Jenseits des Tellerrands – Teil 10: Wie viel Abstraktion brauchen wir?

Je abstrakter, desto besser, oder?

VON JOHANNES LINK UND STEFAN ROOCK

Softwareentwicklung ist weit mehr als Programmierung, und Programmierung ist weit mehr als das bloße Beherrschen von Programmiersprachen und Werkzeugen. Die Artikel dieser Serie greifen wichtige Themen, aktuelle Fragen und auch grundsätzliche Probleme auf, beleuchten diese und stellen „die gängigen Antworten" immer wieder in Frage. Subjektivität ist dabei keineswegs verpönt, sondern das notwendige Salz in der technischen Einheitssuppe.

Es ist eine Binsenweisheit der Softwareentwicklung, dass diese nach immer höheren Stufen der Abstraktion strebt. Meist wird dieses „mehr" gleichgesetzt mit „besser", denn größere Abstraktion lässt uns ja bekanntlich das Wesentliche mit weniger Worten beschreiben. Doch auch Abstraktion kommt mit einem Preisschild daher. Welche Abstraktionen werden den großen Entwicklungsschritt in unserer Branche markieren? Haben die modellgetriebenen Ansätze das Zeug dazu, uns in ungeahnte Produktivitätssphären zu katapultieren?

Johannes Link
(john.link@gmx.de)

Heute sind Autos aus Komponenten wie Getriebe, Motor oder Katalysator zusammengebaut, die jeweils eine klar definierte Aufgabe erfüllen und eine ebenso klar definierte Schnittstelle anbieten. Wer ein Auto bauen möchte, muss die Aufgaben und Schnittstellen der Komponenten verstehen, aber nicht ihren internen Aufbau. Wer eine Komponente bauen möchte, muss ihre Aufgabe und Schnittstelle verstehen und sich Gedanken über den internen Aufbau machen. Er muss aber gar nicht oder nur sehr grob wissen, welche anderen Komponenten im Auto zum Einsatz kommen. Der interne Aufbau der anderen Komponenten muss ihn auf keinen Fall interessieren. Ohne Komponenten müsste man beim Autobau alle Details kennen; Fahrzeuge mit der heute üblichen Komplexität könnten nicht entwickelt werden. Die Komplexität würde uns überfordern und den typischen Autoentwickler träfen wir am ehesten in einer geschlossenen Anstalt.

Die großen Fortschritte in der Automobilbranche und anderen Ingenieursdisziplinen gehen wesentlich auf das Prinzip der Abstraktion zurück: Man macht sich das Leben leicht, indem man auf der Gesamt-Betrachtungsebene Details weglässt. Das kann man auch in eine Definition gießen (siehe Wikipedia, [1]):

"Abstraction is the process of reducing the information content of a concept, typically in order to retain only information which is relevant for a particular purpose. [...] Abstraction typically results in complexity reduction resulting in a simpler conceptualization of a domain in order to facilitate processing or understanding of many specific scenarios in a generic way."

Es geht also um die Reduktion der Inhaltsfülle, um das Verständnis einer Sache zu erleichtern, die wir in ihrer Gesamtheit und ihrem Detailreichtum sonst nur sehr schwer begreifen könnten. Wollen wir nur über bestimmte Konzepte reden, dann genügt meist ein recht informelles Vorgehen: Wir lassen einfach diejenigen Details weg, die wir im Zusammenhang für unwichtig halten, und betrachten den Rest, sodass manches, das sich vorher als unterschiedlich darstellte, plötzlich gleich aussieht.

