DOMÄNENSPEZIFISCHE SPRACHEN: VERSCHIEDENE ANSÄTZE IM VERGLEICH

Es gibt verschiedene Ansätze, nach denen eine domänenspezifische Sprache (DSL) erstellt werden kann. In diesem Artikel wollen wir sie beschreiben und vergleichen. Dabei erörtern wir einerseits interne DSLs, wie Scala und Ruby, und anderseits externe DSLs, wie Xtext und Poseidon for DSL, ein Werkzeug zum Definieren graphischer Sprachen. Wir vergleichen die Ansätze anhand einer gemeinsamen Domäne. Während textuelle DSLs dazu geeignet sind, Verhalten zu modellieren, bieten sich graphische DSLs an, um komplexe strukturelle Abhängigkeiten zu modellieren. Interne DSLs, die eine vorhandene Wirtssprache nutzen, lassen sich oft einfach in bestehende Projekte integrieren. Externe DSLs sind häufig verständlicher für die Domänenexperten.

Domänenspezifische Sprachen (*Domain Specific Languages, DSLs*) erfahren seit einiger Zeit einen erneuten Aufschwung. Früher vor allem bei Unix-Kennern beliebt, werden DSLs heute in verschiedenen Bereichen des Software-Engineerings eingesetzt (vgl. [Gos10]).

Eine DSL ist eine Programmiersprache mit eingeschränkter Aussagekraft und dem Fokus auf einer spezifischen Domäne. Der Nutzen von DSLs wird im Allgemeinen in der höheren Produktivität der Entwicklung und in der verbesserten Kommunikation zwischen Entwicklern und Domänenexperten gesehen. Diese Kommunikation wird durch ein gemeinsames Vokabular vereinfacht (vgl. [Fow10]). Unser Artikel wendet sich an Personen, die eine DSL erstellen wollen. Sie erhalten Einblicke in unsere Erfahrungen mit verschiedenen DSL-Ansätzen. Der Artikel soll eine Entscheidungshilfe bieten, den im konkreten Fall am besten geeigneten Ansatz zu finden. Wir erweitern die von Viktor Sirotin in [Sir09] dargelegte Entscheidungshilfe zwischen UML und DSL im domänenspezifischen Bereich durch einen praxisbezogenen Erfahrungsbericht auf Basis aktueller Technologien. Das im Artikel verwendete Anwendungsbeispiel haben wir in **Kasten 1** zusammengefasst.

Bei der Erstellung einer DSL gibt es verschiedene Möglichkeiten, diese zu implementieren (vgl. [Fow10]):

Um die verschiedenen Ansätze zu vergleichen, wählen wir eine methodenübergreifende Problemstellung. Für alle Ansätze verwenden wir dieselbe Domäne und entwickeln in dieser eine DSL. Als Domäne haben wir uns für die 2D-Physik-Engine *Phys2D* (vgl. [Phy]) entschieden. Phys2D ist Java-basiert und erlaubt es dem Benutzer, eine zweidimensionale Welt mit Rechtecken, Kreisen, statischen Linien und Kräften zu versehen.
Listing 1 zeigt ein Beispiel für eine Einbindung der Physik-Engine in Java-Code. Das Resultat der erstellten Welt sieht man in **Abbildung 1**. Dabei fällt auf, dass die Integration der Phys2D in Java relativ aufwändig ist und viel Code nötig ist, um das Beispiel zu realisieren.

Kasten 1: *Beispieldomäne: 2D Physik Engine.*

```
protected void init(World world) {
  Body body1 = new StaticBody("Ground",
      new Box(200.0f, 20.0f));
  body1.setPosition(350.0f, 280);
  body1.setRotation(-0.7f);
  world.add(body1);
  Body body2 = new Body("Ball 1", new
      Circle(25), 100.0f);
  body2.setPosition(300.0f, 4.0f);
  world.add(body2);
  Body body3 = new Body("Ball 2", new
      Circle(25), 100.0f);
  body3.setPosition(350.0f, 100.0f);
  world.add(body3);
}
```

Listing 1: *Beispiel einer Java-Integration von Phys2D.*

die autoren

Dr. Roman Bertolami
(E-Mail: roman.bertolami@zuehlke.com)
ist Senior Software Engineer bei der Zühlke Engineering AG. Zu seinen Schwerpunkten gehören agile Softwareentwicklung, DSLs und Softwaredesign.

