mehr zum thema:
**http://www.actifsource.com**
**http://actifsource.blogspot.com**

# MODELLE IM RAMPENLICHT: VORAUSSETZUNGEN FÜR AGILES ARBEITEN MIT MODELLEN

Nicht jedes Modellierungswerkzeug lässt sich gleich gut in bestehende Softwareentwicklungsprozesse integrieren. Wenn Modelle ganze Geschäftsbereiche abbilden, kann dies schnell zur Herausforderung für die Tools werden. Das ist somit ein Risiko für die entstehende Software. Durch die kritische Überprüfung einiger Voraussetzungen können solche Risiken minimiert werden und die Chancen, dass das Entwicklungsteam zufrieden und agil modellgetrieben arbeiten kann, erhöhen sich. Dieser Artikel soll zeigen, wie Modelle sorgenfrei und dynamisch eingesetzt werden können. Des Weiteren soll er als Katalog dienen, der alle wesentlichen Voraussetzungen auflistet.

die autoren

Micha Riser
(E-Mail: micha.riser@actifsource.com)
arbeitet als Softwareingenieur und leitet die Entwicklung des modellgetriebenen Codegenerators actifsource.

Reto Carrara
(E-Mail: reto.carrara@actifsource.com)
ist Geschäftsführer der actifsource GmbH. Er arbeitet seit Jahren als Berater für modellgetriebene Softwareentwicklung.

In der Softwareentwicklung werden schon seit langem Modelle verwendet. Früher waren dies vor allem graphische Modelle, die allein der Dokumentation der Software dienten. Kleine UML-Diagramme, die Ausschnitte der Softwarestruktur darstellen, sind in fast jeder Softwaredokumentation vorhanden.

In letzter Zeit finden Modelle aller Art immer größere Verendung. Vorträge zur modellgetriebenen Softwareentwicklung (*Model-Driven Software Development – MDSD*) werden auf vielen Fachkonferenzen gehalten. Bei diesem Ansatz versuchen die Entwickler oftmals, die ganze Domäne der jeweiligen Software in einem Modell zu erfassen (oder, falls die Werkzeuge es erlauben, in mehreren miteinander vernetzten Modellen). Eine komplexe Domäne umfasst jedoch oft mehrere hundert oder tausend Modellelemente. Der MDSD-Ansatz erstellt Teile der Software per Codegenerierung aus den Modellen (siehe Abbildung 1). So hängt die Software unmittelbar von den Modellen ab. Das bringt ganz neue Forderungen an die Werkzeuge mit sich. Der Umgang mit Modellen solcher Größe kann zur Last werden, wenn gewisse Voraussetzungen nicht gegeben sind. Dieser Artikel beschreibt Bedingungen an die Infrastruktur und an die verwendeten Werkzeuge, die erfahrungsgemäß erfüllt sein müssen, um erfolgreich modellgetrieben zu entwickeln. Eine Zusammenfassung findet sich in Kasten 1.

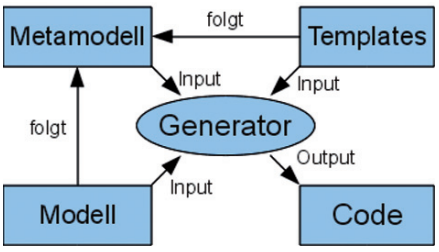


*Abb. 1: Schematische Darstellung von modellgetriebener Codegenerierung.*

## Echtzeit-Validierung

Alle Modelle, die in die Entstehung des Codes involviert sind, sollten automatisch neu validiert werden, sobald sich irgendwelche Elemente ändern. Das ist wichtig, da es um so einfacher ist, Fehler zu korrigieren, je eher sie erkannt werden. Dank moderner Entwicklungsumgebungen erkennen Entwickler Fehler im Quellcode noch während des Codierens. Deshalb sollte diese Anforderung auch beim Erstellen und Bearbeiten von Modellen erfüllt sein (siehe Abbildung 2). Sieht der Entwickler Modellfehler erst, wenn sie zu einem Abbruch des Build-Prozesses führen, kostet das wertvolle Zeit.

