



Best Practices für modellgetriebene Softwareentwicklung

Modellgetriebene Softwareentwicklung ist keine Randerscheinung mehr, sondern wird in mehr und mehr Softwareprojekten mit großem Erfolg eingesetzt. In diesem Artikel geben wir unsere in den letzten Jahren gesammelten Erfahrungen in Form von Best Practices weiter.

von Sven Efftinge, Peter Friese und Jan Köhnlein

W^{eil} die Verwendung von domänenpezifischen Sprachen (DSLs) und Codegeneratoren nicht erst seit gestern verbreitet ist, ist dies natürlich auch nicht der erste Artikel, der sich mit der Beschreibung von Best Practices beschäftigt [1], [2]. Einige der in der Vergangenheit beschriebenen Praktiken haben sich inzwischen fest etabliert, andere sind im Laufe der Zeit weniger wichtig oder sogar obsolet geworden. Den etablierten Ansätzen widmen wir den

ersten Teil des Artikels. Darüber hinaus haben sich neue Best Practices herauskristallisiert, die bisher noch nicht festgehalten wurden. Diese werden im zweiten Teil dargestellt. Los geht es mit den sehr wichtigen Basisempfehlungen.

Trenne generierten und manuellen Code

Eine ganz wichtige Sache und doch immer wieder missachtet: Für jedes Artefakt muss eindeutig klar sein, ob es

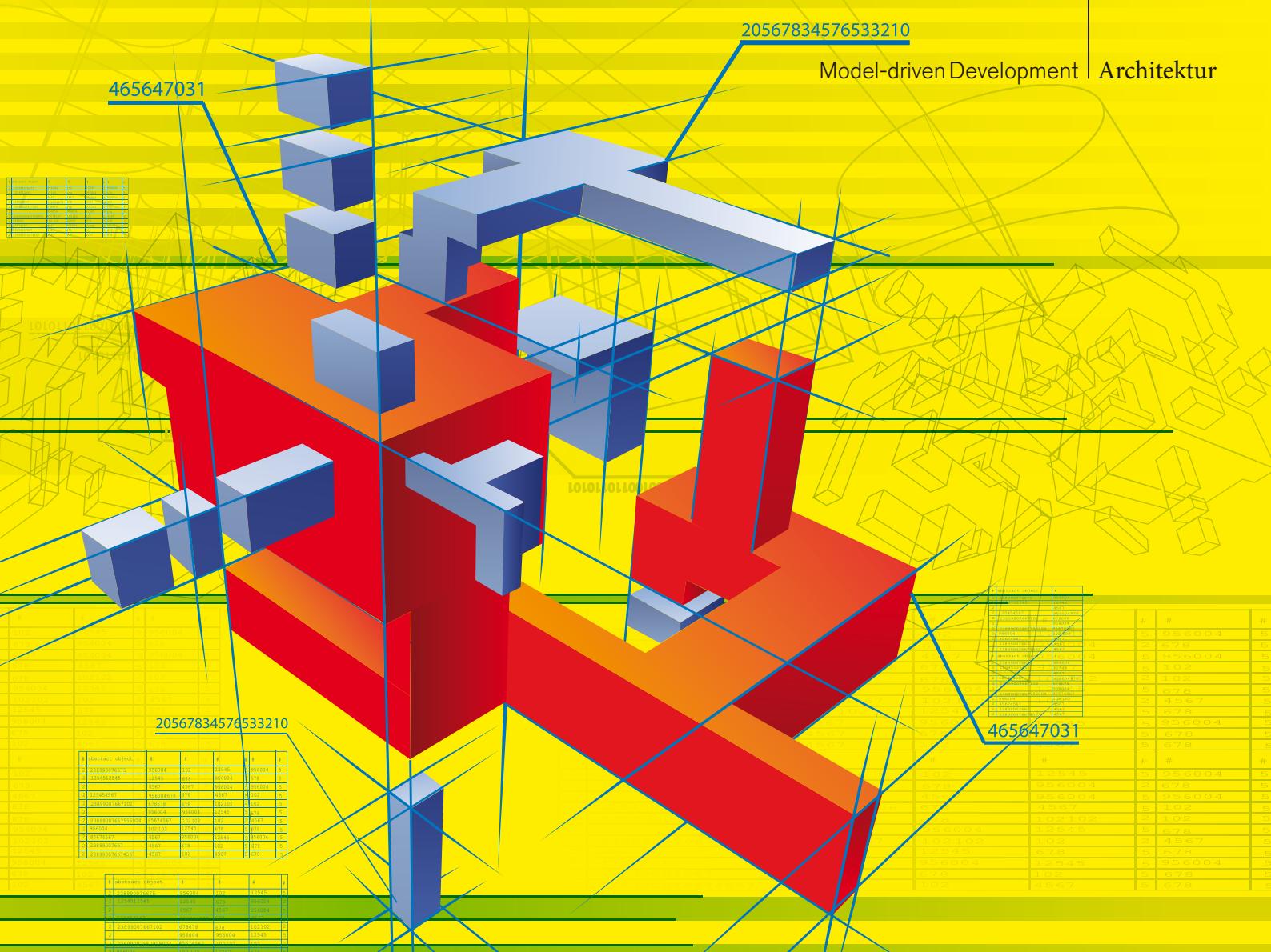
generiert oder vom Entwickler zu manipulieren ist. Das bedeutet nicht, dass man einen Codegenerator nicht auch dazu verwenden sollte, Code zu generieren, der hinterher vom User erweitert bzw. verändert werden kann. Nur sollte dies explizit geschehen und vor allem sollte dieser Code nur einmal generiert werden (Kasten: „Generator“). Code, der ausschließlich generiert wird, sollte weder verändert noch im Repository eingeccheckt werden. Er ist ein Wegwerfprodukt – die wichtigen Artefakte sind die Modelle.