Frank Buchli
(E-Mail: frank.buchli@zuehlke.com)
hat als Softwarearchitekt bei der Zühlke Engineering AG in verschiedenen Rollen unterschiedliche Java-Projekte begleitet. Seine Stärken sind das methodische Vorgehen und das konkrete Wissen und die Erfahrung mit MDSD-Ansätzen in der Praxis.

Marc Hofer
(E-Mail: marc.hofer@zuehlke.com)
ist Software Engineer bei der Zühlke Engineering AG. Zu den von ihm vertieften Themen gehören agile Softwareentwicklung, Geschäftsprozess-Modellierung sowie DSLs.

Matthias Rüedlinger
(E-Mail: matthias.rueedlinger@zuehlke.com)
ist Software Engineer bei der Zühlke Engineering AG. Zu seinen Interessen zählen UML, Xtext und DSLs.

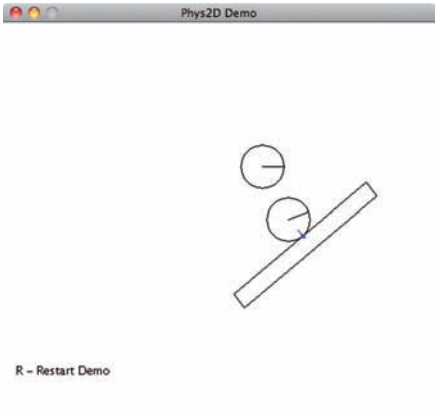

Abb. 1: Screenshot des in Listing 1 dargestellten Beispiels.

- Eine *interne DSL* nutzt eine vorhandene Wirtssprache, um ihre Syntax und Semantik zu definieren. Die interne DSL stellt also eine Untermenge der generellen Wirtssprache dar.
- Eine *externe DSL* ist eine von Grund auf neue Sprache, bei der Syntax und Semantik selbst definiert werden. Infrage kommen eine textuelle oder eine graphische DSL. Eine textuelle DSL basiert normalerweise auf einer Grammatik mit den entsprechenden Werkzeugen. Externe DSLs können auch als graphische DSL realisiert werden. Dabei verwenden Entwickler oft UML Profile. Es ist aber auch möglich, eine eigene graphische Notation zu entwerfen.

In diesem Artikel betrachten wir vier Vertreter aus diesen Bereichen:

- *Ruby* und *Scala* sind zwei Vertreter einer internen DSL.
- *Xtext* ist eine textuelle, externe DSL und *Poseidon for DSL* steht für eine externe, graphische DSL.

Als Entscheidungshilfe zur Auswahl der DSL-Methodik untersuchen wir die von uns gewählten Lösungsansätze nach folgenden Kriterien:

- *Lesbarkeit und Verständlichkeit:* Wie leicht ist es für einen Domänen-Experten, den von einem Software-Engineer erstellten DSL-Code zu verstehen?
- *Integrations- und Wartungsaufwand:* Wie gut fügt sich die DSL in das bestehende Framework ein?
- *Einfachheit:* Ist es für einen Software-Engineer einfach, sich in die Metaprogrammierung der DSL einzuarbeiten?
- *Skalierbarkeit:* Lassen sich mit der DSL größere Mengen von Daten und Datentypen effizient verarbeiten?

Interne DSL mit Scala

Scala ist eine Programmiersprache, die objektorientierte Elemente mit Elementen funktionaler Programmierung integriert (vgl. [Ode08]). Scala ist statisch typisiert und wird vor der Ausführung in Java-Bytecode kompiliert.