## Persistenz von fehlerhaften Modellen

Ein Texteditor ermöglicht es, Quellcode jederzeit zu speichern, auch wenn dieser syntaktisch fehlerhaft ist. Genauso ist es für den Modellierer wichtig, Modelle jederzeit speichern zu können, auch wenn sie Fehler enthalten. Wieso ist dies überhaupt not-

Checkliste für Modellierungswerkzeuge:

| | |
|---|---|
| ■ Zusammenarbeit mit Versionsverwaltungswerkzeugen | *** |
| ■ Skalierbarkeit | *** |
| ■ Lesbarkeit der Codetemplates | *** |
| ■ Integration in die Entwicklungsumgebung | ** |
| ■ Echtzeitvalidierung | ** |
| ■ Persistenz von fehlerhaften Modellen | ** |
| ■ Möglichkeiten zur Modell-Refaktorisierung | ** |
| ■ Möglichkeiten zur Modell-Strukturierung | ** |
| ■ Graphische Manipulation von Modellen | ** |
| ■ Integration ins Build-System | * |

*Kasten 1: Möglichkeiten, die gegeben sein müssen, damit sich auch große Modelle über einen längeren Zeitraum verwalten lassen (*** = sehr wichtig, ** = wichtig, * = weniger wichtig).*

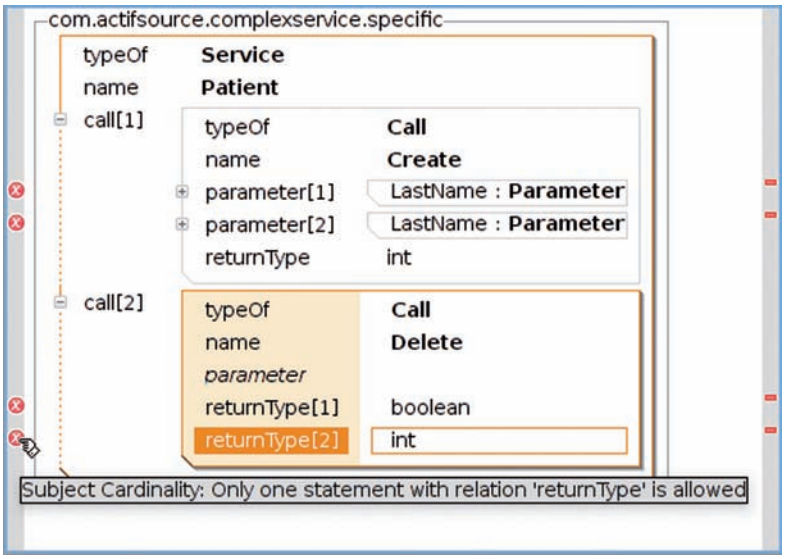


*Abb. 2: Echtzeit-Validierung von Modellen.*

wendig? Plötzlich auftauchende Probleme können einen Entwickler jederzeit zwingen, seinen Arbeitsprozess zu unterbrechen. Ein Modellierungswerkzeug darf ihn deshalb nicht daran hindern, einen halbfertigen Entwicklungsstand zu speichern. Solange er ein solches Modell nicht über ein Repository für andere zugänglich macht, sind temporäre Modell-Inkonsistenzen auch kein Problem.

