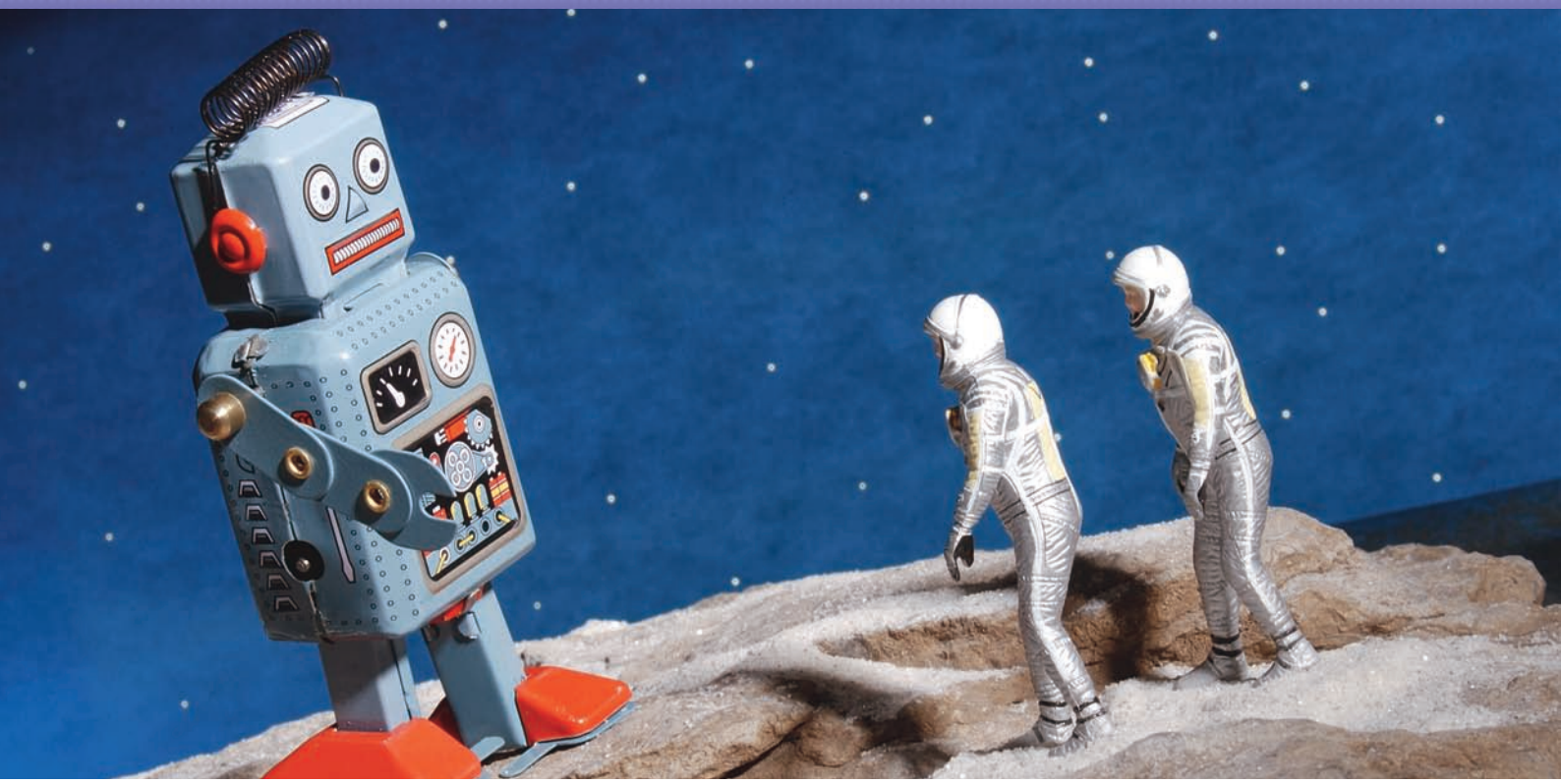

## Generator-Plug-ins mit Eclipse und openArchitectureWare entwickeln

# Lego modellgetrieben

>> PETER FRIESE UND FRANK ZIMMERMANN

Viele Softwaregeneratoren sind nicht in eine Modellierungsumgebung eingebettet und werden über die Kommandozeile gesteuert. Doch es geht auch anders: Anhand des Aufbaus einer vollintegrierten Entwicklungsumgebung für Lego Mindstorms Roboter lässt sich demonstrieren, wie.