Passive Codegenerierung, also einmalige Generierung im Stil eines Wizards, ist in letzter Zeit recht populär geworden. Unter dem Begriff *Scaffolding* wird in fast jedem gehypeten (Web-) Framework der Start in ein neues Projekt möglichst einfach gemacht. Das kann

Generator

Passiver Generator: wird einmalig aufgerufen. Der erzeugte Code wird anschließend manuell weiterentwickelt. Nahezu alle Wizards in Entwicklungsumgebungen fallen in diese Kategorie.

Aktiver Generator: kann immer wieder aufgerufen werden. Wird mithilfe von Konfigurationsdateien oder Modellen konfiguriert. MDSD-Generatoren fallen typischerweise in diese Kategorie [1].



anfangs sehr hilfreich sein und die Akzeptanz eines Frameworks enorm fördern. Da der Start aber nur einen ganz kleinen Teil eines Projektlebenszyklus ausmacht, ist dem Benutzer damit allerdings nur kurzfristig geholfen.

Eine relativ verbreitete Verletzung dieser *Best Practice* ist der Einsatz von *Protected Regions*. Dabei handelt es sich um Textregionen im generierten Code, die der Generator nicht überschreibt. Innerhalb einer *Protected Region* kann der Entwickler manuelle Änderungen vornehmen und der Generator generiert „drumherum“. Dadurch erhebt man allerdings generierten Code zum First-Class-Code, der dann beispielsweise auch eingeccheckt werden muss. Weiterhin müssen die vom Entwickler ausgefüllten *Protected Regions* aktiviert werden, da sie ansonsten nicht geschützt

sind. In der Hitze des Gefechts wird dies leicht vergessen, was dann zu verlorenem Code, langer Fehlersuche und viel Ärger führt.

Wie kann nun die *Best Practice* in der Realität aussehen? Es gibt zwei Varianten, die auch gut gleichzeitig verwendet werden können: Zunächst sollte man für generierten Code ein separates Quellcodeverzeichnis vorsehen (üblicherweise *src-gen*). So erreicht man eine physikalische Trennung von generiertem und nichtgeneriertem Code. Weiterhin sollte man alle generierten Artefakte mit einem Kommentar mit folgendem Inhalt beginnen lassen: *// WARNING! GENERATED CODE, DO NOT MODIFY.* Wer's edel mag, ergänzt dieses Kommentar noch um die Angabe, durch welches Template der Code erzeugt wurde. Beide Varianten lassen sich nun auch

ausnutzen, um in der IDE generierte Dateien anders darzustellen als nichtgenerierte [3].

Checke generierten Code nicht ein

Generierter Code sollte aus denselben Gründen nicht eingeccheckt werden, aus denen auch Java-Bytecode oder sonst ein abgeleitetes Artefakt nicht eingeccheckt werden sollte. Abgeleitete Artefakte sind 100 % redundant, das heißt, sie enthalten keinerlei nicht reproduzierbare Informationen. Solche Artefakte einzutesten, erhöht nur das zu verwaltende Datenvolumen und bei jedem Synchronisieren entstehen Konflikte, die gerne mit einem *Override and Commit* ignoriert werden. Während man alles eingeccheckt, kann immerhin ein neuer Kaffee an den Arbeitsplatz geholt werden...