Abstraktionen in der Softwareentwicklung

Bei der Softwareentwicklung geht es ganz wesentlich um Abstraktion. So sind es vor

Jenseits des Tellerrands

Was bisher geschah

- Teil 1: Das nächste Java?
- Teil 2: Typisierung in Java und darüber hinaus
- Teil 3: Warum gibt es eigentlich Softwareentwicklungsprojekte?
- Teil 4: Softwarearchitektur: eine kritische Bestandsaufnahme
- Teil 5: Textuelle domänenspezifische Sprachen
- Teil 6: Moderne System- und Abnahmetests
- Teil 7: Warum nutzen wir die Mittel zur Softwareverbesserung nicht, obwohl wir sie kennen? Eine Diskussion
- Teil 8: Java-Bashing: Unsere liebste Plattform ist zu kompliziert geworden
- Teil 9: Rich Client vs. Web Client
- **Teil 10: Wie viel Abstraktion brauchen wir?**
- Teil 11: Berufsethos, Qualität, Rollenverständnis
- Teil 12: Time's Arrow: ein Projekt von hinten

allem Abstraktionen, die der IT-Branche das exponenzielle Produktivitätswachstum der Vergangenheit beschert haben. Maschinencode wurde zu Assembler, Assembler wurde zu Cobol und Fortran. Aus diesen entwickelten sich über mehrere Schritte aktuelle Programmiersprachen wie Java oder Python. Letztere abstrahieren über Speicheradressen, Maschinenbefehle, verwendete Mikro-Prozessoren und sogar Betriebssysteme und UI-Toolkits.

Doch auch im Anwendungsbereich finden wir konkrete Gegenstände und Konzepte – wie z.B. ein Sparbuch – und bilden diese in Software ab. Dabei müssen wir zwangsläufig abstrahieren, denn das „echte", das physikalische Sparbuch können wir im Programm nicht verwenden. Später finden wir heraus, dass es verschiedene Arten von Sparbüchern gibt. Wenn es uns gelingt, eine Abstraktion über all diese Sparbuch-Arten zu finden, können wir die Entwicklungs- und Wartungsaufwände deutlich senken.

Abstraktionen manifestieren sich beim Programmieren in zahlreichen Konstrukten. So stellt bereits die Auslagerung mehrerer Programmzeilen in eine benannte Unterfunktion eine Form der Abstraktion dar. Und auch die Einführung von Parametern, Unterklassen, Interfaces etc. sind häufig Abstrahierungswerkzeuge. Hinter jeder Modularisierung steckt am Ende auch eine Abstraktion, und die Fähigkeit zur Komposition unserer Module hängt nicht selten von der Qualität der zugrunde liegenden Abstraktionen ab.

Bisher haben wir mit der Komponentenbildung beziehungsweise Modularisierung eine Art der Abstraktion beschrieben. Orthogonal zur Komponentenbildung ist die Gleichförmigkeit (Generizität), bei der man im Detail unterschiedliche Konzepte gleichförmig behandelt. Bereits sehr früh in der Geschichte der Softwareentwicklung wurde Generizität als Abstraktionskonzept verwendet, um Compiler und andere Generatoren zu entwickeln.

In der Objektorientierung ist Generizität mittels Vererbung und Polymorphie ein Grundkonzept, genauso wie Modularisierung über das Klassenkonzept. Hier zeigt sich, dass Generizität und Modularisierung keine Gegensätze sind, sondern sich gut ergänzen.

Vom Abstrakten zum Konkreten

Um eine Abstraktion für die automatisierte Verwendung in Softwaresystemen – durch Compiler, Interpreter und Generatoren – nützlich werden zu lassen, muss die Bedeutung der Abstraktion exakt sein, d.h., es muss zu einem bestimmten Zeitpunkt eine ausreichend eindeutige Abbildung des abstrakten Konzepts auf konkretere Konzepte geben. Als Standardbeispiel dafür dient häufig die Entwicklung der unterschiedlichen Generationen von Programmiersprachen: Maschinencode, Assembler und 3GLs (3GL = 3rd Generation Language wie z.B. Pascal oder Java). Dabei werden 3GLs auf eindeutige Weise in Assembler übersetzt und dieser wiederum in Maschinencode. Die Abbildungen sind nicht wirklich eindeutig, da es beispielsweise unterschiedliche Compiler für Java nach Bytecode gibt, aber ausreichend eindeutig, da (hoffentlich) alle Compiler die implizierte – und in Testsuiten spezifizierte – Semantik Javas erfüllen.

