11

Zusammenfassung und Ausblick

Das Entscheidende am Wissen ist,
dass man es beherzigt und anwendet.
Konfuzius

In Band 1 [Rum11] und diesem Band 2 wurden einige modellbasierte Konzepte und Techniken für das sich derzeit rasant weiter entwickelnde *Portfolio der Softwaretechnik* eingeführt, die basierend auf praktischen Erfahrungen und fundierten analytischen Überlegungen einen doppelten Brückenschlag vornehmen. Zum einen werden mit Hilfe der UML theoretische Erkenntnisse für die Praxis aufbereitet und so für die industrielle Softwareentwicklung besser anwendbar. Zum anderen werden Konzepte agiler Methoden auf den Einsatz der UML übertragen.

Bisher wird die UML zumeist in plangetriebenen Methoden als Grundlage für die Definition von Meilensteinen und Entwicklungsphasen eingesetzt. Mit den in diesem Buch diskutierten Konzepten gelingt es, das Wertesystem, die Prinzipien und die Entwicklungspraktiken agiler Methoden mit der UML zu kombinieren.

Als wesentliches Ergebnis wurde in Band 1 ein präzises, für viele Typen von Anwendungen gut verwendbares Sprachprofil der UML/P erstellt, das (1) auf weniger wichtige Konzepte verzichtet, (2) eine präzisierte Erklärung der Bedeutung einzelner Konstrukte bietet und (3) durch zusätzliche Konzepte auf den Einsatz als Programmier-, Modellierungs- und Testfalldefinitionssprache zugeschnitten ist.

Die wesentlichen technischen Konzepte agiler Methoden, die die Verwendung einer Sprache direkt betreffen, sind die Codegenerierung, die Möglichkeit zur Definition automatisierter Tests, die Testbarkeit des generierten Codes und die Evolution der Modelle aufgrund sich verändernder Anforderungen oder einer verbesserungswürdigen Softwarearchitektur.

Deshalb wurden diese Techniken auf die UML/P übertragen und dabei diskutiert, wie mit Hilfe der UML/P-Modelle automatisierte Tests definiert und Refactoring-Techniken auf UML/P-Modelle angewandt werden. Eine Anzahl von Testmustern, speziell für die Verbesserung der

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_11, © Springer-Verlag Berlin Heidelberg 2012

Testbarkeit objektorientierter Software sowie für funktionale Tests verteilter und nebenläufiger Systeme wurde entworfen. Mit Hilfe dieser Tests wurde ein präziser Beobachtungsbegriff festgelegt, der als Grundlage für Refactoring-Schritte dient.

Das in dieser Arbeit vorgestellte UML/P-Sprachprofil mit den darauf basierenden Techniken bildet die Basis für eine effiziente Entwicklung. Flexibilität, Effizienz und Kosten des Prozesses, Qualität und Wartbarkeit des Produkts, Time-to-Market und letztendlich die Kundenzufriedenheit werden optimiert, indem nicht nur der richtige Prozess mit den dazugehörenden Entwicklungstechniken ausgewählt wird, sondern auch, indem die genutzten Notationen auf diesen Prozess zugeschnitten sind.

Die vorgestellten Techniken zur Generierung von Code und Testfällen sowie zum transformationellen Refactoring von UML Modellen stellen eine hervorragende Grundlage für die „Model Driven Architecture“ (MDA) dar. MDA impliziert eine stark modellgetriebene Vorgehensweise, in der verschiedene Schichten von Modellen nacheinander entwickelt und idealerweise automatisch generiert werden. Die in der UML/P vorhandenen Konzepte, wie die explizite Markierung der Unvollständigkeit „...“, oder Stereotypen wie ≪match:initial≫ zur Markierung der Präzision einer durch Sequenzdiagramme gegebenen Beobachtung, erlauben es, sehr elegant Modelle auf verschiedenen Abstraktionsebenen zu entwerfen und durch Transformationen ineinander zu überführen.

UML/P und die in diesem Band beschriebenen Transformationen unterstützen deshalb MDA und erweitern es sogar deutlich. MDA fordert primär „top-down“-Transformationen von abstrakten, Plattform-unabhängigen Modellen zu detaillierten, Plattform-abhängigen Modellen und zum Code. Refactoring von Modellen ist demgegenüber eher „horizontal“ angelegt: Refactoring-Techniken verbessern die Architektur eines Systems, ohne notwendigerweise die Abstraktionsebene zu verlassen.

Das Zusammenspiel vertikaler und horizontaler Transformationen ist in diesem Buch durch die Kombination von Codegenerierung und Refactoring beschrieben und erfordert für den effizienten Einsatz eine weitestgehende Automatisierung der Generierung. Dies ist, obwohl mittlerweile Stand der Kunst, mit den heutigen Werkzeugen oft noch nicht adäquat zu realisieren und daher ein wesentliches Differenzierungsmerkmal bei der Werkzeugauswahl.