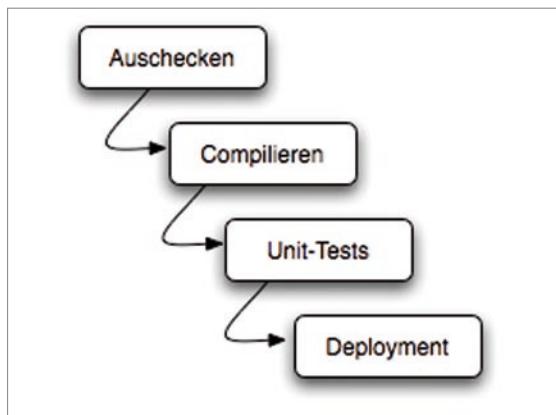


Abb. 1: Build-Prozess ohne Generator

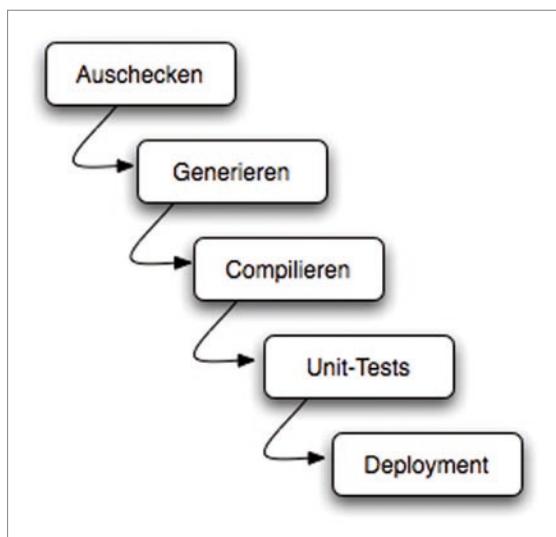


Abb. 2: Build-Prozess mit Generator

Bei *Protected Regions* wird das Problem noch viel größer, weil man hier nicht ohne Weiteres alles überschreiben darf. Es könnte ja sein, dass ein anderer Entwickler in der Zwischenzeit einen geschützten Bereich verändert hat.

Der einzige Fall, in dem es sinnvoll sein kann, generierten Code einzufügen, ist, wenn man aus irgendeinem Grund nicht in der Lage ist, den Generatorlauf in den Build-Prozess zu integrieren.

Integriere den Generator in den Build-Prozess

Die Automatisierung des Build-Prozesses ist eine Maßnahme, welche die Produktivität eines Entwicklungsteams maßgeblich steigern kann. Wichtig hieran ist, dass der Prozess tatsächlich vollständig automatisiert und kein manuelles Eingreifen erforderlich

ist. Normalerweise besteht der Build-Prozess aus den Phasen *Auschecken*, *Compilieren*, gegebenenfalls *Linken*, *Deployment auf die Testplattform* und *Ausführung der Unit-Tests* (Abb. 1). In MDSD-Projekten kommt nun noch die Phase der Generierung hinzu (Abb. 2).

Es versteht sich von selbst, dass der Build-Prozess nicht nur lokal, sondern auch auf einem dafür vorgesehenen Server ausgeführt wird. So ist sichergestellt, dass der Code aller am Projekt beteiligten Entwickler kontinuierlich integriert wird (*Continuous Integration*).

Aus diesem Vorgehen leiten sich zwei neue *Best Practices* ab: *Checke Modell und abhängigen Code immer gemeinsam ein* und *Partitioniere deine Modelle richtig*.

Nutze Mittel der Zielplattform

Die Konzepte der Zielplattform können bei unterschiedlichen Aufgaben helfen. In vielen Fällen wird nicht 100% einer Anwendung aus Modellen abgeleitet, sondern oft nur die Teile, die sich in der Zielsprache (z.B. Java) schlecht ausdrücken lassen.

Hierbei handelt es sich in der Regel um schematische Informationen, wie Domänenmodelle oder Serviceschichten. Die Logik, die diese Schichten ebenfalls enthalten, kann sehr gut in einer beliebigen Programmiersprache definiert werden.

Statt nun *Protected Regions* zu verwenden und die Logik in den generierten Code zu integrieren, bietet es sich an, mit den Mitteln der Zielsprache die unterschiedlichen Aspekte in zwei verschiedenen Dateien zu hinterlegen. Eine wird generiert und eine wird vom Entwickler durch die fehlenden Informationen angereichert.

In Java kann man beispielsweise Vererbung einsetzen. Die dreistufige Vererbung [MDSD] hat sich in vielen Szenarien bewährt. Dabei wird vom generischen Framework eine abstrakte Basisklasse bereitgestellt, die von einer generierten und ebenfalls abstrakten Klasse überbaut wird. Die konkrete Klasse erbt schließlich von der generierten Klasse.

Natürlich gibt es noch etliche weitere Varianten, wie die Kombination von Vererbung und Delegation. Bei anderen

Zielsprachen müssen gegebenenfalls weitere Konzepte verwendet werden, zum Beispiel *Includes*.

Generiere sauberen Code

Nur weil Code automatisch erstellt wurde, heißt das nicht, dass er nie wieder von Menschen gelesen werden wird. Der generierte Code sollte deshalb denselben Qualitätsansprüchen genügen, die für manuell erstellten Code gelten. Eine homogene Codebasis erleichtert später das Verständnis und die Fehlersuche.

Durch den Einsatz eines *Code Formatters* im direkten Anschluss an die Generierung kann die Einrückung und Formatierung automatisiert werden, ohne die Komplexität der Templates zu erhöhen. Der generierte Code sollte geltende *Coding Styles* einhalten und gängige Entwurfsmuster benutzen.

Natürlich besteht eine große Gefahr darin, dass an vielen Stellen identischer Code generiert wird. Schließlich möchte man sich ja gerade die Arbeit für die stupid, immer wiederkehrenden Aufgaben erleichtern. Die DRY-Regel (*Don't Repeat Yourself!*) scheint für generierten Code nicht zu gelten, da man globale Änderungen ja tatsächlich nur an einer zentralen Stelle – im Generator – durchführen kann. Es ist dennoch besser, gemeinsamen Code in die Plattform zu extrahieren, auf der der generierte Code aufsetzt. Im Generator kann man sich dann auf die Unterschiede konzentrieren und der Code bleibt für alle verständlich.

