von kerstin dittert

mehr zum thema:
c2.com/cgi/wiki?AntiPatterns

SOFTWAREARCHITEKTUR: MYTHEN UND LEGENDEN

Was zeichnet eine gute Softwarearchitektur aus? Welche Werkzeuge setze ich für den Entwurf ein, welche Entwurfsmuster garantieren mir Stabilität und Flexibilität? Wie gehe ich möglichst effektiv mit geänderten Anforderungen um? Mit diesen und verwandten Fragen beschäftigen sich viele Bücher und Artikel. So genannte „Best-Practices" unterstützen den Softwarearchitekten bei der Beherrschung seines komplexen Aufgabengebietes. Gleichzeitig kursieren aber auch Mythen und Legenden bezüglich der optimalen Vorgehensweise in der Welt der Softwareprojekte. Der Artikel stellt einige dieser Mythen vor und empfiehlt diese im Kontext eines konkreten Projekts kritisch zu hinterfragen.

die autorin

Kerstin Dittert (E-Mail: kerstin.dittert@oocon.de) ist freie Beraterin mit den Schwerpunkten Softwarearchitektur, OO-Methodik, J2EE und Projektmanagement. Sie unterstützt Unternehmen in allen Gebieten des objektorientierten Softwareentwicklungsprozesses sowie in Technologiefragen.

Moderne Legenden

Anfang 1990 veröffentlichte der Göttinger Volkskundler Prof. Dr. Rolf Wilhelm Brednich ein populäres Buch mit dem Namen „Die Spinne in der Yucca-Palme" (siehe [Bre90]). Brednich stellte darin moderne Mythen und Wandersagen vor, allesamt Horrorgeschichten, die unter der Hand kursierten, nach dem Motto: „Du glaubst nicht, was dem Freund eines guten Freundes passiert ist ...". Das Buch wurde schnell zum Bestseller und landete schließlich auch in meinem Bücherregal. Ich fing an zu lesen und amüsierte mich, wer auf solche Geschichten hereinfiel. Sehr bald entdeckte ich jedoch die erste Geschichte, die auch mir anvertraut worden war. Natürlich war es der Freundin eines Mitbewohners eines Freundes passiert. Mit wohligem Schauern hatte ich die Geschichte damals tatsächlich für wahr gehalten!

Was hat das alles mit Softwarearchitektur zu tun? Auch in diesem Gebiet existieren zahlreiche Legenden, die von Projekt zu Projekt weitergegeben werden. Wie jede ordentliche moderne Legende enthalten sie natürlich auch einen Funken Wahrheit, allerdings nur im Rahmen eines passenden Kontextes. Über diesen Gültigkeitsbereich hinaus sind ihre Empfehlungen jedoch eher schädlich als nützlich. Als Kommunikationsmittel dient häufig die „Stille Post"[1]). Hierbei werden Tatsachen hinzugefügt oder auch einfach weggelassen. Dies kann fatale Folgen haben: Beispielsweise wird ein Lösungsansatz, der für mobile Endgeräte sinnvoll war, im Großrechner-Umfeld nur selten adäquat sein.

Im Folgenden stelle ich einige dieser Legenden vor, die sich hartnäckig halten und die mir im Projektalltag immer wieder begegnet sind. Sollten Sie mit einer dieser „Wahrheiten" konfrontiert werden, so empfehle ich Ihnen, diese kritisch in Ihrem konkreten Projektumfeld zu beleuchten. Sobald Sie feststellen, dass diese Lösungsansätze für *Ihr* Projekt nicht passen, gehen Sie agil vor und werfen Sie diese Legenden über Bord.

Einige dieser Mythen und Legenden sind eng verwandt mit den so genannten „Anti-Patterns" (siehe Kasten 1). Auf derartige Bezüge werde ich später noch eingehen.

Mythos 1: Frei konfigurierbare Schnittstellen garantieren höchste Flexibilität

Im Zeitalter von Objektorientierung, Komponenten-Architekturen und verteilten Anwendungen scheint die Forderung nach festen Schnittstellen ein Anachronismus zu sein. In der Tat weisen frei konfigurierbare Schnittstellen einige *Vorteile* auf:

- Sie sind leicht erweiterbar.
- Sie lassen sich in vielen Fällen zur Laufzeit ändern.
- Das erneute Kompilieren (und gegebenenfalls Verteilen der Software) *aufgrund der Schnittstellenänderung* entfällt.