Für die Abbildung des abstrakten Konzepts auf die nächst konkretere Schicht existieren im Wesentlichen zwei Möglichkeiten: Sie kann als eigener Schritt während der Entwicklung stattfinden (z.B. durch Compiler oder Code-Generatoren) oder zur Programm-Laufzeit (z.B. durch Frameworks). Prinzipiell sind beide Umsetzungsansätze austauschbar. Generatoren führen zu schnelleren Systemen, verlangsamen durch den zusätzlichen Generierungsschritt aber die Entwicklung. Da passt es gut ins Bild, dass Generatoren eher in statisch typisierten Programmiersprachen (z.B. Cobol, C/C++) dominieren, während dieselbe Funktionalität in dynamisch typisierten Programmiersprachen (z.B. Lisp, Python oder Ruby) eher als Framework realisiert wird. In dynamischen Programmiersprachen lassen sich viele Dinge sehr einfach als Framework realisieren, die in statischen Sprachen nur mit erheblichem Aufwand umsetzbar wären. Außerdem sind dynamische Programmiersprachen systembedingt ohnehin etwas langsamer als statische, kompilierte Sprachen. Da fällt der Overhead für das Framework häufig kaum ins Gewicht. Dafür leben dynamische Programmiersprachen von kurzen Turn-around-Zeiten (Anmerkung: Die Zeit vom Tippen bis zur Ausführung des Programms), sodass ein zusätzlicher Generierungsschritt nicht so recht ins Konzept passen will.

Darüber hinaus bieten die meisten dynamischen Programmiersprachen von Haus aus Sprachmittel für Closures (Anmerkung: Programmblöcke, die wie Daten behandelt und an Variablen gebunden werden können), sodass sich zahlreiche Abstraktion innerhalb der Programmiersprache leichter als außerhalb formulieren lassen.

Java befindet sich in diesem Spannungsfeld irgendwo in der Mitte. Java benötigt nur einen Kompilier-, aber keinen statischen Link-Vorgang, und im Zusammenhang mit inkrementellen Compilern (wie sie etwa Eclipse und IntelliJ mitbringen) wird ein Programm sofort kompiliert, sobald man seine Änderungen am Quellcode speichert. In Bezug auf Turn-around-Zeiten kommt man den dynamischen Programmiersprachen damit schon sehr nahe.

Auch kennt Java mit anonymen inneren Klassen eine Art Closure-Mechanismus, wenn auch mit aufwendiger und abschreckender Notation. Wie eine clevere Closure-Notation in Java aussehen könnte, zeigt Groovy. Dieser und einige andere Mängel der Sprache führen dazu, dass im Java-Umfeld zunehmend externe Darstellungen für bestimmte Problembereiche geschaffen und diese häufig durch Generatoren umgesetzt werden.

Abstraktion in der Programmierung

Abstraktion bietet also große Potenziale, hat aber auch Pferdefüße:

1. Abstraktes ist schwieriger zu verstehen als Konkretes.
2. Stabile Abstraktionen lassen sich häufig nur mit viel Erfahrung und hohem Aufwand finden.
3. Erweist sich eine Abstraktion als falsch oder instabil, können die Änderungskosten sehr hoch werden.
4. Unvollständige Abstraktionen können häufige Kontextwechsel erfordern und so ihren Nutzen reduzieren.

5. Generizität beschränkt den Entwicklungsprozess.

Verständlichkeit von Abstraktionen

Abstraktes ist schwieriger zu verstehen als Konkretes. Zumindest neue Entwickler benötigen einen längeren Einarbeitungsprozess. Der Nutzen der Abstraktion muss diese zusätzlichen Kosten übersteigen.

Aufwand stabiler Abstraktionen

Stabile Abstraktionen findet man in etablierten Anwendungsbereichen häufig bereits vor. Was die Abstraktion *Sparbuch* bedeutet, ist im Bankenkontext seit langem geklärt. Viel schwieriger wird es, wenn wir Abstraktionen in unserem eigenen, noch jungen Anwendungsbereich Softwareentwicklung suchen. Deshalb scheitern so viele Entwicklungsteams daran, sich erfolgreiche Frameworks zur eigenen Arbeitserleichterung zu schaffen.