Ausblick

Mit dem in Band 1 skizzierten Vorgehensmodell und der zugrunde liegenden Notation kann nun der geneigte Leser weitere praktische Erfahrungen sammeln. Dies kann durchaus ohne ein ausgereiftes Werkzeug erfolgen. Es hat sich bereits gezeigt, dass der Einsatz der UML/P und der darauf basierenden Konzepte für Tests und Refactoring positive Effekte auf die Systemarchitektur und -qualität hat. Auch die manuelle Umsetzung von

Testmodellen verbessert das Verständnis und die Effektivität der Entwicklung und erhöht die Qualität resultierender Tests.

Aber es ist notwendig, dass Werkzeughersteller auch weiterhin intensiv daran arbeiten, über Klassendiagramme hinaus Funktionalität zur Generierung von Code und Tests anzubieten. Dabei werden allgemeine Generierungsverfahren genauso benötigt wie die Möglichkeit, bei der Generierung spezialisierte Domänen, Frameworks und Komponentenarchitekturen zu adressieren. Aufgrund des Aufwands, der für die Erstellung komfortabler graphischer Softwareentwicklungswerkzeuge notwendig ist, ist es für eine Konzepterprobung heute oft besser, die zu realisierenden Vorhaben als Erweiterungen (Plugins) zum Beispiel für ein verfügbares Open-Source-System zu erstellen. Dazu gehört beispielsweise die Frage nach Überdeckungsmetriken von Tests für UML/P, die Testfallgenerierung oder der Anschluss verifizierender Werkzeuge für Refactoring-Regeln.

Zum Zeitpunkt der Publikation der beiden Bücher ist die UML in der Version 2.3 publiziert. Einige Elemente des UML/P-Sprachprofils werden in der UML 2.3 besser unterstützt, aber im Allgemeinen ist die UML 2 mit ihren vielen Neuerungen noch nicht ganz als ausgereifter und stabiler Standard zu betrachten. Gerade deshalb ist die Eleganz, Einfachheit und Klarheit der UML/P, die sich auch durch die vergleichsweise kleine Beschreibung der abstrakten Syntax im Anhang von [Rum11] widerspiegelt, von wesentlichem Vorteil gegenüber dem Sprachstandard, der unter zu vielen politischen Einflüssen leidet.

Mit der Konsolidierung durch die UML/P ist eine Basis geschaffen, auf der eine Reihe von Erweiterungen und Untersuchungen der Praktikabilität möglich sind. Dazu gehört die empirische Untermauerung der Qualität und Effektivität der bekannten Testverfahren, für die bisher vor allem Daten basierend auf prozeduralen Sprachen existieren. Die verwendete Entwicklungssprache hat aber wesentlichen Einfluss auf die Testcharakteristika. Auch die Anzahl und Form zielführender Refactoring-Regeln und die Messung der Qualität einer Softwarestruktur bedarf empirischer Untersuchungen, die für einen UML/P-basierten Ansatz zu erheben sind.

Auf Basis der UML/P können sehr gut Domänen-spezifische Anpassungen entwickelt werden. So eignet sich die eigenschaftsorientierte Modellierungssprache OCL hervorragend zur Definition von Geschäftsregeln oder Konfiguration von Cloud-Systemen, bei deren Erfüllung jeweils bestimmte Aktionen ausgelöst werden. Komplexe Systeme wie zum Beispiel SAP/R3 bieten heute eine Vielzahl von Parametern, die die Geschäftslogik des Systems widerspiegeln. Statt also OCL und andere UML/P-Modelle direkt in Code zu übersetzen und damit die getroffenen Aussagen im System zu fixieren, kann eine Interpretation der Artefakte vom System stattfinden und damit ein dynamischer Austausch während der Laufzeit des Systems ermöglicht werden, wie dies etwa im Energie-Monitoring System [FKP+10] der Fall ist.

Eine weitere Fragestellung beschäftigt sich mit der Verwendung der UML/P zur Modellierung von Komponenten und deren Schnittstellen. Für eine Komponentenschnittstelle bietet sich die Verwendung eines aus Interfaces bestehenden Klassen- oder Objektdiagramms an. Für ein abstraktes Zustandsmodell der Komponente können Statecharts verwendet werden. OCL-Methodenspezifikationen beschreiben die Rahmenbedingungen für Methodenaufrufe an eine Komponente, und Sequenzdiagramme legen die erlaubten Interaktionsmuster fest. Für Tests kann ein Komponenten-Dummy teilweise automatisch erstellt werden. Umgekehrt kann eine Komponente durch ihre Schnittstelle von außen auf Konformität zu den zugesicherten Eigenschaften getestet werden. Der Handel mit Komponenten wird dann interessanter, weil durch automatisierte Tests das Zutrauen in die Korrektheit einer gekauften fremden Komponente deutlich erhöht werden kann.