„Leichte Änderbarkeit? Genau das benötigen wir doch, um agil auf sich ändernde Anforderungen reagieren zu können! Deshalb dürfen wir ausschließlich frei

Anti-Pattern

Ein Anti-Pattern ist ein Entwurfsmuster, das beschreibt, wie man von einem Problem zu einer *schlechten Lösung* gelangt.
Ein gut beschriebenes „AntiPattern" erklärt

- weshalb die schlechte Lösung zunächst attraktiv erscheint (zum Beispiel weil sie in einem sehr begrenzten Kontext funktioniert),
- wieso die Auswirkungen so negativ sind,
- welche positiven Muster statt dessen eingesetzt werden können.

Quelle: [WikiAntPat]

Kasten 1

[1]) *Das „globale Dorf" Internet eignet sich natürlich besonders gut für „stille Post".*

konfigurierbare Schnittstellen entwerfen. Anderenfalls brauchen wir später zu lange, um neue Anforderungen zu integrieren." Dieses Argument fällt häufig in Designdiskussionen. Gegner dieser Sichtweise werden nicht selten der „Steinzeit-Objektorientierung" verdächtigt.

Frei konfigurierbare Schnittstellen sind jedoch komplexer als fest definierte Schnittstellen. In der Regel übernehmen Laufzeit-Parameter die Rollen von Methodennamen. Hierdurch ergeben sich einige gravierende *Nachteile*:

- Frei konfigurierbare Schnittstellen sind schwieriger zu verstehen und führen zu längeren Einarbeitungszeiten bei neuen Teammitarbeitern.
- Methodendeklarationen sind nicht mehr „sprechend". Zulässige Parameter und ihre Bedeutung müssen sorgfältig dokumentiert werden.
- Fehler werden nicht mehr beim Kompilieren entdeckt, sondern zur Laufzeit. Ein wesentlicher Vorteil kompilierter Sprachen bleibt ungenutzt.
- Fehler werden erst spät im Entwicklungsprozess entdeckt – bei der Integration einzelner Komponenten. Durch die Vielzahl kombinatorischer Möglichkeiten steigt der Aufwand für die Fehlerbereinigung.
- Die Schnittstellenmethoden arbeiten häufig fast nur noch string-basiert. Die mangelnde Typbindung geht mit Performanzverlusten (durch *Casting*, Instanziierung per *Reflection-API* etc.) einher. Darüber hinaus werden wiederum Fehleridentifikationsmöglichkeiten während der Kompilierung verschenkt.
- In extremen (fast paradoxen) Fällen können kryptische Schnittstellen sogar dazu führen, dass überhaupt nichts mehr geändert wird, da niemand die Seiteneffekte im Griff hat.
- *Teile* des gesamten Softwarepaketes müssen in der Regel *immer* geändert werden. Wäre dies nicht der Fall, wer sollte sonst die neuen Schnittstellen-Informationen verarbeiten? In der Regel muss mindestens eine Komponente des Gesamtsystems neu kompiliert werden. Dadurch wird am Ende doch ein neues Anwendungs-Release erzeugt, das häufig auch neu verteilt werden muss.

Dies alles führt dazu, dass fest definierte Schnittstellen in den meisten Fällen weitaus wartungsfreundlicher und leichter änderbar sind als frei konfigurierbare Schnittstellen (siehe auch Abb. 1).