Der Lego Mindstorms NXT [1] wurde 2006 als Nachfolger von Lego Mindstorms RCX der Öffentlichkeit vorgestellt. Durch seinen Preis – unter 300 Euro – wird der NXT im Hobbybereich gerne genutzt, aber auch Hochschulen setzen ihn in der Lehre ein. Die NXT Generation verfügt über 256 KB Flash Memory und 64 KB RAM. Verglichen mit aktuellen Handys ist das ein sehr kleiner Speicher, was gewisse Herausforderungen mit sich bringt. Im Basispaket sind drei Motoren und vier Sensoren enthalten. Lego Mindstorms NXT wird mit einem eigenen Betriebssystem ausgeliefert. Programme werden in der vom MIT mitentwickelten Programmiersprache NXT-G geschrieben. Die Programmierumgebung erlaubt die visuelle Kombination und Konfiguration von Programmblöcken. Erweiterungen sind in Form von angepassten Programmblöcken im Internet verfügbar oder können mit der kommerziellen Umgebung LabView selbst programmiert werden.

Ein Beispiel für eine NXT-G-Anwendung ist der Line Follower [2]. Ein mobiler Roboter soll einer schwarzen Linie folgen (Abb. 1). Dazu nutzt er einen Lichtsensor, der die Helligkeit unter sich misst. Ist der Helligkeitswert niedrig, so befindet sich der Roboter über der Linie, und der Roboter kann vorwärts fahren. Ist der Helligkeitswert

Abb. 1: Mindstorms Linefollower

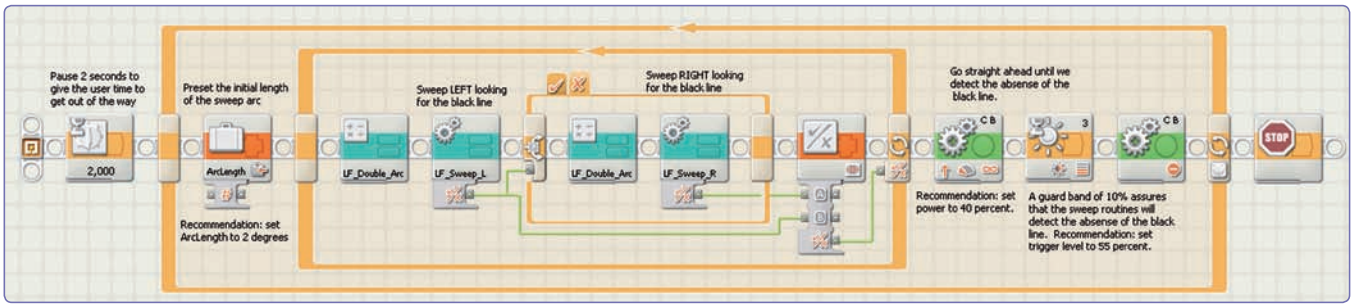


Abb. 2: Line Follower mit NXT-G

hoch, so ist der Roboter nicht über der Linie. Da allerdings unklar ist, ob sich der Roboter links oder rechts der Linie befindet, muss das Programm den Roboter in sich aufschaukelnden Rechts-/Linksbewegungen um seine eigene Achse drehen, bis der Roboter die Linie wiedergefunden hat. Abbildung 2 zeigt das Programm in NXT-G.

## leJOS Java Operating System

Neben der Lego-eigenen Umgebung gibt es noch zahlreiche weitere Ansätze, den NXT mit herkömmlichen Programmiersprachen wie etwa C, C#, Phyton usw. zu programmieren. Für die Sprache Java gibt es das Betriebssystem leJOS [3], das eine virtuelle Java Maschine auf dem NXT, samt zugehöriger Infrastruktur, wie angepasste Klassenbibliotheken und native Methoden zum Ansteuern der Hardware, umfasst. Der prominenteste Einsatz von leJOS ist wahrscheinlich der Lego-Roboter Jitter, der 2001 auf der ISS die Erde umrundete und in der Lage war, Gegenstände in der Schwerelosigkeit einzusammeln [4]. Als typsichere objektorientierte Sprache eignet sich Java besonders gut für die Programmierung von autonomen Robotern, denn das Fehlen einer geeigneten Testinfrastruktur kann zum Teil durch die starke Typisierung kompensiert werden. Andererseits stellt Java als objektorientierte Programmiersprache umfangreiche Mechanismen zur Entwicklung komplexer Anwendungen zur Verfügung. Diese Eignung ist nicht zufällig, sondern entspringt der ursprünglichen Intention, Java zur Programmierung eingebetteter Systeme zu nutzen [5]. Aufgrund der geringen Speichergröße ist zurzeit unter leJOS nur eine eingeschränkte Klassenbibliothek verfügbar. Weil keine Collection-Klassen vorhanden sind, muss man mit Arrays vorliebnehmen. Auch gibt es in leJOS das Java-Konzept der Classloader nicht, also Klassen, die dynamisch Klassen nachladen können. Stattdessen müssen die *.class*-Dateien durch einen statischen Link-Prozess in eine *.nxj*-Datei gebunden werden.