Nutze den Compiler

Eine weitere Möglichkeit, die Zielplattform zu nutzen, ist über den Compiler der Zielsprache mit dem Entwickler zu kommunizieren. Wird beispielsweise die oben beschriebene dreistufige Vererbung verwendet, so könnte für zu implementierende Methoden eine abstrakte Methode generiert werden. Dann fordert der Compiler den Entwickler auf, diese in der konkreten Klasse zu implementieren.

Für komplexe Abhängigkeiten, die ohne Weiteres nicht vom Compiler beanstandet werden würden, kann auch ein Stück exemplarischer *Dummycode* generiert werden, der Code verwendet, der vom Entwickler noch zu erstellen ist.

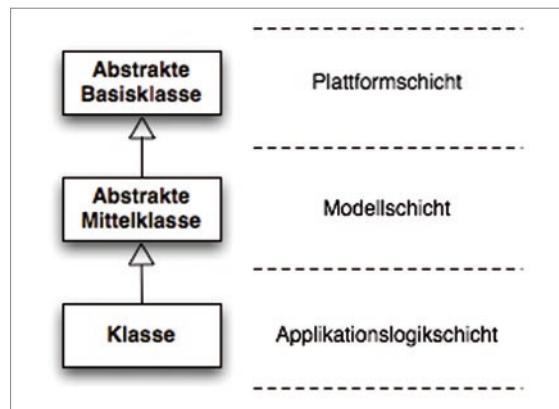


Abb. 3: Build-Prozess mit Generator

Für Fälle, in denen die Zielsprache keinen Compiler bietet und in denen man als Entwickler des Generators trotzdem sicherstellen möchte, dass der Entwickler der Zielplattform bestimmte Dinge manuell hinzufügt (natürlich nicht in den generierten Code!), kann der Einsatz des *Recipe Frameworks* des openArchitectureWare-Projekts [4] helfen. *Recipes* weisen den Entwickler nach jedem Generatorlauf an, zu überprüfen, ob bestimmte Bedingungen erfüllt sind, beispielsweise dass eine bestimmte Klasse manuell erstellt wurde, die wiederum von einer generierten Klasse erbt. Ist eine solche Bedingung nicht erfüllt, wird ein entsprechender Hinweis an den Entwickler ausgegeben.

Sprich Metamodell

Eric Evans hat in seinem Buch über *Domain-driven Design* [5] das Konzept der *Ubiquitous Language*, also einer allgegenwärtigen Sprache, beschrieben. Dabei geht es im Grunde darum, die Konzepte der Fachlichkeit zu benennen und damit eine einheitliche Sprache zu finden, in der alle Projektbeteiligten kommunizieren können. Auf diese Weise werden die Konzepte klar herausgearbeitet und somit zu einem wichtigen und „lebendigen“ Bestandteil des Projekts. Die Konzepte und deren Bezeichnungen finden sich auf allen Ebenen des Projekts wieder.

Verwendet man dieses Muster auch für die Konzepte der Metamodelle, bekommen auch die technischen Konzepte eindeutige Namen und eine eindeutige Bedeutung. Durch die konsequente Verwendung der Begrifflichkeiten, beispielsweise in Teambesprechungen, können Aspekte klarer beschrieben wer-

den. Außerdem fällt schneller auf, wenn die gewählten Konzepte nicht mehr so gut zu den Anforderungen passen. Dann muss die DSL an die veränderten Rahmenbedingungen angepasst werden.

Entwickle DSLs iterativ

Domänen spezifische Sprachen werden nicht mittels eines *Big-Up-Front-Ansatzes* entwickelt, sondern entstehen idealerweise, genau wie herkömmliche APIs, inkrementell: Zuerst werden die offensichtlichen Kernabstraktionen definiert und dann werden nach und nach, wenn das Verständnis für die zu beschreibende Domäne wächst, einzelne Dinge hinzugefügt oder auch wieder entfernt.

DSLs sind allerdings öffentliche Schnittstellen, also gelten hier die Grundsätze der API-Entwicklung. Je nach Vereinbarung über den Lebenszyklus der Schnittstelle dürfen beispielsweise bestehende Verträge bei der Weiterentwicklung nicht gebrochen werden.

In jedem Fall hat auch die der DSL zugrunde liegende Technik einen enormen Einfluss auf die Migrationsfähigkeiten. Um ein Modell manuell zu migrieren, also zum Beispiel nicht mehr vorhandene Konzepte durch neu eingeführte zu ersetzen, ist es notwendig, das alte Modell im Editor laden zu können. Bei objektbasierter Speicherung, z.B. über XMI oder in einer Datenbank, ist das Laden alter Modelle oft nicht mehr möglich, da die gespeicherten Modelle aufgrund der Änderungen nicht mehr kompatibel sind. Die Migration muss dann entweder programmatisch mittels einer Modelltransformation vom alten in das neue Metamodell erfolgen, oder man begibt sich auf die Ebene des Speicherformats und versucht, die alten Informationen kompatibel zu machen.

Wenn die konkrete Syntax der DSL dieselbe ist, in der das Modell gespeichert wird (sprich textuell), fällt eine Migration natürlich sehr viel leichter.