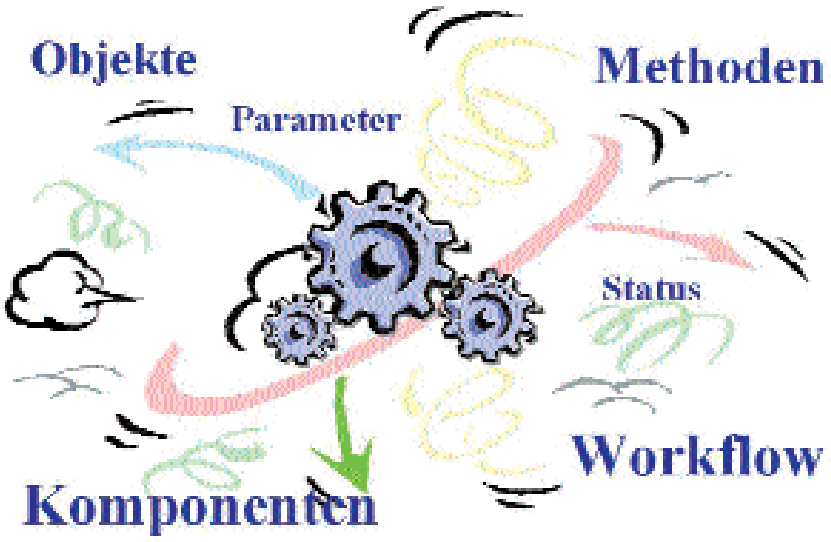

Abb. 1: Chaos durch übertriebene freie Konfigurierbarkeit

Eine Ausnahme bilden jedoch Systeme, bei denen das so genannte „Hot-Deployment" (d. h. Austausch einer Komponente zur Laufzeit ohne Neustart der Anwendung) eine zwingende Anforderung ist. In diesen Fällen sind frei konfigurierbare Schnittstellen fast immer unumgänglich. Scheuen Sie sich jedoch nicht, auch diese Anforderung kritisch zu hinterfragen. Nicht selten wird sie eher von einem technischen Hype getrieben, als dass ein wirtschaftlicher Mehrwert damit erreicht werden kann.

Gerade bei einer agilen Vorgehensweise sind feste Schnittstellen meist die bessere Wahl, da Änderungen einfacher und mit weniger Risiken durchgeführt werden können. Gemäss dem *YAGNI*-Prinzip (siehe Kasten 2) sollten Sie frei konfigurierbare Schnittstellen erst dann einführen, wenn Sie hierdurch einen Mehrwert erzielen können. Kurz und knapp gesagt: „Nach allen Seiten offen ist nicht ganz dicht"[2]).

Wollen Sie das Thema der unerschöpflichen Flexibilität noch weiter vertiefen, so schauen Sie bei Wiki unter dem *Anti-Pattern* des universell einsetzbaren „Schweizer Offiziermesser" nach (vgl. [WikiSwArm]).

Mythos 2: Eigenentwicklung macht (hersteller-)unabhängig

Wer Software entwickelt, möchte möglichst unabhängig von den Herstellern eingesetzter Drittkomponenten sein. Das Ziel ist berechtigt, da jeder Release-Wechsel

YAGNI

Abk. aus dem Englischen:
„You aren't gonna need it."

Prinzip des **Extreme Programming,** welches besagt:

„Implementieren Sie Dinge erst zu dem Zeitpunkt, wenn Sie sie *tatsächlich benötigen*, aber niemals, wenn Sie nur *vorhersehen*, dass Sie diese Dinge *später einmal* brauchen werden.

Quelle: [WikiYAGNI]

Kasten 2

[2]) *Herzlichen Dank an meine Kollegin Petra Bremer, die jene äußerste treffende Bemerkung machte, als unser Projektteam sich in einem Dschungel generischer Schnittstellen verirrt hatte.*