Verschiedene Spracheigenschaften machen Scala zu einem exzellenten Werkzeug für das Erstellen interner DSLs für bestehende Java-APIs. Die Lesbarkeit einer DSL kann erhöht werden, weil rein syntaktische Sprachelemente wie Klammern und Punkte in Scala oft optional sind. Mit der Spracheigenschaft der impliziten Konversion lassen sich bestehende Klassen einfach mit einer automatischen Typumwandlung erweitern. Partielle Funktionen sind in Scala gut integriert, sodass es möglich ist, die DSL an spezifischen Punkten zu erweitern.

Vermeidung von unnötigem Code

Mit Scala lässt sich unnötiger Code, also der Code, der nichts mit der eigentlichen Aufgabe des Codes zu tun hat (zum Beispiel Getter- und Setter-Methoden in Java), auf ein Minimum reduzieren. Falls eine Methode nur einen Parameter annimmt, kommt diese Methode ohne Klammersetzung aus. Damit kann die DSL lesbarer und einer natürlichen Sprache ähnlicher gestaltet werden.

Companion Object

Um eine Instanz einer Klasse zu erstellen, muss häufig unnötiger Code (beispielsweise das Schlüsselwort new in Java) verwendet werden. Mit Hilfe von *Companion Objects* kann man diesen Code in Scala auf einfache Weise eliminieren. So ergibt sich eine elegantere DSL. Ein Beispiel ist in Listing 2 dargestellt.

```
// Class Box with case companion object
case class Box(val x: Float, val y:
  Float, w: Float, h: Float) {
}
-----------------------
// DSL code
World add Box (175,200,100,20)
```

Listing 2: Das Erstellen einer Box kann ohne das Schlüsselwort „new" und ohne Punktnotation beschrieben werden.

```
// implicit conversion
implicit def intToMyInt(x: Int) = new MyInt(x)
implicit def myIntToInt(x: MyInt) = x.value
// MyInt: an extension of Int
class MyInt(val value: Int) {
    def seconds = this
    def minutes = new MyInt(60*value)
}
-----------------------
// DSL code
after ( 1 seconds,
      () => World add Box(10,0,10))
```

Listing 3: Erweiterung der Integer Klasse mittels „implicit".

Bestehenden Typ mit „implicit" erweitern

In Scala können Klassen nicht dynamisch verändert werden. Das mächtige Scala-Konstrukt implicit ermöglicht es aber trotzdem, einem bestehenden Typen zusätzliche Funktionalität hinzuzufügen, indem eine implizite Typkonversion deklariert wird. Listing 3 zeigt, wie wir implicit einsetzen, um die Klasse Integer als Zeiteinheit in der DSL zu verwenden.

```
// command class that executes a
  function on an event
case class CollisionCommand(shape1:
  Shape, shape2: Shape,
  collisionInstruction: (Shape,
  Shape) => Unit) {
  def actOnEvent(event:
    CollisionEvent) {
  collisionInstruction(shape1,
    shape2)
  }
}
-----------------------
// DSL code, including custom
  extension
World add
  CollisionCommand(box2, box3,
    () => println("Autsch"))
```

Listing 4: Gezielte Erweiterung mittels Funktion höherer Ordnung.

```
# opening the symbol class to add a
  method that improves our DSL.
class Symbol
  def is val
    # do something with the value...
  end
end
-----------------------
# DSL code, improved readability with
  open classes
:position .is [0, 120]
```

Listing 5: Verwendung von Open classes zur Erweiterung des Symbols „:position".

DSL-Erweiterungen gezielt einbauen

Weil in Scala Funktionen höherer Ordnung definierbar sind, können wir gezielt Elemente in die DSL einbauen, die es dem Anwender erlauben, selbst erstellte Funktionalität hinzuzufügen. Listing 4 zeigt, wie eine Funktion hinzugefügt werden kann, die bei einer Kollision von zwei graphischen Elementen ausgeführt wird.