Instabile Abstraktionen

Abstraktionen sind zu Beginn immer instabil. Sie müssen sich erst bewähren und werden während dieser Bewährungszeit immer wieder angepasst. Leider kann man nur sehr schlecht vorhersagen, wie lange dieser Reifungsprozess dauern wird. Neu entwickelte Abstraktionen sind daher der Albtraum jedes Projektleiters: Es gibt zwar die Hoffnung, durch die Abstraktion das Projekt schnell und kostengünstig voranzubringen; leider gibt es aber keine sinnvolle Grundlage für Aufwandsschätzung und Zeitplanung. Aus diesem Grund geben sich viele Teams bereits nach kurzer Zeit mit mangelhaften Abstraktionen zufrieden.

Kontextwechsel durch unvollständige Abstraktionen

Insbesondere bei generischen Lösungen ist die Vollständigkeit der Abstraktion relevant, aber häufig nur schwer herzustellen. Betrachtet man die typische Geschäftsanwendung, verbringt diese einen Großteil ihrer Lebenszeit mit Daten-Schubsen: Anwender geben Daten ein, die gespeichert und später wieder angezeigt werden können. Da sollte man sich doch das Leben als Softwareentwicklung deutlich vereinfachen können! Ein einfacher Generator für die Definition von Datenstrukturen ist schnell programmiert, doch unglücklicherweise kommt man damit nicht sehr weit. Schnell stellt unser Entwickler fest, dass er Wiederholungen in der Definition seiner Datenstrukturen benötigt (ein Auftrag hat mehrere Positionen). Und auch Beziehungen zwischen Datenstrukturen sind notwendig (der Auftrag braucht eine Referenz auf den Kunden). Aber damit noch immer nicht genug: Natürlich braucht man auch ein paar Berechnungen (Positionspreis, Einzelpreis, Mehrwertsteuer) und Plausibilitätsprüfungen. Und wenn der Entwickler dann noch nicht aufgegeben hat, dann sieht er sich mit Berechtigungen und Internationalisierung konfrontiert. Und schon ist der anfänglich einfache Formalismus zum Beschreiben von Datenstrukturen zu einer fast vollständigen Programmiersprache geworden. Vom Aufwand dafür mal abgesehen: Wo liegt der Vor-

Anzeige

teil? Haben wir wirklich eine relevante Abstraktion geschaffen?

Generizität beschränkt den Entwicklungsprozess

Wird ein generischer Mechanismus in einem Projekt entwickelt, hat dies eine sehr unangenehme Auswirkung auf den Entwicklungsprozess. Solange die Abstraktion nicht stabil ist, muss für viele Anforderungen die Abstraktion immer wieder angepasst werden: Sie entwickelt sich evolutionär. Das ist gut. Weniger gut ist, dass man die Arbeit an der Abstraktion nicht beliebig parallelisieren kann. So verhindern noch unreife generische Mechanismen, dass man durch zusätzliche Entwickler die Programmierung beschleunigt. Denn gerade bei gut erkennbaren Duplikationen, wo sich generische Mechanismen aufdrängen, ließe sich die Arbeit ohne generischen Mechanismus sehr gut parallelisieren.

Ergo: Die meisten Abstraktionen scheitern

Eigene Abstraktionen zu schaffen ist eine sehr schwierige Aufgabe. Die meisten Versuche in dieser Richtung liefern daher nur mäßige Ergebnisse oder scheitern komplett.

Gegenspieler und problematische Abstraktionen

Eine Art Gegenbewegung zur Abstraktion stellen Explizitheit und Konkretisierung dar. Je expliziter und/oder konkreter ein Vorgang beschrieben ist, desto mehr kann man über die Details herauslesen [3]. Das hat Vorteile und ist manchmal sogar wesentlich, um Code überhaupt ändern zu können. Stellen wir uns vor, dass die Berechnung des Dispokredits eines Kunden fehlerhaft ist und wir mit der Beseitigung des Defekts beauftragt wurden. Nur wenn wir die Möglichkeit haben, alle Details der Berechnung in ihrer tatsächlichen Reihenfolge zu sehen, werden wir in der Lage sein, dem Programmfehler auf die Spur zu kommen. Existiert für uns jedoch keine Möglichkeit, hinter die Abstraktion „Dispokredit“ zu schauen, dann stecken wir bei der Fehlersuche fest. Trotz oder gerade wegen der vorhandenen Abstraktion.