## Zustandsautomaten

In der Entwicklung von embedded Systems haben sich Zustandsautomaten als besonders geeignete Technik erwiesen (vgl. [6], Kapitel 13). In der Informatikausbildung stellen Automatenmodelle die Grundlage für viele Anwendungen dar, von der theoretischen Informatik bis hin zur Spezifikation von Kommunikationsprotokollen in der Nachrichtentechnik. Automatenmodelle bestechen durch ihre intuitive Verständlichkeit und durch ihre Fähigkeit, komplexe Probleme zu strukturieren. In der Umsetzung von Automaten jedoch zeigt sich schnell, dass die zweidimensionale graphenbasierte Struktur von Automaten nur unter deutlichem Verlust an Ausdruckskraft in die eindimensionale Struktur von Programmtext übertragen werden kann. Um das Automatenmodell für die Programmierung von Lego Mindstorms sinnvoll nutzen zu können, ist eine Entwicklungsumgebung wünschenswert, die die Automatenmodelle direkt in Programmcode übersetzt. Im Rahmen modellgetriebener Softwareentwicklung werden in der Regel nur die architekturzentrierten Bestandteile generiert. Aufgrund der geringen Komplexität der Lego-Anwendungen kann man in Betracht ziehen, die gesamte Anwendung aus dem Modell zu generieren. Dazu müssen die Modelle um Java-Codeschnipsel (Snippets) ergänzt werden. In diesen Snippets werden Variablen deklariert, die Aktoren und Sensoren abgefragt oder eigene Klassen eingebunden. Letzteres stellt sicher, dass komplexe Codeteile in die Anwendung mit einbezogen werden können, ohne das Modell unnötig aufzublähen. Für das EMF-Modell sind die Snippets nur