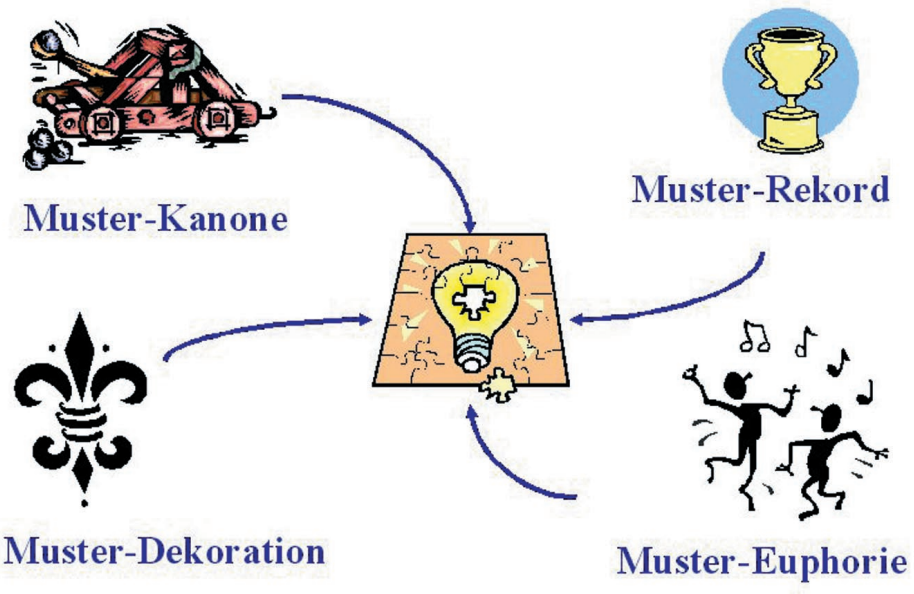

Abb. 2: Fehlentwicklungen beim Einsatz von Entwurfsmustern

fremder Komponenten das Risiko erheblicher Integrationsaufwände und -kosten birgt. Um eine hohe Investitionssicherheit zu erlangen, sollten Sie deshalb möglichst auf Produkte und Komponenten setzen, die offenen Standards genügen. Dies sichert in der Regel einen entsprechenden Wettbewerb und die Verfügbarkeit von alternativen Produkten.

Eine absolute Herstellerunabhängigkeit können Sie jedoch auch mit diesem Vorgehen nicht erzielen. Jeder Produzent von Drittkomponenten bewegt sich im Spannungsfeld zwischen maximaler Kundenbindung und Offenheit durch die Unterstützung allgemein anerkannter Standards. Beinhaltet eine Komponente Zusatz-Features, die den Standard sinnvoll erweitern, so kann dies einerseits den Kundennutzen erhöhen und andererseits die Austauschbarkeit erheblich erschweren.

Die Frage des Einsatzes fremder Produkte und Komponenten beschäftigt Softwarearchitekten und das Entwicklungsteam bereits in einem frühen Projektstadium. Häufig plädiert mindestens einer der Beteiligten vehement für eine Eigenentwicklung der in Frage kommenden Komponenten. Besonders beliebt scheint beispielsweise die Neuentwicklung von Persistenz-Frameworks zu sein – eine Aufgabe, an der sich schon zahlreiche Entwicklungsteams verschlissen haben. Die Fraktion der „Neuentwickler" bringt meist einige der folgenden Argumente ein:

- „Die Produkte ABC und YXZ sind zu teuer und entsprechen nicht hundertprozentig unseren Anforderungen."
- „Wir benötigen nur einen geringen Teil des Leistungsumfangs der angebotenen Produkte. Den Rest bezahlen wir völlig umsonst mit."
- „Produkt ABC genügt unseren Anforderungen zwar optimal, aber dessen Hersteller ist zu neu am Markt, wir haben keine Investitionssicherheit."
- „Deshalb sollten wir die Komponente selbst entwickeln. Die von uns wirklich benötigten Anforderungen haben wir relativ schnell und zu einem Bruchteil der Produktkosten umgesetzt."

Dahinter steckt auf Seiten der Entwickler häufig der nachvollziehbare Wunsch, eine spannende und anspruchsvolle Aufgabe elegant zu lösen. Treffen diese Entwickler auf Architekten, die den Entwicklungs-, Test- und späteren Wartungsaufwand schwerlich einschätzen können, so steht der munteren Neuentwicklung nichts mehr im Wege (siehe auch *Anti-Patterns* „Das Rad neu erfinden" und „Architekten sind keine Entwickler", [WikiRInWh] und [WikiArCod]).

In den meisten Fällen ist ein derartiges Vorgehen wesentlich teuerer als die Verwendung von Drittkomponenten. Ein erheblicher Zeit- und Kostendruck wirkt auf das Projekt ein. Besonders der Zeitfaktor führt nicht selten dazu, dass die Eigenentwicklung schließlich eingestellt wird und in letzter Sekunde doch noch eine Drittkomponente integriert werden muss, um den Projekterfolg nicht zu gefährden.

