolgreichem Test wird die Zielumgebung aktualisiert. Hierfür wurde unser Komponententest-Framework NatUnit verwendet, das um eine Schnittstelle zur Integration in Jenkins erweitert wurde, damit die Testergebnisse wie von JUnit gewohnt ausgewertet werden können (s. Abb. 3). Der Build wird automatisch nach jedem Push ins Git-Repository gestartet, langlaufende Integrationstests werden jedoch zeitgesteuert nächtlich ausgeführt.

Der nächste Schritt ist nun die Automatisierung des Deployments auf die Test- und Produktionsumgebung. Um sicherzustellen, dass nur projektspezifische Änderungen an gemeinsam genutzten Modulen veröffentlicht werden, soll die Basis hierfür der Gitflow-Workflow von Atlassian sein. Zurzeit befinden wir uns noch in der Analyse der notwendigen Anpassungen und Ergänzungen der technischen Infrastruktur.

Fazit

Abschließend lässt sich festhalten, dass ein moderner Entwicklungsprozess auf Basis von Natural zwar umsetzbar, aber mit viel Eigenleistung verbunden ist. Die vorhandenen Tools waren für unsere Anforderungen entweder nicht geeignet oder zu kostenintensiv. Es waren einige Anpassungen an der Infrastruktur notwendig, die hier nicht im Detail erläutert werden konnten, da unser Fokus auf der Praxis und Lesbarkeit und nicht auf der Vollständigkeit und Genauigkeit der Beschreibung lag. Falls Interesse an den Details oder am grundsätzlichen Gedankenaustausch besteht, können die Autoren gerne kontaktiert werden.

Die größte Herausforderung bei der Einführung des neuen Entwicklungsprozesses war aus Sicht der Autoren trotz der technischen Hindernisse vielmehr das organisatorische Umdenken. Die Umstellung der Entwickler, die jahrelang im Terminal auf dem Server entwickelt haben, auf eine durch die Versionsverwaltung gesteuerte Arbeitsweise wurde langfristig vorbereitet und durch Schulungen begleitet. Auch der durchgängige Einsatz des Komponententest-Frameworks NatUnit wird aktuell noch durch ständiges Coaching verbessert. Die fehlende Verbreitung moderner Entwicklungsprinzipien im Natural-Umfeld erschwert die Umstellung ebenfalls, da Lösungen für viele aus der Java-Entwicklung bekannte Entwurfsmuster, wie eine Factory zur Auflösung von Modulabhängigkeiten, erst aufwendig für Natural konzipiert werden mussten.

Die IDE natclipse macht die Natural-Entwicklung deutlich produktiver und weniger fehleranfällig. Außerdem arbeiten sprachübergreifende Entwicklerteams nun mit dem gleichen Werkzeug und entwickeln somit langfristig vielleicht ein besseres Verständnis für die jeweils andere Programmiersprache. Dennoch sollte man bei der Projektplanung die Besonderheiten von Natural-Projekten kennen, da insbesondere Tools für Refactorings oder die Ermittlung der Testabdeckung fehlen. Die Entwicklungsgeschwindigkeit könnte darunter leiden.

Insgesamt ist festzuhalten, dass eine moderne Softwareentwicklung weniger von Programmiersprachen und Werkzeugen abhängt, sondern vielmehr von der Einstellung der Entwickler. Wenn man den Blick über den Tellerrand wagt, lassen sich Konzepte und Ideen aus anderen Sprachen auch auf solche übertragen, die scheinbar nicht dafür geeignet sind.

Links

[Git] `http://git-scm.com/`
[Gitflow] `https://www.atlassian.com/git/tutorials/comparing-workflows`
[Jenkins] `http://jenkins-ci.org/`
[JUnit] `http://junit.org/`
[Mylyn] `http://eclipse.org/mylyn/`
[natclipse] `http://www.innowake.de/natclipse-27-de.html`
[NatUnit] `http://sourceforge.net/projects/natunit/`
[Natural] `http://www.softwareag.com/corporate/products/adabas_natural/natural/overview/default.asp`
[NaturalONE] `http://techcommunity.softwareag.com/ecosystem/communities/public/adanat/products/naturalone/downloads-community-edition/index.html`
[Redmine] `http://www.redmine.org/`

Stefan Macke ist seit 2003 bei der ALTE OLDENBURGER Krankenversicherung AG beschäftigt und als Softwarearchitekt und -entwickler tätig. Sein Aufgabenschwerpunkt liegt im Bereich der serviceorientierten Architektur und der Ausbildung des Entwicklernachwuchses. Außerdem ist er Kernentwickler des NatUnit-Projekts.
E-Mail: stefan.macke@alte-oldenburger.de

Markus Weßjohann ist seit 2001 Softwareentwickler und seit 2008 bei der ALTE OLDENBURGER Krankenversicherung AG im Bereich der Tarifmodellierung tätig. Vor Natural entwickelte er in mehreren Projekten mit unterschiedlichen Programmiersprachen. Außerdem ist er Kernentwickler des NatUnit-Projekts.
E-Mail: markus.wessjohann@alte-oldenburger.de