Entwickle Modellvalidierung iterativ

Modellgetriebene Verfahren der Softwareentwicklung basieren darauf, dass die verarbeiteten Modelle formal und maschinenlesbar vorliegen. Die Form eines Modells wird durch sein Metamodell

vorgegeben, das somit auch gleichzeitig Grundlage für das verwendete Modellierungs-Tool sein sollte (es gibt tatsächlich Modellierungs-Tools, die unabhängig von Metamodellen operieren und demzufolge zu großen Problemen in MDSD Toolchains führen). Nun sollte man ja davon ausgehen, dass ein vernünftiges Modellierungs-Tool nur formalkorrekte Modelle erzeugen kann, die vom Generator problemlos verarbeitet werden können. Doch dies ist leider ein Trugschluss, da das Metamodell nur die Statik der erzeugten Modelle beschreibt. Die Semantik kann mit diesen statischen Informationen nicht ausgedrückt werden. Ein Beispiel: „In einem Zustandsautomaten darf jeder Zustand nur einmal vorkommen.“ Diese Bedingung kann in den üblichen Metamodellierungssprachen nicht ausgedrückt werden. Zur Lösung dieses Problems wurden *Constraint-Sprachen* wie z.B. OCL und Check erfunden. Mit ihnen können Bedingungen wie die eben geschilderte ausgedrückt werden.

Die Bedingung „In einem Zustandsautomaten muss jeder Zustand über eine Transition mit einem anderen Zustand verbunden sein“ könnte man wahlweise auch direkt im Metamodell (durch Multiplizitäten 1...n) implementieren. In solchen Fällen ist oft trotzdem ein *Constraint* vorzuziehen: Zum einen bleibt das Metamodell dadurch überschaubarer, zum anderen sind Zwischenzustände, die beim Entwurf eines Modells auftreten, trotzdem schemakonform, was bei vielen Tools die Grundvoraussetzung für die Speicherung ist.

Die Entwicklung der *Constraints* soll iterativ erfolgen: Zunächst werden alle offensichtlich sinnvoll erscheinenden *Constraints* definiert. Sobald im Verlauf des Projekts Generierungsfehler auftauchen, die auf eine unvollständige Modellüberprüfung zurückzuführen sind, ergänzt man die *Constraints* so, dass auch dieser Fall abgedeckt wird. Auf diese Weise wird die Modellvalidierung stetig lückenloser.

Teste den Generator mittels Referenzmodell

Auch Codegeneratoren sollten getestet werden, und zwar am besten mittels einer automatischen Test-Suite. Doch

wie wird diese implementiert? Eine weit verbreitet Variante ist, den generierten Quelltext einfach mit dem erwarteten Quellcode zu vergleichen. Das ist allerdings eine sehr fragile Konstruktion, denn selbst die kleinste Änderung am Generator hat zur Folge, dass die Tests gebrochen sind. Eigentlich möchte man ja sicherstellen, dass sich der generierte Code auf der Zielplattform den Anforderungen entsprechend verhält.

Das kann man sehr einfach durch Unit-Tests für das Generator erreichen. Ein entsprechendes Framework gibt es für die meisten Zielsprachen. Dazu werden *Referenzmodelle* erstellt, die die Konzepte der DSL exemplarisch verwenden. Exemplarisch bedeutet hier, dass man nicht irgendeine Fachlichkeit „erfindet“ und diese als Beispiel verwendet, sondern die Modellelemente danach benennt, welche DSL-Konzepte sie darstellen. Ein Referenzmodell für eine Domänenmodell-DSL könnte zum Beispiel folgendermaßen aussehen:

```
abstract entity AbstractEntity {
    attr String oneStringAttr
    attr String[] manyStringAttr
    ref SimpleEntity toOneReference
}

entity SimpleEntity {
}
// usw...
```

Idealerweise besteht die Test-Suite aus mehreren solchen Referenzmodellen und dazugehörigen Tests, die unterschiedliche Aspekte und Kombinationen beschreiben. In späteren Iterationen können Bugs mithilfe eines neuen Referenzmodells nachgestellt werden, bevor sie behoben werden. Die bestehenden Referenzmodelle und die dazugehörigen Tests werden dabei nicht beeinträchtigt.

Übrigens können und sollten natürlich auch Modellvalidierung und Modelltransformationen auf diese Weise getestet werden.

Wähle die passende Technologie aus

Es gibt sehr viele verschiedene Arten, Modelle darzustellen und zu editieren.