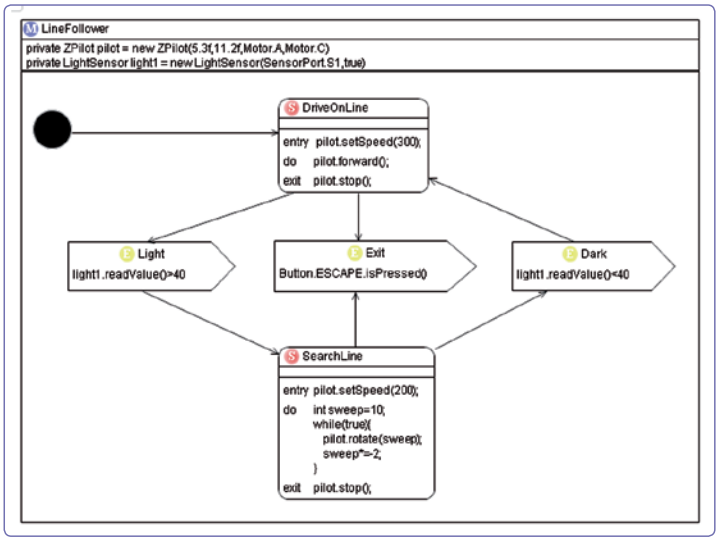


Abb. 3: Linefollower Statemachine

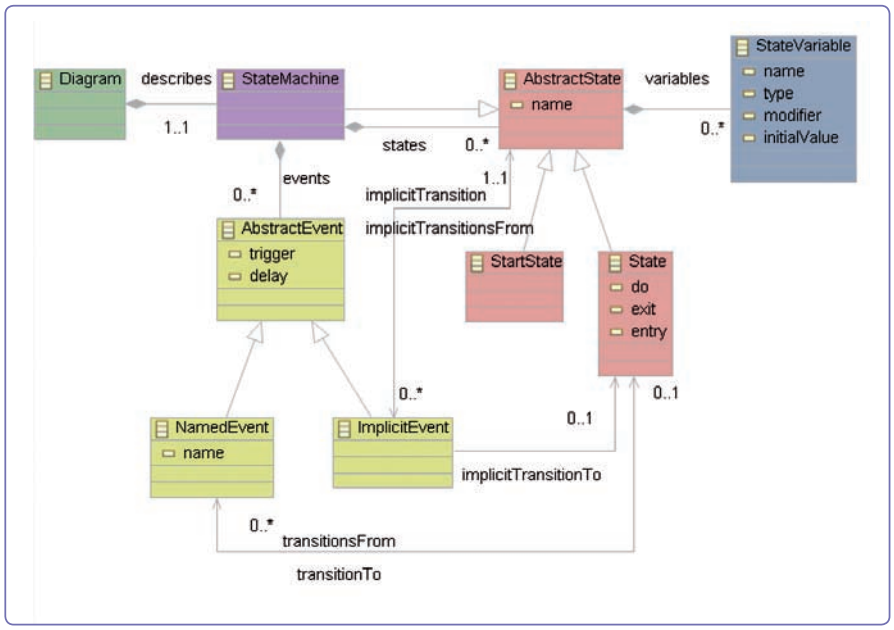


Abb. 4: Metamodell des NXT-Zustandsautomaten

Texte, erst ein Generator kopiert sie an die richtige Stelle in einer Java-Klasse. Fehler in den Snippets werden so zwar erst bei der Generierung erkannt, da die Generierung jedoch nahtlos in den Modellierungsprozess eingebaut ist, ist für ein einigermaßen direktes Feedback an den Benutzer gesorgt. Die kurzen Turnaround-Zeiten tun ein Übriges, um die Akzeptanz bei den Entwicklern zu erhöhen.

Der Vorteil der höheren Abstraktionsstufe wird anhand der Lösung des Line-Follower-Problems deutlich. Abbildung 3 zeigt die Version der Robotersteuerung mit Zustandsautomaten. Der Zustandsautomat ist im Wesentlichen selbsterklärend, das Modell dient gleichzeitig zur Dokumentation und Ausführung. Die verwendeten Snippets sind einerseits Java-Anweisungen, wie *pilot.forward()* zum Vorwärtsfahren oder *pilot.stop()* zum Stoppen beider Motoren, und andererseits Bedingungen, wie *light.readValue()>40* zum Überprüfen der vom Lichtsensor gemessenen Helligkeit.

### MDSD-Werkzeugkette

Zur Entwicklung eines Development Toolkits mit MDSD kann man auf eine Reihe von Open-Source-Werkzeugen zurückgreifen. Auf die Evaluation verschiedener Alternativen soll hier nicht eingegangen werden. Zur Speicherung der Modelle dient das Eclipse Modeling Framework [7]. Statt auf die naheliegende Möglichkeit zurückzugreifen, die UML Statecharts zu nutzen, wurde eine domänenspezifische Sprache entwickelt. Das reduziert den Aufwand für die Generierung erheblich, denn das Metamodell der UML ist für die Generierung von NXT-Anwendungen zu komplex. Abbildung 4 zeigt das domänenspezifische Metamodell. Für die Erstellung der Modelle wird ein GMF-basierter [8] grafischer Editor benutzt. openArchitectureWare [9] stellt mit Check und Xpand geeignete Werkzeuge zur Modellvalidierung und Generierung von Java-Anwendungen aus EMF-Modellen bereit.

### Entwicklung eines Generators

Der Generator verarbeitet das Model der Statemachine und erzeugt daraus Java-Code, der auf der leJOS-Plattform ausgeführt werden kann. Dieser Code stützt sich auf ein selbst entwickeltes

**Listing 1**

```
public void configure() throws CoreException {
 prepareClassPath();
 createDirectories();
 addBuilders();
}

private void addBuilders() throws CoreException {
 IProjectDescription desc = project.getDescription();
 ICommand[] commands = desc.getBuildSpec();
 ICommand[] newCommands = new
ICommand[commands.length + 2];
 System.arraycopy(commands, 0, newCommands, 1,
                                  commands.length);
 // Insert generator as first
 ICommand command = desc.newCommand();
 command.setBuilderName(LejosBuilder.BUILDER_ID);
 newCommands[0] = command;
 // Insert linker as last
 command = desc.newCommand();
 command.setBuilderName(LejosLinker.BUILDER_ID);
 newCommands[newCommands.length - 1] = command;
 desc.setBuildSpec(newCommands);
 project.setDescription(desc, null);
}
```