Die Zukunft der Modellierung

Die Zukunft der Verwendung von Modellierungstechniken im Softwareentwicklungsprozess ist schwer vorherzusagen. Die Community ist sich einerseits einig, dass Modellierung noch jede Menge Potential für Verbesserungen sowohl bei den Sprachen als auch bei den zugehörigen Werkzeugen hat. Allerdings ist die Optimierung der Werkzeuglandschaft schwierig, weil Werkzeuge typischerweise nicht isoliert eingesetzt werden können, sondern integrierter Teil einer komplexeren Werkzeuglandschaft sein müssen.

Desweiteren sehen wir eine immer spezifischere Ausprägung der Entwicklungsansätze für Software in den verschiedenen Domänen. Je nach den dort vorherrschenden Komplexitätstreibern und Risiken werden sehr unterschiedliche Entwicklungsprozesse eingesetzt, die teilweise auch sehr detaillierten Normen unterliegen. Dazu kommt, dass nach der Welle der Integration von Modellierungssprachen, die im UML Sprachstandard gipfelt, nun eine Hinwendung zu Domänenspezifischen Sprachen (DSLs) stattfindet, die zwar jeweils eigenständig zu definieren sind, sich bei Ihrer Anwendung dann aber durch Kompaktheit und Problemnähe angenehm einsetzen lassen.

Es scheint, als ob in Zukunft eine grobe Klassifizierung von Projekten in die Gruppen UML-basiert, DSL-basiert und agilen, modellierungsfreien Projekte vorgenommen werden kann. Dabei können sich die drei Ansätze jedoch mischen, wenn etwa agil aus UML-Modellen generiert wird und eine DSl zur Laufzeitkonfiguration eingesetzt wird.

Für einen agilen Einsatz der UML wird jedoch noch einiges an effizienter Werkzeuginfrastruktur notwendig sein. Zum Einen fehlen weiterhin gute Werkzeuge zur Versionierung, zur Variantenbildung, zum effektiven Refactoring, Tracing von Anforderungen, komfortable Editoren, etc.

Sehr wichtig ist auch der in diesen Büchern beschriebene leichtgewichtige Einsatz von Modellen zur Generierung von Code. Dazu gehört etwa, dass der generierte Code nicht manuell verändert werden und dass er nicht

einmal gelesen werden muss. Das ist heute oft noch nötig, um etwa Coderahmen aus Klassendiagrammen mit Leben zu erfüllen oder zumindest zu verstehen, gegen welche generierten Funktionen programmiert werden kann. *Modularität* und explizite *Schnittstellen* sind für Modelle notwendig, um die gekapselten Interna verstecken zu können und so den Erfolg der Modularisierung aus Programmiersprachen auf Modelle zu heben.

Neben der Generierung sind ausgereifte Analyse- und Synthesetechniken notwendig, die neben syntaktischer Konsistenz weitreichende Eigenschaften des Systems frühzeitig erkennen lassen. Dazu gehören vor allem automatisierte Analyseverfahren, die etwa die Verletzung von Invarianten erkennen lassen und Beispiele dafür erzeugen. Dazu gehören aber auch Syntheseverfahren, die aus unvollständig definierten Sichten vollständige Modelle (intern) berechnen und diese zur Animation oder zur Codegenrierung einsetzen. So ist es heute bereits vergleichsweise gut machbar, einen OCL-Kontrakt und einen konkreten Satz von Eingabedaten in einen SAT-Solver zu geben, der eine Lösung berechnet, die als Ausgabe des spezifizierten Methodenaufrufs verstanden werden kann. Allerdings ist die automatisierte Ableitung eines Algorithmus aus einem OCL-Kontrakt heute noch nicht möglich.

Die bereits genannte Modularität der Modelle führt insbesondere bei heterogenen Sprachen wie der UML auch zu der Frage wie Sprachen modular definiert werden können. Dies ist zum Beispiel in [Völ11] diskutiert, aber noch nicht auf die UML angewendet. Eine für unabhängige Analysen und Syntheseverfahren geeignete Modularisierung der Sprache UML erscheint auch intrinsisch schwierig, weil die UML weitgehend als Monolith definiert wurde.

So ist der Einsatz einer Modellierungssprache für verschiedene Zwecke an verschiedenen Stellen erwähnt, aber es existieren in der UML noch keine Mechanismen, die solche Vewerndungsvielfalt unterstützen würden. Beispielsweise können Statecharts je nach Verwendungsform sehr unterschiedlich im Code repräsentiert sein (das ist verstanden), aber eben daher auch unterschiedliche Konsistenzbedingungen im Kontext von Klassen-, Sequenz- oder Aktivitätsdiagrammen haben (und das ist in der UML nicht adäquat reflektiert).

Auch wenn die UML Defizite hat, so ist sie doch derzeit die beste allgemein verwendete Modellierungssprache im Bereich der Softwareentwicklung. Viele erfolgreiche Projekte zeigen, dass sie ist einerseits bereits ganz gut einsetzbar ist. Andererseits zeigen aktuelle Forschungen unter anderem zu den oben genannten Themen auch, wo weitere Verbesserungen zu finden sein werden.