## Versionierung mit dem Quellcode

Modelle verändern sich mit der Zeit. Es ist wichtig, dass alte Versionen nicht verloren gehen. Zu einem gewissen Stand des Quellcodes gehört immer auch eine gewisse Modellversion. Passt der Stand des Modells nicht zum Code, lässt sich die Software in der Regel nicht mehr bauen. Eine wichtige Anforderung für die Entwicklung und Wartung von Software ist deshalb, dass ein alter Stand jederzeit komplett und einfach wiederhergestellt werden kann. Hier hat die Erfahrung gezeigt, dass es am besten ist, wenn die Infrastruktur die Modelle nicht in einer zentralen Datenbank, sondern in einzelnen Textdateien ablegt. Diese haben den Vorteil, dass sie sich zusammen mit dem Quellcode in einem einzigen Tool zur Versionsverwaltung (z. B. „CVS" oder „SVN") speichern lassen. Das Anzeigen und Zusammenführen von Änderungen funktioniert mit den gängigen Versionierungswerkzeugen so zumindest auf Textbasis. Dies auf Basis des Textes zu tun, ist zwar nicht optimal, da der Entwickler meist gezwungen ist, mit kryptischen XML-Dateien umzugehen. Sind lediglich Binärdaten einer Datenbank vorhanden, bleibt diese Notfalllösung ganz außen vor. Auch ist es ineffizient, bei jeder Änderung am Modell die gesamte Datenbank in die Versionsverwaltung einzuchecken. Bietet die Modell-Datenbank eine eigene Versionierung an, besteht dennoch das Problem, dass Quellcode und Modelle nicht gemeinsam versioniert sind. Wenn ein Werkzeug den Modellierer mit *Diff* und *Merge* auf der Modellebene unterstützt, ist das natürlich noch besser (siehe Abbildung 3). Graphische Modelle haben hier das Problem, dass sowohl der Inhalt als auch die Position der Elemente gleichzeitig behandelt werden müssen. Zeilenweise aufgebaute oder baumartige Darstellungen der Modelle (zeilenweise mit Verschachtelungen) sind deshalb einfacher zu handhaben. So erstaunt es nicht, dass Modellierungswerkzeuge, die zeilenbasiert sind, auch häufiger Unterstützung für das Zusammenführen von Modelländerungen anbieten.

## Auch Metamodelle ändern sich

Jedes Modell folgt in seiner Struktur einem gewissen Metamodell. Bei einem UML-Diagramm ist dies eine bestimmte Version des UML-Standards. Im Fall einer domänenspezifischen Sprache ist dies eine vom Entwickler definierte und der Domäne angepasste Syntax. Obwohl sich Metamodelle deutlich seltener ändern als die Modelle an sich, sind auch sie nicht vor Änderungen gefeit (vgl. [Fav03]). Der Versuch, das Metamodell einer bestehenden, modellbasierten Software mit vielen Elementen anzupassen, stößt – je nach Werkzeug – auf unterschiedlichen Widerstand. Das Metamodell lässt sich in der Regel immer anpassen. Im schlimmsten Fall kann das Modellierungswerkzeug dann aber bestehende Modelle nicht mehr öffnen, weil sie nicht zum geänderten Metamodell passen. Das Modell müsste also von Grund auf neu erstellt werden, was natürlich nicht tolerierbar ist. Faktisch gesehen bedeutet das, dass in einem solchen Fall das Metamodell nicht mehr änderbar ist, sobald es eine bestimmt Größe erreicht hat, was einer agilen Entwicklung ganz zuwider läuft.

Ein gutes Modellierungswerkzeug erkennt, an welchen Stellen das alte Metamodell nicht mehr zum neuen passt, und markiert fehlende und nicht mehr länger gültige Modellelemente als Fehler. Diese kann der Entwickler dann auf die

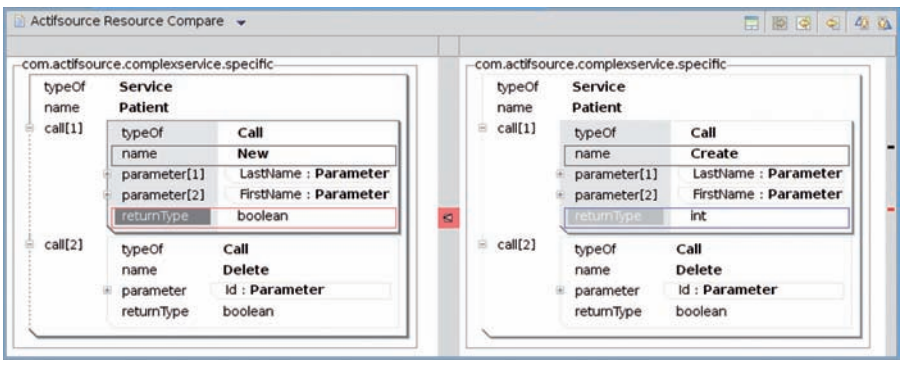


*Abb. 3: Konflikteditor für baumartig strukturierte Modelle.*

gleiche Weise korrigieren, wie er normale Modelländerungen durchführt. Je nach Größe der Modelle ist das jedoch eine mühselige Arbeit und schränkt somit wiederum die Agilität ein. Noch besser ist es, wenn die Möglichkeit vorhanden ist, den Umbau von der alten auf die neue Version des Metamodells generisch zu beschreiben. Solche automatischen *Model Refactorings* finden bei den aktuellen Versionen der Tools zur Zeit jedoch kaum Beachtung. In [Wac07] wird beschrieben, wie sich Metamodelle evolutionär ändern und die Modelle sich dabei mit ändern müssen, um kompatibel zu bleiben. Dabei wird für jede Transformation des Metamodells eine entsprechende Modelltransformation beschrieben.