Fazit

Durch die Integration von funktionalen und objektorientierten Elementen steht mit Scala eine mächtige Sprache zum Bauen interner DSLs zur Verfügung. Scala-DSLs eignen sich gut für die Nutzung in Java-Projekten, da Scala auch auf der JVM läuft. Scala ist jedoch keine einfache Sprache – um eine lesbare DSL zu erstellen, ist einiger Aufwand erforderlich. Funktionale Elemente lassen sich gut parallelisieren, sodass Scala sich zum effizienten Verarbeiten größerer Projekte gut eignet.

Die Autoren von Scala waren darauf bedacht, Skalierbarkeit in verschiedenen Aspekten zu ermöglichen (vgl. [Ode08]), wovon man beim Erstellen einer DSL profitiert. Einerseits lassen sich problemlos größere Modelle verwalten, da man auf etablierte Möglichkeiten der Entwicklungsumgebungen zurückgreifen kann. Anderseits wächst eine interne DSL mit Scala auch gut mit der Komplexität der Domäne. So lassen sich einfache Probleme in einer einfachen DSL darstellen, es ist aber auch möglich, für komplexe Domänen eine passende DSL zu erstellen.

```
class Line
end
class World
  def add object
    puts "added ", object
  end
end
def line positions
  Line.new
end
# definition of play which yields to
  block
def to_play
  yield World.new
end
-----------------------
# DSL code, play is called with the
  world object to be used in the
  block
to_play do |world|
  world.add line :position1 => [0, 0],
    :position2 => [200, 100]
  world.add line :position1 => [500,
    0], :position2 => [300, 100]
end
```

Listing 6: Verwendung von Closures oder Blocks.

Interne DSL mit Ruby

Ruby ist eine dynamische, objektorientierte Programmiersprache, deren Syntax stark an Perl oder Smalltalk erinnert (vgl. [Tho09]). Durch das bekannte Web-Framework „Ruby on Rails" hat sie in den letzten Jahren große Beliebtheit erlangt. Ruby hat verschiedene Eigenschaften, die es erlauben, interne DSLs effizient zu entwickeln. Insbesondere *Open classes*, *Blocks/Closures*, *Method Missing* sowie ganz einfache Funktionen oder globale Variablen erleichtern dem Software-Engineer das Entwickeln einer DSL.

Bei der Erstellung einer internen DSL mit Ruby können dynamische oder zeitabhängige Elemente auf einfache Weise in einem Modell textuell beschrieben werden.

Open classes

Open classes erweitern eine bestehende Klasse um weitere Methoden. Diese lassen sich nutzen, um eine sprachspezifische Syntax durch Einfügen sprechender Methoden zu verschönern. Listing 5 zeigt als Beispiel, wie mit *Open classes* ein Symbol mit einem sprechenden Methodenaufruf erweitert wird.

Blocks

Blocks oder *Closures* sind bekannte Mechanismen in der funktionalen Programmierung, um Funktionen als Parameter an eine Methode zu übergeben. Mit Ruby entsteht für unsere DSL ein Block, der auf mehreren Codezeilen den Inhalt unserer Physikwelt beschreibt. In Blöcke können beliebige Parameter mitgegeben werden, wie in Listing 6 das Objekt world.

```
class World
  def method_missing(method_name, *a)
    # find the effective method name
      to be called
    method_to_call = find_method(
      method_name)
    self.send(method_to_call,*a)
  end
end
-----------------------
# DSL code, the called method does not
  exist but is being used to extract
  the required information
world.read_all_circles_radius_bigger_20
```

Listing 7: Verwendung von Method Missing.

Method Missing

Method Missing wird standardmäßig aufgerufen, wenn die ursprünglich verlangte Methode bzw. der ursprünglich vorgesehene Empfänger einer Nachricht nicht vorhanden ist. Anstelle von generischen Methoden mit vielen Übergabe-Parametern können auf diese Weise wirklich sprechende Methoden definiert werden. Das ist ein Sprach-Feature, das in Ruby on Rails oft verwendet wird.