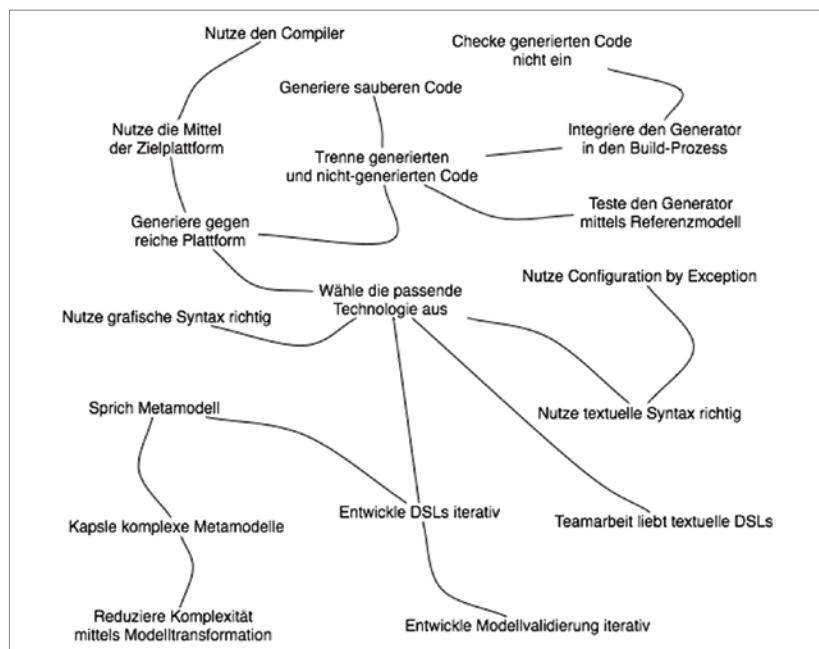


Abb. 4: Die Best Practices im Überblick

Welche Technologie und Syntax geeignet ist, hängt von den jeweiligen Rahmenbedingungen des Projekts ab. Hier ist nicht nur die Art der Domäne ausschlaggebend, sondern auch die Frage nach der Flexibilität der Technologie oder wie groß der Aufwand bei der Entwicklung im Vergleich zu anderen Alternativen ist. Eine weitere wichtige Frage ist, wie gut sich die Technologie in den Entwicklungsprozess und die bestehende Werkzeuglandschaft der Anwendungsentwickler integrieren lässt. Eine Integration in die Entwicklungsumgebung (z.B. Eclipse) wird heute schon fast vorausgesetzt.

Idealerweise stellt die Syntax die Kernabstraktionen deutlich heraus und lässt sich mit wenig Aufwand erweitern. Sie sollte gleichzeitig sowohl anschaulich als auch schnell editierbar sein. Das Feedback bei der Modellierung sollte schnell und umfassend sein und die Turnaround-Zeiten (also modellieren, generieren, ausführen) möglichst kurz.

Die ideale Technologie gibt es leider noch nicht. Modellierung mit der UML beispielsweise bedeutet oft einen geringen initialen Aufwand bei der Definition der Sprache: Die Definition eines Profils. Dafür ist der Turnaround meist sehr lang. Weder DSL noch Werkzeug lassen sich einfach erweitern und man muss sich bei der Verarbeitung der Modelle mit einem sehr großen und komplexen Metamodell

auseinandersetzen. Nach unserer Erfahrung lohnt sich in den allermeisten Projekten der Einsatz von „richtigen“ DSLs, die exakt auf die Anforderungen der jeweiligen Domäne zugeschnitten sind und keinen Ballast enthalten.

Kapsle UML (und andere komplexe Metamodelle)

Sollte die Wahl der Technologie trotz aller Warnungen auf die UML fallen, sollte man die UML-Modelle vor der Weiterverarbeitung auf ein einfacheres Metamodell für ihre Domäne übersetzen. Durch diese Maßnahme werden die nachgelagerten Verarbeitungsschritte vereinfacht und somit weniger fehleranfällig. Die Validierung des Modells sollte dann ebenfalls nach der Transformation stattfinden. So ist sichergestellt, dass auch die Constraints einfach und gleichzeitig spezifisch für die Domäne formuliert werden können.

Diese Entkopplung bietet weiterhin die Möglichkeit, zu einem späteren Zeitpunkt auf eine „richtige“ DSL umzusteigen, ohne den Generator dabei verändern zu müssen.

Nutze grafische Syntax richtig

Grafische Modellierung hat durchaus Vorteile. Sie stellt Beziehungen zwischen wichtigen Elementen übersichtlich und einfach dar. Viele Informationen können schneller erfasst werden, weil die In-

formation nicht ausformuliert, sondern visuell dargestellt ist.

Wenn grafische Modelle eingesetzt werden, sollte man sich die Möglichkeiten der Visualisierung zunutzen machen und versuchen, eine Syntax zu finden, die die wichtigsten Eigenschaften der Modelllemente visuell darstellt. Die UML stellt in diesem Zusammenhang ein sehr nützliches Vorbild dar. So werden hier z.B. abstrakte Klassen durch Kursivschrift des Namens gekennzeichnet. Die Annotation mit Operatoren, zum Beispiel #, +, - für die Darstellung der Sichtbarkeit eines Attributs kann sowohl in grafischen als auch in textuellen DSLs genutzt werden.

Aber Vorsicht! Eine Syntax kann sehr schnell mit Informationen überladen werden. Die Diagramme sollten auf jeden Fall ein ruhiges Bild ergeben, beispielsweise darf Text nicht in allen möglichen Varianten und Schriftarten erscheinen. Es kommt auf den richtigen Kompromiss an. Die Grundprinzipien des Screendesigns können hier hilfreiche Tipps geben.

Nutze textuelle Syntax richtig

Bei der Definition einer textuellen Syntax ist es wichtig, einen guten Kompromiss zwischen Kompaktheit und Ausführlichkeit zu finden. Je nachdem, wie viel Wissen dem Modellierer zugetraut werden kann, kann auf erklärende Schlüsselwörter verzichtet werden.