**Listing 2**

```
<workflow>
 <property name="metaModel" value=
        "lejos.stateMachine.StateMachinePackage" />

 <!-- load model and store it in slot ‚model' -->
 <component class="org.eclipse.mwe.emf.Reader">
  <uri value="platform:/resource/${model}" />
  <modelSlot value="model" />
 </component>

 <!-- semantical checks -->
 <component class="org.openarchitectureware.
                                  check.CheckComponent">
  <metaModel id="mm"
   class="org.openarchitectureware.type.emf.
                                  EmfMetaModel">
   <metaModelPackage value="${metaModel}" />
  </metaModel>
  <checkFile value="check::statemachine" />
  <expression value="model.eAllContents" />
 </component>

 <!-- generate code -->
 <component class="org.openarchitectureware.
                                  xpand2.Generator">
  <metaModel id="mm"
   class="org.openarchitectureware.type.emf.
                                  EmfMetaModel">
   <metaModelPackage value="${metaModel}" />
  </metaModel>

  <expand value="template::statemachine::main FOR
                                  model" />
  <outlet path="${basedir}/src-gen"/>
 </component>
</workflow>
```

Framework, das die Arbeit mit Threads und Nebenläufigkeit vereinfacht und somit der Best Practice „Generiere gegen eine mächtige Plattform“ entspricht [10]. Eines der Ziele bei der Entwicklung des LeJOS Statemachine Development Toolkits war eine möglichst intuitive Bedienung. Die Tatsache, dass die vom Benutzer modellierten Zustandsautomaten durch einen Generator in Java-Code transformiert werden, sollte so gut wie möglich vor dem Anwender verborgen werden. Idealerweise sollte der gerade bearbeitete Zustandsautomat bereits beim Speichern in leJOS-kompatiblen Java-Code übersetzt werden. Ein modellierter Zustandsautomat für den Mindstorms Brick wird in fünf Schritten ausgeführt:

1. Generieren des Java-Programms (*.java*-Datei) aus dem Modell
2. Übersetzen des Java-Programms (*.class*-Datei)
3. Binden des Java-Programms (*.nxj*)
4. Upload der *.nxj*-Datei auf den Brick
5. Starten des Programms

Die Schritte 4 und 5 müssen explizit vom Entwickler angestoßen werden. Wie die Schritte 1 bis 3 mithilfe von Eclipse Plug-ins transparent für den Statemachine-Entwickler gestaltet werden können, wird im Folgenden beschrieben.

## Incremental Project Builder

Zur Automatisierung von Abläufen, die beim Speichern von Dateien durchgeführt werden sollen, werden im Eclipse-Umfeld üblicherweise Incremental Project Builder eingesetzt [11]. Eclipse selbst macht intensiven Gebrauch von diesem Feature. So wird z.B. Java-Code durch einen Incremental Project Builder bereits beim Speichern in Bytecode übersetzt. Ein Projekt kann durchaus von mehreren Buildern bearbeitet werden. Welche Builder dabei in welcher Reihenfolge gestartet werden, wird über die Nature des Projekts festgelegt. Projekte können wiederum mehrere Natures haben. Durch diese Gliederung können recht komplexe Buildprozesse umgesetzt werden.

Im gegebenen Problem werden korrespondierend zu den Schritten 1 bis 3 drei Builder benötigt. Während zur Umsetzung von Schritt 2 der JavaBuilder mit