Reflection

Reflection kann verwendet werden, um Objektinstanzen zu erstellen, ohne dabei typenspezifischen Code schreiben zu müs-

```
class ElementFactory
  def Circle
    puts "create a Circle here..."
  end
end
def create element
  ElementFactory.new.send element
end
-----------------------
# DSL code, use a symbol to create an
element
create(:Circle)
```

Listing 8: Verwendung von Reflection.

```
while true
  # read line from STDIN and interpret
    it as ruby script
  x = STDIN.readline
  Kernel.eval x
end
_____

# executing DSL script and adding a
  circle
> ruby main.rb
world.add(:Circle)
```

Listing 9: Reflection zur dynamischen Eingabe der DSL.

Abb. 2: Screenshot des Poseidon-for-DSL-Modells.

```
node Circle {
  metamodel_element: Circle
  default_size: 2 * 2
  minimum_size: 1 * 1
  shape: ELLIPSE
  name_position: NO_NAME
  keep_proportions: true
}
edge Joint {
  metamodel_element: Joint
  source: Box Circle
  target: Box Circle
  property FillColor : RED
}
```

Listing 10: Ausschnitt der Konfiguration der Poseidon-for-DSL-Workbench.

sen. Listing 8 zeigt, wie das Symbol :Circle verwendet werden kann, um über *Reflection* das entsprechende Objekt über eine Factory-Methode zu instanziieren.

Ebenfalls über einen weiteren *Reflection*-Mechanismus können wir zusätzliche Code-Stücke ausführen. Das Beispiel in Listing 9 zeigt, wie über die Konsole dynamisch ein weiteres Element in unsere Welt eingefügt wird.

Fazit

Nicht zufällig hat die Ruby-Community einen großen Anteil an der steigenden Popularität von DSLs. Durch die dynamische Meta-Programmierung lassen sich in Ruby einfach interne DSLs erstellen. Häufig schimmert die Wirtssprache verhältnismäßig stark durch, was zu einer geringeren Lesbarkeit führen kann. Mit dem „jRuby Interpreter" kann man die erstellte DSL in einfacher Weise in bestehende Java-Projekte integrieren.

Wie auch bei anderen textuellen DSLs (intern oder extern) sehen wir in Ruby eine gute Möglichkeit, auch große Modelle zu erstellen und zu verwalten. Die Skalierbarkeit bzw. Performance zum Ausführungszeitpunkt ist dagegen stark abhängig von der zu Grunde liegenden Implementierung, tendenziell aber eher schneller als bei interpretierten Modellen.

Graphische DSLs mit Poseidon

„Poseidon for DSL" ist ein kommerzielles Framework der Firma Gentleware (vgl. [Gen]) zum Entwickeln graphischer DSLs. Poseidon for DSL war früher Teil von „Poseidon for UML" und wird nun als eigenständiges Produkt vermarktet.

Metamodell von Poseidon for DSL

Poseidon for DSL definiert ein eigenes Metamodell auf Basis von Ecore (vgl. [EMF]). Dieses Metamodell dient als Grundlage für die Definition unserer graphischen DSL. Dafür stehen die Grundelemente *Nodes* und *Edges* zur Verfügung. Auf der Basis des vordefinierten Metamodells bilden wir die konkrete Syntax. Für die Abbildung der Welt erstellen wir ein Element mit dem Name World, dazu kommen noch Body, Box und Circle. Abbildung 2 zeigt einen Screenshot des Metamodells für Phys2D.

Konfiguration der Workbench

Der konkreten Syntax unseres Modells weisen wir die graphische Repräsentation zu. Dazu gehört, dass eine Box als Rechteck und ein Circle als Kreis in der Workbench abgebildet werden. In dieser definiert man die Grammatik unseres Modells, z. B. welche Elemente sich miteinander verbinden lassen, und zusätzliche Eigenschaften, wie Standardgröße und Farben.

Graphische DSL in der Workbench

Abbildung 3 zeigt die fertig konfigurierte „Poseidon for DSL Workbench" für die erstellte DSL. In der Toolbar auf der linken Seite sind die konkreten Elemente der Sprache verfügbar. Die Modellierung der Welt kann intuitiv mittels *Drag&Drop* vorgenommen werden.