Im Folgenden vergleichen wir die von Java bekannte Syntax zur Definition einer abstrakten Klasse mit einer fiktiven, aber möglichen Alternative:

JAVA: public abstract class Foo

FIKTIV: class name=Foo visibility=public
abstract=true

Es ist wohl unumstritten, dass die Java-Syntax lesbarer ist als die fiktive Syntax. Allerdings ist sie auch nicht so selbst-erklärend. Bei der fiktiven Syntax wird explizit gesagt, welche Eigenschaft welchen Wert bekommt. Der Anwender muss also nur das Metamodell kennen, die Syntax ist klar.

XML funktioniert genauso und es ist tatsächlich ein kleiner, wenn auch aus

unserer Sicht nicht ausschlaggebender, Pluspunkt für XML: Die Syntax ist klar und einfach zu verstehen.

Nutze Configuration By Exception

Ein weiterer wichtiger Punkt ist die richtige Wahl und der überlegte Einsatz von Voreinstellungen (*Configuration by Exception*). Im Gegensatz zu einer grafischen Syntax, bei der meist in einem Eigenschaftsfenster alle Informationen angezeigt werden, können bei der textuellen Modellierung Informationen einfach weggelassen werden, wenn sie dem Default entsprechen.

Wir wollen dies ebenfalls an Java und einer fiktiven Alternativsyntax illustrieren:

```
Java : class Foo
Fiktiv : class Foo visibility=package final=false
abstract=false
```

Wie man unschwer erkennen kann, erhöht die richtige Wahl der Voreinstellungen die Lesbarkeit enorm. Dabei sollte man sich allerdings von vornherein klar machen, dass diese auch ein Teil der API sind. Voreinstellungen können im Nachhinein nicht geändert werden, ohne bestehende API-Verträge zu brechen.

Teamarbeit liebt textuelle DSLs

Klassische Modellierungs-Tools haben ein Problem mit der Teamarbeit. Grund ist die Art der Speicherung der Modelle: So speichern UML-Tools ihre Modelle üblicherweise in einer einzigen großen XMI-Datei. XMI (XML Metadata Interchange) ist ein sehr technisches und generisches Datenformat. Wer schon einmal versucht hat, einen Konflikt auf Basis eines XMI-Modells zu lösen, weiß, welche Probleme das bereitet. Deshalb sollte man in diesem Fall versuchen, Konflikte zu vermeiden, indem man beispielsweise nur exklusiven Schreibzugriff auf das Modell erlaubt oder die Modelle feingranular partitioniert.

Um das Problem an der Wurzel zu fassen, sollte man darüber nachdenken, die Modelldaten in einem lesbaren und wartbaren Format zu speichern. Es ist heute durchaus möglich, mehrere Syntaxen für eine DSL zu haben und beispiels-

weise einen grafischen Editor für eine textuelle DSL zu nutzen.

Viele professionelle UML-Werkzeuge bieten Teamserver an, die auf unterschiedliche Weise das Modellieren im Team ermöglichen. Dabei entsteht aber leider ein Bruch in der Werkzeugkette, dazur Synchronisation des Quellcodes – MDSD-Modelle sind Quellcode! – zwei verschiedene Werkzeuge und zwei verschiedene Repositories verwendet werden müssen. Das hat weitere negative Auswirkungen, beispielsweise auf den Integrations-Build.

Nutze Modelltransformation zur Reduktion der Komplexität

Komplexe Codegeneratoren entstehen, wenn die Übersetzung – der Schritt vom Modell zum Code – besonders groß ist. Ein Mittel, um über die Komplexität Herr zu werden, ist die Aufteilung des Generators in zwei Teile, frei nach dem Motto „Divide and Conquer“.

Dabei wird ein Metamodell entworfen, das möglichst nah am zu generierenden Code liegt. Eine Modelltransformation übersetzt die Input-Modelle in Zwischenmodelle vom Typ des neuen Metamodells. Der Codegenerator muss nun nur noch vom Zwischenmetamodell auf die Zielpлатzform abbilden.

Dieses Vorgehen hat den angenehmen Nebeneffekt, dass durch das Metamodell eine klare Schnittstelle entsteht, und so die beiden Teile von verschiedenen Teams entwickelt und unabhängig voneinander in anderen Szenarien wieder verwendet werden können.

Natürlich kann man beliebig viele Zwischenschritte dieser Art in einen Codegenerator integrieren, die Praxis hat aber gezeigt, dass eine Schicht (in vielen Fällen auch keine) völlig ausreicht. Die MDA empfiehlt zur Nutzung von Modelltransformationen viele kaskadierte Schichten zur Entkopplung der verschiedenen Plattformen. Das ist aus Sicht der Autoren nicht praxistauglich. Hier gilt aber sicher: Ausnahmen bestätigen die Regel.

Generiere gegen eine reichhaltige Plattform

Ein weiteres wichtiges Mittel, um den Übersetzungsschritt vom Model zum

Lösungen

aus der Praxis für die Praxis!

Als **Software- und Beratungshaus** mit über 15 Jahren Projekterfahrung bieten wir Lösungen und Tools, die direkt aus dem Praxisalltag und den Bedürfnissen der betrieblichen Anwendungsentwicklung entstanden sind.