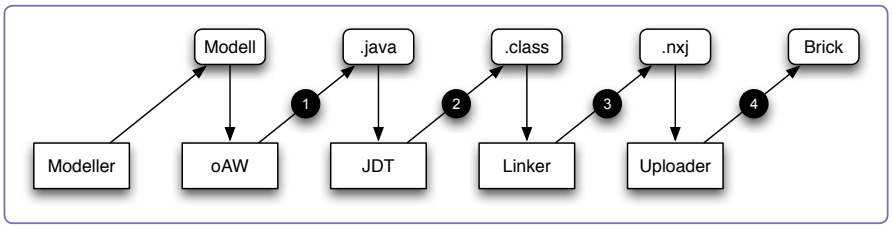


Abb. 5: Entwicklungsprozess

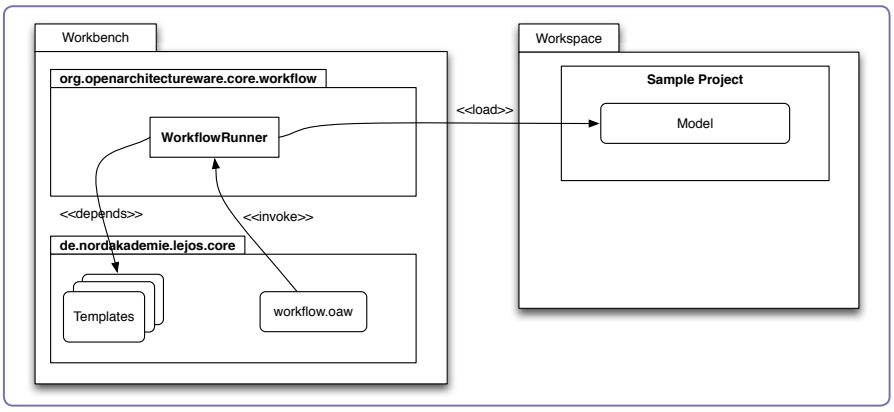


Abb. 6: Struktur des Generators

der JavaNature assoziiert ist, werden Schritt 1 und 3 mit einer „neuen“ LeJOS-Nature verknüpft.

Beim Konfigurieren der leJOSNature sind nun drei Dinge zu erledigen. Erstens sind die erforderlichen Bibliotheken in den Classpath des Projekts zu bringen. Insbesondere muss das Java Runtime Environment durch das NXT-spezifische Runtime Environment ersetzt werden.

Zweitens müssen Verzeichnisse für die Modelle und für die generierten Klassen angelegt werden. Und schließlich müssen die Builder in der richtigen Reihenfolge hinzugefügt werden. Listing 1 zeigt einen Codeausschnitt aus der LeJOSNature-Klasse, in dem deutlich wird, wie man Builder zu einem Projekt hinzufügen kann.

## Ein Builder für den Generator

Wenngleich openArchitectureWare bereits über eine gute Integration in Eclipse verfügt und der für die Zustandsautomaten entwickelte Generator prinzipiell über das Kontextmenü des Workflows (Run As ... oAW Workflow) gestartet werden könnte, muss im Sinne der gewünschten Transparenz die Transformation des Modells in Java-Code durch einen Incremental Project Builder gestartet werden. Diese Model-to-Code-Transformation wird durch den in Listing 2 abgebildeten oAW Workflow gesteuert. Zunächst wird das Modell geladen, wobei der Pfad zur Modeldatei relativ zur Plattform (d.h. dem Workspace) angegeben wird. Anschließend wird es auf etwaige Fehler inhaltlicher Art überprüft (hierzu wird die oAW-Sprache „Check“ eingesetzt). Sofern keine Constraint-Verletzungen festge-

stellt wurden, wird das Model an den Codegenerator übergeben, der daraus Java-Code erzeugt.

Beim Aufruf des Workflows aus dem Builder heraus gilt es nun zwei Dinge zu beachten. Der WorkflowRunner befindet sich in einem anderen Plug-in als der Builder und die Templates. Im Manifest haben wir eine Dependency von *de.nordakademie.lejos.core* auf *org.openarchitectureware.core.workflow*. Dies ist erforderlich, damit der Builder den WorkflowRunner finden und aufrufen kann. Allerdings befinden sich die Templates nicht im Zugriffspfad des Workflows und können demzufolge nicht gefunden werden. Dieses Problem lässt sich durch den Einsatz von Buddy-Classloading lösen. Im vorliegenden Fall muss dazu also der Classloader der Workflow-Engine so eingerichtet werden, dass er abhängige Plug-ins in seinen Klassenpfad mit einbezieht. Seit Version 4.2 ist für die relevanten openArchitectureWare Plug-ins standardmäßig die Buddy-Classloading Policy dependent eingestellt. Diese sorgt dafür, dass alle Plug-ins, die eine Dependency auf openArchitectureWare definieren, automatisch im Klassenpfad des jeweiligen oAW Plug-ins liegen. Die Klassenladerhierarchie wird quasi umgekehrt.