Codegenerierung

Durch die Konfiguration von Poseidon for DSL ist die Physik-Engine noch nicht eingebunden. Dazu müssen Code-Generatoren

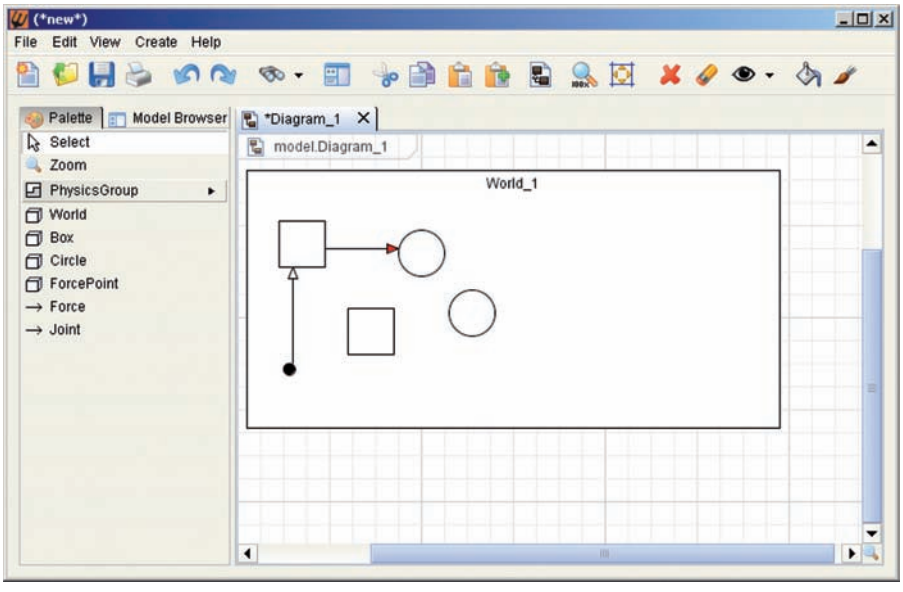

Abb. 3: Screenshot der DSL.

geschrieben werden. Wir verwenden zum Generieren *openArchitectureWare (oAW)* mit der funktionalen Template-Sprache XPand (vgl. [oAW]). Mit oAW kann ein Template programmiert werden, das die definierten Elemente World, Box und Circle verarbeitet und das den entsprechenden Code für Phys2D erstellt.

Fazit

Mit der graphischen Notation und dem zugehörigen Editor kann der Domänen-Experte verständliche und gut lesbare DSLs erstellen. In der verwendeten Domäne wird das besonders deutlich, weil diese eine intrinsische graphische Notation hat. Schwierig ist es, sehr viele graphische Elemente zu modellieren. Eine gute Skalierbarkeit ist mit einem Mehraufwand zu erreichen, indem man komplexe Domänen in einzelne Diagramme auslagert. Der größte Aufwand für den Software-Engineer ist das Erstellen des Generators. Die Integration der DSL ist einfach zu realisieren, da mit dem Generator Code in der gewünschten Programmiersprache des Projekts erstellt werden kann. Dadurch hat man den zusätzlichen Vorteil, dass die DSL unabhängig von der zu erstellenden Software ist.

Textuelle DSL mit Xtext

Xtext ist ein Eclipse-Plug-In, mit dem textuelle DSLs – basierend auf der erweiterten Backus-Naur-Form – einfach und schnell erstellt werden können (vgl. [Xte]). Xtext erzeugt einen Eclipse-Editor, der Syntax-Hervorhebung und Auto-Vervollständigung für die DSL unterstützt und damit die Bearbeitung des DSL-Codes vereinfacht.

```
World:
  'World' name=ID ':' ('relative mass'
    '=' relativeMass = INT )?
  scenery = Scenery;
Scenery: 'Scenery:'
  (elements += ElementDef)+;
ElementDef: element=Element ('at'
  position=Point)?;
Element: Box | Circle | Line;
Box: 'Box' width=INT 'x' heigth=INT;
Circle: 'Circle' radius=INT;
Line: 'Line' 'from' p1=Point 'to' p2=
  Point;
Point: x=INT ':' y=INT;
```