Um das obige Problem zu vermeiden, wünschen wir uns vollständige Abstraktionen. Eine Abstraktion nennen wir dann „vollständig“, wenn wir sie (fast) nie verlassen müssen: nicht bei Änderungen und Refactorings des Codes, nicht beim Debuggen und auch nicht beim Testen unserer Software. Eine vollständige Abstraktion erlaubt es uns, kein Wissen über tiefere Schichten zu sammeln. Eine quasi vollständige Abstraktion bietet z.B. die JVM. In den allerwenigsten Fällen müssen Entwickler wirklich wissen, welcher Bytecode entsteht oder gar, wie dieser Bytecode auf dem Prozessor ausgeführt wird.

Fachliche – und auch die meisten technischen – Abstraktionen sind jedoch immer irgendwo „leaky“ [2] oder „löchrig“, d.h., die darunter liegende konkretere Ebene scheint immer mal wieder durch und muss von den Verwendern der Abstraktion verstanden werden. Und genau hier ist es nun plötzlich wesentlich, wie einfach wir – als Programmierer oder Tester – auf diese konkretere Ebene wechseln können:

- Haben wir die Abstraktion lediglich als Unterfunktion oder Delegationsobjekt vor uns, dann wechseln wir die Abstraktionsebene durch einen Sprung im Quellcode zur Klasse *Dispo*.
- Liegt die Konkretisierung des abstrakten Konzepts in einer Fremdbibliothek, so müssen wir uns eventuell bereits mit dekompiliertem – also deutlich schlechter lesbarem – Code herumschlagen.
- Haben wir unser abstraktes Konzept mithilfe von AOP (aspektorientierter Programmierung) umgesetzt, müssen wir möglicherweise lange suchen, um die Laufzeitsemantik eines Funktionsaufrufs herauszufinden.
- Wurde unser Code gar aus Modellen generiert, erfordert die Fehlersuche sogar häufig einen Technologiewechsel, beispielsweise von UML hin zum generierten Quellcode.

Das Versprechen der Abstraktion ist es, die Modellierung und Programmierung von Systemen einfacher zu machen, indem sie unwichtige Dinge verbirgt und nur noch die wesentlichen Aspekte eines Konzepts sichtbar macht. Um mich als Entwickler wirklich auf eine Abstraktion einlassen zu können, muss diese (für die meisten Fälle) vollständig sein. Unvollständige Abstraktionen verlangen den häufigen Wechsel der Abstraktionsebene, d.h., das Detailwissen über mehrere Abstraktionsebenen muss beim Entwickler vorhanden sein. Dies ist dann weniger ein Problem, wenn wir uns zwar in unterschiedlichen Abstraktionsebenen, aber der gleichen Technologieebene bewegen. Das ist bei Komponentenbildung der Fall, nicht aber bei Generatoren. Dynamische Programmiersprachen haben mit ihren eleganten Ausdrucksmöglichkeiten für Closures hier die Nase vorn. Mit ihnen lassen sich viele Abstraktionen innerhalb der gewählten Programmiersprache elegant ausdrücken, die in statischeren Programmiersprachen einen eigenen Formalismus und damit einen Generator erfordern.

Modellgetrieben

Bei modellgetriebener Entwicklung entsteht also – bedingt durch die Codegenerierung aus Modellen – immer ein Technologiebruch. Für die Arbeit auf Modellebene bedeutet dies, dass die Ermittlung der semantischen Details schwieriger und aufwendiger wird, während im Quellcode selbst die Abstraktion verloren gegangen ist. Zudem ist die Entwicklung von Werkzeugen, die auch Debugging, Refactoring und Testen in der Sprache der Modelle erlauben, mit den heutigen Werkzeugen so aufwendig, dass sie sich nur für die wenigsten Anwendungsbereiche lohnt. Der Nutzen modellgetriebener Entwicklung liegt damit überwiegend in der Vermeidung von dupliziertem Code und Logik. An die beteiligten Entwickler jedoch werden höhere Anforderungen gestellt, es gilt also keineswegs „Unsere Entwickler müssen von der darunter liegenden Technologie nichts verstehen.“ Das Gegenteil ist häufig der Fall.