Praktische Ansätze zur Automatisierung bieten Programmierschnittstellen, über die die Modelle modifiziert werden können, oder die Modelltransformationssprache *ATL (Atlas Transformation Language)*, bei der eine Refaktorisierung als eine Transformation vom alten auf das neue Metamodell beschrieben wird (vgl. [Fau07]). Ein auf diese Weise einmal definierter Umbau lässt sich dann auf alle bereits vorhandenen Modelle anwenden.

## Lesbarkeit der Templates

Ein Codegenerator erzeugt mit Hilfe von Templates Code aus den Modelldaten. Robert C. Martin schreibt in seinem Buch „Clean Code" (vgl. [Mar08]), wie wichtig es ist, dass Code lesbar geschrieben wird, und zeigt Wege auf, dies zu erreichen.

Auch Templates müssen gelesen und gewartet werden. Unsere Erfahrung hat gezeigt, dass die Entwickler deutlich mehr Zeit für Anpassungen an den Templates als für Änderungen der Modelle aufwenden. Templates vermischen Zielsprache mit Referenzen auf das Modell und Anweisungen, wie das Modell iteriert werden soll. Dazu führt das Modellierungswerkzeug in der Regel eine neue Sprache ein (das „Eclipse Modelling Framework" (vgl. [EMF]) bietet mit „Xpand" und „JET" gleich zwei zur Auswahl). Das Vermischen beider Sprachen in einem Text macht die Templates jedoch schwieriger zu lesen, was dem Prinzip der *Clean Templates* entgegenläuft. Eine andere Möglichkeit besteht darin, die Iterationen des Templates graphisch darzustellen und Referenzen auf die Modelle ähnlich wie Hyperlinks auf einer Webseite zu markieren. Auf diese Weise bleibt der Template-Code frei von zusätzlichen *Markup-Tags* (siehe Listing 1 und Abbildung 4).

```
«IMPORT demo::wizard»
«EXTENSION WizardExtensions»

«DEFINE main FOR Wizard»
«FILE this.packageSubDir()+"/"+Wizard.name+".java"»

package «this.package()»;

public class «this.name» extends GeneratedWizard implements «this.interface()» {

  «FOREACH this.page AS Page-»
  private «Page.name» «Page.pageFieldName()»;
  «ENDFOREACH-»

  @Override
  public void addPages() {
    «FOREACH this.page AS Page-»
    create«Page.name»();
    «ENDFOREACH-»
  }

  «FOREACH this.page AS Page-»
  private void create«this.name»() {
    «Page.pageFieldName()» = new «Page.name»();
    addPage(«Page.pageFieldName()»);
  }
  «ENDFOREACH-»

}

«ENDFILE»
«ENDDEFINE»
```

*Listing 1: Xpand Template mit Tags für Generatoranweisungen.*

## Skalierbarkeit

Bevor man sich entschließt, die gesamte Geschäftsdomäne mit Hilfe eines Modellierungswerkzeugs abzubilden und daraus die Software zu generieren, sollte man sicherstellen, dass das Werkzeug auch bis zur Größe des endgültigen Modells skaliert. Während ein Modell für einen Machbarkeitsnachweis, wie es im Rahmen eines Prototypen vorkommt, in der Regel keine Herausforderung für die Werkzeuge ist, sollte man folgende Punkte mit der erwarteten Größe des vollständigen Modells abklären:

- Wie groß werden die gespeicherten Modelldaten?
- Stellen die Datenmengen ein Problem für die Netzwerkinfrastruktur dar?