Listing 11: *Xtext Grammatik zur Phys2D.*

Grammatik und textuelles Modell

Analog zum Metamodell von Poseidon for DSL erstellen wir eine Grammatik für die textuelle DSL. Anhand von Ableitungsregeln sowie terminalen und nicht-terminalen Symbolen bestimmt die Grammatik die Syntax der DSL. **Listing 11** zeigt die Grammatik der textuellen DSL für die Physik Engine. Xtext erstellt zu dieser Grammatik einen Eclipse-Editor, mit dem textuelle Modelle erstellt werden. **Listing 12** zeigt ein textuelles Modell auf der Basis der Grammatik.

Generatoren

Da wir nicht auf eine Wirtssprache zurückgreifen können, müssen wir den erforderlichen Programmcode mittels Generatoren erzeugen. Dazu verwenden wir denselben oAW-basierten Ansatz wie bei Poseidon for DSL.

Fazit

Mit Xtext können wir die DSL unabhängig von einer Wirtssprache erstellen, was die Lesbarkeit für den Domänen-Experten erhöht. Um jedoch die Phys2D-Engine effektiv anzusteuern, muss ein Code-Generator geschrieben werden. Der Einstieg gestaltet sich für den Software-Engineer relativ einfach. Es gibt bereits eine große Community, die bei Problemen aktiv hilft. Zahlreiche Projekte bestätigen, dass sich Xtext auch für größere Projekte eignet, da die Skalierbarkeit gegeben ist.

Diskussion

In einer kurzen Diskussion vergleichen wir die verschiedenen Ansätze unter bestimmten Gesichtspunkten, um die Stärken und Schwächen hervorzuheben.

Gegenüberstellung von externen und internen DSLs

Externe DSLs erlauben eine freie Gestaltung im Rahmen einer formalen Sprache. Bei internen DSLs sind wir hingegen an die Wirtssprache gebunden. Diese enge Verbindung mit der Wirtssprache hat jedoch den Vorteil, dass die Entwickler die bestehende Funktionalität der Wirtssprache für die Definition der Semantik der DSL verwenden können. Das erlaubt ein effizientes Entwickeln der DSL. Durch die Nähe zur Wirtssprache entsteht bei internen DSLs deutlich weniger Aufwand beim Erstellen und Warten der benötigten Werkzeuge. Zwar ist sowohl bei Ruby als auch bei Scala eine Sprache hinzugekommen; diese läuft aber zusammen mit der bestehenden Code-Basis auf der gleichen JVM. So ist die Hürde DSL für involvierte Projektmitarbeiter einfacher zu nehmen. Allerdings besteht auch die Gefahr, dass DSL-Code und sonstiger Code vermischt werden.

```
World MyWorld :
Scenery:
  Box 20 x 20 at 100 : 200
  Box 40 x 40 at 300 : 200
  Box 400 x 10 at 50 : 400
  Circle 20 at 150 : 380
```

Listing 12: *Phys2D-Modell mit Xtext.*

Scala oder Ruby als Wirtssprache interner DSLs?

Sowohl Scala als auch Ruby eignen sich als Wirtssprache für DSLs. Mit beiden Sprachen kann man den unnötigen Code auf ein Minimum reduzieren und so die DSL lesbar gestalten. Ruby hat den Vorteil, dass dynamische Meta-Programmierung verwendet werden kann – beispielsweise können Klassen zur Laufzeit verändert werden. Da Scala statisch typisiert ist, bietet es diese Möglichkeit nicht – dafür kann bei Scala mit einer besseren Laufzeit-Performance gerechnet werden. Sowohl Ruby als auch Scala bieten eine interaktive Shell, mit der das aktuelle Modell zur Laufzeit verändert werden kann. Mit Ruby ist es außerdem möglich, mittels der „JSR223: Scripting for the Java Platform" (vgl. [JSR]) die Ruby-DSL zur Laufzeit in eine Java-Applikation zu integrieren.