Auch der Einsatz domänenspezifischer Sprachen (DSL) gerät durch diese Überlegungen in ein Dilemma. Da Umfang und konkrete Semantik einer DSL während der Entwicklung oft starken Änderungen unterworfen sind, kann der Domänenexperte zwar Anforderungen und Geschäftsregeln in Form einer DSL schreiben, aber Testen und Debuggen müssen es die Entwickler! Um die Feed-

back-Schleife zwischen Design und Verwendung einer DSL möglichst klein zu halten, drängt sich daher der Einsatz interner DSLs auf, also solcher, die in der Programmiersprache selbst geschrieben sind. Doch diese sind wiederum für den Kunden und Domänenexperten schwerer verständlich, vor allem in Java.

Modellgetriebene Entwicklung in der einen oder anderen Form ist zurzeit ein Hype-Thema. Das sollte aber nicht darüber hinwegtäuschen, dass mit der modellgetriebenen Entwicklung nur eine Art der Abstraktion umgesetzt wird: automatisierte Gleichförmigkeit (Generator-Ansatz). Daneben stehen uns nach wie vor Komponentenbildung und manuelle Gleichförmigkeit (z.B. in Form von Entwurfsmustern oder Musterarchitekturen) zur Verfügung, die in vielen Fällen leichter handhabbar sind.

Was bringt die Zukunft?

Allgemeine projektübergreifende Abstraktionen sind in den letzten 30 Jahren kaum entstanden: Die Konzepte der Objektorientierung fanden sich bereits vorher in Simula, und die Flexibilität aktueller dynamischer Programmiersprachen reicht immer noch nicht an LISP heran [4]. Viele der Fortschritte der Vergangenheit hängen tatsächlich eng damit zusammen, dass die Mainstream-Programmiersprachen dynamischer geworden sind. Möglicherweise kommt weiterer Produktivitätszuwachs ganz von alleine, wenn die Hardware immer schneller wird und damit den Einsatz dynamischere Programmiersprachen in immer mehr Anwendungsbereichen erlaubt.

Ein echter „Abstraktionssprung“ durch Generierung (modellgetriebene Ansätze) ist eben keine offensichtliche und triviale Angelegenheit, sondern wird vor allem durch die schnelle Veränderung der domänenspezifischen Abstraktionen selbst verhindert. Technologie kann uns da ein bisschen zur Hilfe kommen, aber Wunder werden auch MDA & Co. nicht vollbringen.

Nach drei Jahren in der akademischen und industriellen IT-Forschung arbeitet **Johannes Link** seit 1999 als Softwareentwickler, Projektleiter, Entwicklungscoach und Qualitätskämpfer. Er ist Autor der Bücher „Softwaretests mit JUnit“ und „Unit Testing in Java“. Auf zahlreiches Feedback zur Serie und zu diesem Artikel wartet er gespannt unter john.link@gmx.net.

Dipl.-Inform. Stefan Roock ist Senior-IT-Berater bei it-agile in Hamburg. Er verfügt über mehrjährige Erfahrung aus agilen Softwareprojekten (vor allem: eXtreme Programming) als Projektleiter, Entwickler, Kundenberater, Entwicklerberater und XP-Coach. Darüber hinaus hat er zahlreiche Artikel und Tagungsbeiträge über agile Softwareentwicklung verfasst und ist Autor der Bücher „Software entwickeln mit eXtreme Programming“, „Refactorings in großen Softwareprojekten“ und „Hibernate. Persistenz in Java-Systemen mit Hibernate 3“.

Links & Literatur

[1] en.wikipedia.org/wiki/Abstraction

[2] Joel Spolsky: The Law of Leaky Abstractions: www.joelonsoftware.com/articles/0LeakyAbstractions.html

[3] Martin Fowler: To Be Explicit. IEEE Software, November/December 2001

[4] Paul Graham: Revenge of the Nerds, 2001: www.paulgraham.com/icad.html

Anzeige