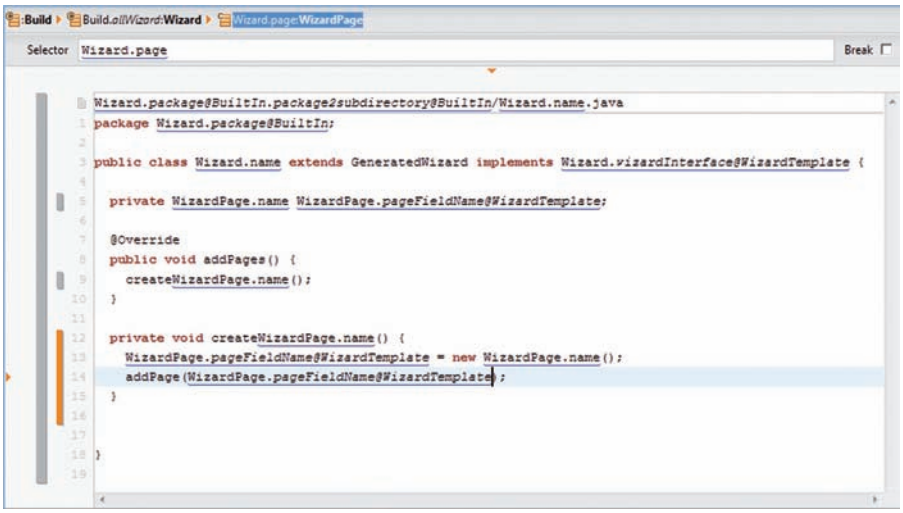


*Abbildung 4: Actifsource Template mit graphischen Markierungen anstatt spezieller Template-Sprache.*

- Wie lange dauert es, Modelle zu öffnen und zu ändern?
- Wie lange wird der Build-Prozess voraussichtlich dauern?

Diese Abklärungen mögen mit einigem Aufwand verbunden sein, können dem Projekt aber großen Ärger ersparen, wenn früh festgestellt wird, dass eine angedachte Lösung später zu Problemen führt. Um nicht riesige Modelle nur für den Test erstellen zu müssen, kann man das Verhalten der Zeitdauer bzw. der Datengröße beobachten, wenn man z. B. die Menge der Elemente verdoppelt. Daraus lassen sich dann die Daten für die erwartete Modellgröße grob extrapolieren.

## Strukturierung von Modellen

Je größer ein Modell ist, desto stärker wird der Wunsch, ein Modell zu unterteilen. Graphische Modelle werden unübersichtlich, wenn sie eine bestimmte Anzahl von Elementen übersteigen. Dabei hilft es, wenn zwischen einer Grob- und einer Detailansicht der Elemente umgeschaltet werden kann. Doch sobald Verbindungslinien anfangen, sich unweigerlich zu schneiden, sind die Grenzen dessen erreicht, was gleichzeitig auf einem Bildschirm darstellbar ist. Wir brauchen also eine weitere Möglichkeit zur Strukturierung, die möglichst hierarchisch sein sollte. Eine nahe liegende Lösung ist es, Modellelemente in Paketen zu gruppieren. Dieses Konzept ist schon von der Programmiersprache Java bekannt und lässt Hierarchien zu, da Pakete wiederum untergeordnete Pakete enthalten können. Die ganze höherwertige Struktur kann so in einer Übersicht als Baum dargestellt werden.

Einige Werkzeuge bieten die Möglichkeit, verschiedene Modelle zu erstellen und diese miteinander zu verknüpfen. Dabei hängt ein Modell jeweils von einem oder mehreren anderen Modellen ab, die es so erweitern kann. Das bietet die Möglichkeit, die Domäne in verschiedene Bereiche zu unterteilen und jeden Bereich jeweils in ein eigenes Modell abzubilden. Erfahrungsgemäß erleichtert das die Verbreitung von Modellen bei der Softwareentwicklung: Ein Kernmodell kann für eine bestimmte Domäne in einem separaten Modell um zusätzliche Informationen angereichert werden, ohne dabei das Kernmodell zu verändern. Ebenso ist es möglich, dass ein drittes Modell zwei unabhängig voneinander entstandene Modelle miteinander verknüpft (siehe Abbildung 5). So hilft der Multimodell-Ansatz, dass Modelle zur zentralen Informationsquelle werden, die die Informationen der Domäne für die Softwareentwicklung jederzeit verfügbar machen. Da die Informationen so nur einmal vorhanden sind, verhindert man damit Redundanzen und Inkonsistenzen.