Vergleich von graphischen und textuellen DSLs

Der Vorteil einer graphischen DSL liegt in der intuitiven Anwendung, insbesondere dann, wenn die Domäne schon eine entsprechende graphische Notation kennt, wie z. B. bei Schaltsymbolen, oder wenn eine solche Notation intuitiv klar ist, z. B. für die Platzierung von Rechtecken in der Physik-Engine.

Textuelle DSLs wiederum gestatten es, etablierte Prozesse und Werkzeuge der Softwareentwicklung zu verwenden, die bei

graphischen DSLs unter Umständen schwierig zu realisieren sind. Beispiele hierfür sind Quellcode-Management, Freitext-Suche oder paralleles Arbeiten.

Eine graphische DSL eignet sich gut, um komplexe Beziehungen und Abhängigkeiten einer beschränkten Anzahl von Elementen darzustellen, während textuelle DSLs Vorteile in der Modellierung von Verhalten aufweisen.

Einfachheit der Integration in eine bestehende Applikation

Die erstellten internen DSLs erlauben eine unmittelbare Ausführbarkeit der mit der DSL erstellten Modelle. Während bei Scala noch ein Kompilieren in JVM-Bytecode nötig ist, kann man mit Ruby und dem jRuby Interpreter die Modelle direkt interpretieren. Durch die unmittelbare Ausführbarkeit können die DSLs selbst – wie auch die mit der DSL erstellten Modelle – im selben Lebenszyklus entwickelt und getestet werden.

Die Implementierung der internen DSLs mit Ruby und Scala wird durch die direkte Einbindung bestehender Java-Ressourcen erleichtert. Dadurch können sowohl der bestehende Java-Code, als auch alle gängigen Java-Bibliotheken direkt angesprochen werden. Im Gegensatz zu internen DSLs wird bei externen DSLs von einem Generator Code in der Zielsprache erzeugt. Dieser wird dann mit dem bestehenden Code integriert und betrieben. Der generierte Code ist unabhängig von der DSL. Das bietet den Vorteil, dass er – sollte die Wartung der DSL eingestellt werden – separat weiterentwickelt werden kann.

Schlussfolgerungen

Der in diesem Artikel vorgenommene Vergleich verschiedener Ansätze zum Erstellen einer DSL zeigt, dass es nicht den einen richtigen Weg gibt, um eine DSL zu erstellen. Die Wahl hängt stark von der zu modellierenden Domäne ab. Aber auch weitere Faktoren, wie bestehende Erfahrungen im Team, Integrationspunkte mit bestehender Software oder Kenntnisse der DSL-Anwender, spielen bei der Auswahl eine wichtige Rolle. ■

Literatur & Links

[EMF] Eclipse Modeling Framework Project (EMF), siehe: http://www.eclipse.org/modeling/emf/

[Fow10] M. Fowler, Domain Specific Languages, Addison-Wesley Professional 2010

[Gen] Gentleware, siehe: http://www.gentleware.com

[Gos10] D. Gosh, DSLs in Action, Manning Early Access Program, Manning 2010

[JSR] JSR 223: Scripting for the Java Plattform, siehe: http://jcp.org/en/jsr/detail?id=223

[oAW] openArchitectureWare – The leading platform for professional model-driven software development, siehe: http://www.openarchitectureware.org/

[Ode08] M. Odersky, L. Spoon, B. Venners, Programming in Scala: A Comprehensive Step-by-step Guide, Artima Incorporation 2008

[Phy] Phys2D physics engine, siehe: http://www.cokeandcode.com/phys2d/

[Sir09] V. Sirotin, UML vs. DSL: Die Qual der Wahl, in: OBJEKTspektrum 2/2009

[Tho09] D. Thomas, C. Fowler, A. Hunt, Programming Ruby, Pragmatic Bookshelf 2009

[Xte] Xtext – Language Development Framework, siehe: http://www.xtext.org/