Weiterhin muss der WorkflowRunner Zugriff auf das Modell erhalten. Auch dies ist nicht ganz unproblematisch, da sich das Modell in unserem Fall in einem vom Anwender angelegten Projekt im Workspace befindet. Für den Zugriff auf Ressourcen im Workspace kann kein Buddy-Classloading eingesetzt werden. Stattdessen greifen wir auf Context Classloading zurück. Diese Technik kommt oft zum Einsatz, wenn reflektiv arbeitende Frameworks wie z.B. Spring oder Hibernate auf Ressourcen (oder Klassen) zugreifen müssen, die sich außerhalb des Klassenpfades befinden. Die dazu benötigte Klasse *ProjectIncludingResourceLoader* wird Bestandteil von oAW 5.0 sein, kann jedoch bereits als „Bleeding Edge“ im Rahmen des zu Eclipse migrierten Modeling Workflow Engine Projekts [12] verwendet werden. Listing 3 zeigt, wie der Kontextklassenlader vor und nach dem Aufruf des WorkflowRunners umgeschaltet wird.

**Listing 3**

```
try {
 ProjectIncludingResourceLoader resourceLoader =
  new ProjectIncludingResourceLoader(
  project);
 ResourceLoaderFactory.
                    setCurrentThreadResourceLoader
                                  (resourceLoader);

 WorkflowRunner runner = new WorkflowRunner();
 boolean success = runner.run("workflow/generator.
                                              oaw",
  null, properties, slotMap);
} finally {
 ResourceLoaderFactory.
              setCurrentThreadResourceLoader(null);
}
```

## Schneller Einstieg in komplexe Vorgänge

## Fazit

Durch das beschriebene Vorgehen lässt sich mit relativ geringem Aufwand eine integrierte Entwicklungsumgebung aufbauen, deren Bedienung so intuitiv ist, dass auch ungeübte Entwickler schnell den Einstieg finden. Trotz der komplexen Vorgänge (Generierung, Kompilierung und Binden) können kurze Turnaround-Zeiten sichergestellt werden, sodass eine effiziente Entwicklung sichergestellt ist. Erweiterungen, wie zum Beispiel das automatisierte Generieren von Testrahmen für Zustände, sind ohne großen Aufwand integrierbar. Das hier beschriebene Visual leJOS Development Toolkit kann kostenlos von der Projektseite [13] bezogen werden.

**Frank Zimmermann** ist Professor an der Fachhochschule Nordakademie. Dort unterrichtet er die Programmiersprache Java, Softwareengineering und die Entwicklung von Informationssystemen. Er ist Autor des leJOS Statemachine Development Toolkits. Kontakt: frank.t.zimmermann@googlemail.com

**Peter Friese** arbeitet als Softwarearchitekt für die itemis AG in Kiel (http://oaw.itemis.de). Er ist Committer für die Open-Source-Projekte openArchitectureWare und Eclipse Modeling. Peter ist Autor verschiedener Artikel zu den Themenkomplexen Eclipse, Spring und Modellgetriebene Softwareentwicklung und hält zu diesen Themen regelmäßig Vorträge auf Softwarekonferenzen. Kontakt: peter.friese@itemis.de.

## >> Links & Literatur

[1] mindstorms.lego.com

[2] home.earthlink.net/~xaos69/NXT/Line_Follower/Line_Follower.html

[3] www.lejos.org

[4] Schwarzbach; Java goes to Space; *Java Magazin*, September 2002: it-republik.de/jaxenter/artikel/Java-goes-to-space-0232.html

[5] Gosling; The Green UI; blogs.sun.com/jag/entry/the_green_ui/

[6] Sommerville; Software Engineering; Pearson Studium 2001

[7] www.eclipse.org/modeling/emf/

[8] www.eclipse.org/modeling/gmf/

[9] www.openarchitectureware.org

[10] Efftinge, Friese, Köhnlein: Best Practices für MDSD; *Java Magazin* 5/2008

[11] Peter Friese; Entwicklungsturbo Incremental Project Builder in Eclipse; IX 8/2004, Seite 121

[12] wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)/

[13] www.assembla.com/spaces/vldt/

[14] Neuhaus, Niehues, Wendehals; Welten wachsen zusammen; *Eclipse Magazin* Vol. 13, S. 49-53

[15] Stahl, Völter, Efftinge, Haase: Modellgetriebene Softwareentwicklung; dpunkt.verlag