Wir helfen Ihnen bei der Gestaltung **durchgängig modellgetriebener Entwicklungsprozesse** – von den Anforderungen bis zur lauffähigen Anwendung.

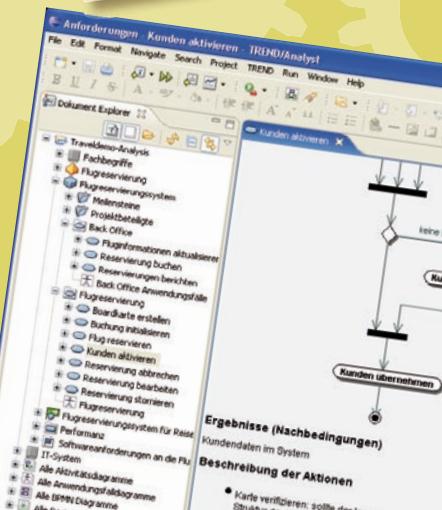
Requirements Engineering mit Eclipse

Testen Sie die kostenlose Community Edition unseres **TREND/Analyst Requirements Engineering Werkzeugs**.

- Komplett in Eclipse integriert
- UML-Modelle & Texte perfekt verbunden
- Einfache Versionierung
- Maximale Anpassbarkeit
- Flexible Export-Möglichkeiten
- Gemeinsame Plattform für Entwicklung und Fachabteilung

www.gebit-community.de

Kostenloses RE-Tool
für den professionellen Einsatz!
Jetzt mit erweitertem Metamodell!



www.gebit.de

Berlin • Düsseldorf • Stuttgart

Code möglichst klein zu halten und damit die Komplexität des Generators zu verringern, ist die Entwicklung einer Plattform, die dem Generator so weit wie möglich entgegenkommt.

Statt beispielsweise eine Persistenzschicht zu generieren, sollte ein geeignetes Framework (z.B. eine EJB-3-Implementierung) verwendet werden. Darauf werden zusätzlich projektspezifische Abstraktionen entwickelt (z.B. eine *AbstractEntity*-Klasse), gegen die der Code generiert wird.

Partitioniere deine Metamodelle richtig

Sinnvoller ist jedoch, bereits die Metamodelle geschickt zu partitionieren. Die Bildung von Schichten ist eine bewährte Praxis des Softwareengineering, und wir können uns dieses Prinzip auch hier zunutze machen: Wir partitionieren die Metamodelle

entlang der technischen Schichten der Domäne(n), die modelliert werden sollen.

Fazit

Die vorliegenden *Best Practices* spiegeln unsere Erfahrungen (und die unserer Kollegen) aus mehreren Jahren MDSD in der Praxis wider. Unser wichtigster Ratschlag an die Leser ist: Sei pragmatisch. Richtig eingesetzt können DSLs und Codegeneratoren ein sehr wertvolles Werkzeug sein. Im Mittelpunkt sollte aber immer das zu lösende Problem stehen. In vielen Fällen macht es Sinn, bestimmte Aspekte mittels DSLs zu beschreiben, aber nicht in allen. Projekte, die up-front entscheiden modellgetrieben vorzugehen, verletzen diesen letzten Rat. Wenn Sie unsicher sind, ob und wie sie MDSD-Technologie einsetzen sollten, lassen Sie sich beraten. ■



Sven Efftinge leitet die Kieler Niederlassung der itemis AG. Er ist Projektleiter im Textual Modeling Framework (TMF) sowie Committer bei anderen Eclipse-Modeling-Projekten. Sven ist weiterhin Architekt und Entwickler von openArchitectureWare 4 und dem Xtext Framework.



Peter Friese ist Softwarearchitekt und MDSD-Experte bei der itemis AG in Kiel. Er verfügt über umfangreiche Erfahrung im Einsatz von modellgetriebenen Verfahren der Softwareentwicklung und in der Entwicklung von Softwareentwicklungs-Tools. Peter ist Committer im Eclipse Modeling Project sowie für die Open-Source-Projekte openArchitectureWare, AndroMDA und FindBugs.



Dr. Jan Köhnlein arbeitet ebenfalls als Softwarearchitekt bei der itemis AG in Kiel. Er entwickelt seit mehreren Jahren Entwicklungs-Tools für MDSD und ist Committer für Eclipse Modeling und openArchitectureWare.

Die drei Autoren sind Teil der itemis labs in Kiel [6], die sich auf die Weiterentwicklung der Eclipse-Modeling-Projekte konzentrieren und für diese professionelle Dienstleistungen anbieten.

Links & Literatur

- [1] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase: Modellgetriebene Softwareentwicklung, dpunkt.verlag 2007.
- [2] Markus Völter, Jörn Bettin: Patterns for MDSD: www.voelter.de/data/pub/MDDPatterns.pdf
- [3] Peter Friese: How to distinguish generated code from non-generated code: www.peterfriese.de/how-to-distinguish-generated-code-from-non-generated-code/
- [4] openArchitectureWare: www.openarchitectureware.org/
- [5] Eric Evans: Domain-Driven Design.Tackling Complexity in the Heart of Software. Addison-Wesley Longman 2003.
- [6] itemis labs kiel: oaw.itemis.de
- [7] Thomas Hunt: The Pragmatic Programmer. Addison-Wesley 2000.