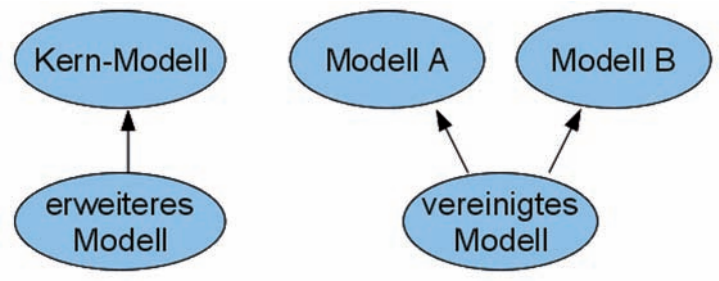


*Abb. 5: Zwei mögliche Arten, Modelle in einem Multimodell-Ansatz aufzuteilen.*

## Integration mit dem Build-System

Werden die Modelle für die Generierung von Code eingesetzt, muss die Codegenerierung in den Entwicklungsablauf eingebaut werden. Die Entwickler müssen den automatisch erstellten Code mit den von Hand geschriebenen Teilen integrieren. Wenn das alles in einer einzigen Entwicklungsumgebung stattfindet, kann der ganze Ablauf vollständig automatisiert werden: Sobald ein Entwickler seine Änderungen des Modells speichert, erstellt das Werkzeug den generierten Teil der Software neu. Anschließend prüft der Compiler die Integration mit dem restlichen Code auf Fehler und erstellt das Gesamtsystem neu. Um lange Build-Zeiten zu vermeiden, sollte der Codegenerator generierte Dateien nicht neu schreiben, wenn sie sich durch die Änderungen am Modell nicht verändert haben. Können so alle Schritte des Builds inkrementell durchgeführt werden, dauert dieser – je nach Programmiersprache und Größe der Änderungen – bloß einige Sekunden bis hin zu wenigen Minuten.

Setzt die Softwareentwicklungsabteilung bereits ein Tool wie „CruiseControl" oder „Hudson" zur kontinuierlichen Integration ein, sollte geprüft werden, ob sich die Modellvalidierung und Codegenerierung darin integrieren lassen. Ist das nicht der Fall, bringt das die ganze Idee der kontinuierlichen Integration in Gefahr, da nicht mehr das ganze System daran beteiligt ist.

## Fazit

Der Einsatz von Modellen und Codegenerierung bei der Entwicklung von Software kann die Effizienz des Entwicklungsteams und die Qualität des Codes erhöhen. Dabei ist es wichtig, an die eingesetzten Modellierungswerkzeuge dieselben Mindestanforderungen zu stellen, wie man sie von modernen integrierten Entwicklungsumgebungen kennt. Ansonsten werden sich die Entwickler zu Recht durch die neue Technologie in ihrer Alltagsarbeit behindert fühlen. Entscheidend ist es, die Skalierbarkeit aller Werkzeuge zu prüfen, bevor man ein größeres Projekt in Angriff nimmt. Ansonsten besteht die Gefahr, dass das Entwicklungsteam Stunden in die Erarbeitung von Modellen investiert, bloß um irgendwann festzustellen, dass die Werkzeuge ein Arbeiten praktisch unmöglich machen, weil sie nicht bis zur endgültigen Größe des Softwaresystems skalieren. ■

## Literatur & Links

**[EMF]** The Eclipse Foundation, Eclipse Modeling Framework Project, siehe: www.eclipse.org/modeling/emf/

**[Fau07]** C. Faure, F. Allilaire, The List Metamodel Refactoring example, 2007, siehe: www.eclipse.org/m2m/atl/basicExamples_Patterns/

**[Fav03]** J.M. Favre, Meta-model and model co-evolution within the 3D software space, in: Proc. of the Int. Workshop of Large-scale Industrial Software Applications, 2003

**[Mar08]** R.C. Martin, Clean Code, Prentice Hall 2008

**[Wac07]** G. Wachsmuth, Metamodel Adaptation and Model Co-adaptation, Lecture Notes in Computer Science, Volume 4609, 2007