Auch auf lange Sicht rechnet sich eine Eigenentwicklung meistens nicht. Befürworter einer Neuentwicklung stellen den einmaligen Entwicklungskosten gern die folgenden Kosten für fremde Komponenten gegenüber:

- Produktkosten,
- Support-Kosten,
- erwartete Kosten für die Integration zukünftiger Releases sowie
- erwartete Kosten für die Migration zu einem anderen Produkt.

Hierbei wird oft vergessen, das auch die eigene Software gewartet werden muss und dort ebenfalls Anpassungen an neue Standards erforderlich werden können. Und wer kann schon realistisch vorhersagen, ob ein Hersteller- oder Produktwechsel mittelfristig erforderlich sein wird? In gewisser Weise begegnen wir hier wieder dem YAGNI-Prinzip im neuen Gewand (vgl. Kasten 2). Der Versuch, durch eine Eigenentwicklung alle Eventualitäten abzusichern, wird teuer bezahlt. Er macht Projekte so unflexibel, dass Sie manchmal nur noch auf der Stelle treten und die Zukunft nie erreichen.

Mythos 3: Der Einsatz von Entwurfsmustern führt stets zu besserer Qualität

Ursprung der Softwarearchitekturmuster waren die Arbeiten von Christoph Alexander, der Architekturmuster für den

Bau von Gebäuden entwickelt hatte: „Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, sodass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen" (vgl. [Ale99]).

Mitte der neunziger Jahre wurden diese Ideen auf die Architektur von Softwareprojekten übertragen, unter anderem von der „Gang of Four" (GoF), bestehend aus Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. 1995 erschien ihr Buch „Design Patterns" (vgl. [Gam95]), das den Beginn einer neuen Ära in der objektorientierten Softwareentwicklung einleitete.

Entwurfsmuster beschreiben die Essenz langjähriger Programmiererfahrung. Sie bewahren das Wissen über erprobte Lösungen und erleichtern aufgrund ihres standardisierten Aufbaus die Kommunikation im Softwareentwicklungsteam. Der Bezug auf ein gängiges Entwurfsmuster ersetzt lange und umständliche Beschreibungen.

Objektorientierte Softwareentwicklungs- und Entwurfswerkzeuge können sich heute ohne weitreichende Unterstützung für Entwurfsmuster kaum noch am Markt behaupten. Das Generieren einer „Fabrik" oder „Fassade" geschieht mit den entsprechenden Werkzeugen auf Knopfdruck. Klingt doch bestens, oder?

Natürlich sind Entwurfsmuster ein hervorragendes Hilfsmittel für den (Software-) Architekten, sonst hätten sie sich nicht so schnell durchgesetzt. Der Erfolg eines Softwareentwicklungsprojekts steht und fällt jedoch nach wie vor mit dem *Team*. Stellen Sie Ihr „Dream-Team" aus fähigen Architekten, begeisterten Entwicklern und kreativen Designern zusammen und es wird mit hoher Wahrscheinlichkeit eine gute Anwendung erstellen. Verwendet dieses Team Entwurfsmuster, so wird es noch produktiver werden und die Qualität der Anwendung wird sich noch weiter verbessern. Ihr Team wird das Werkzeug Entwurfsmuster effektiv einsetzen.

Weniger erfahrene Entwickler, Architekten oder Designer schießen beim Einsatz vom Entwurfsmustern jedoch häufig übers Ziel hinaus (siehe Abb. 2). Typische Fehlentwicklungen sind:

- **Muster-Kanone:** „Wir hätten die zwei (!!!) Eingabemasken zwar statisch miteinander verknüpfen können. Dies erschien uns aber nicht zukunftssicher. Wir haben deshalb das Verteiler-Enwurfsmuster eingesetzt, auf Basis von XML und per XSL transformiert. Damit konnten wir sogar noch deutliche Verbesserungen gegenüber dem Standard-Verteiler-Entwurfsmuster erzielen, sodass die Zustandsübergänge zur Laufzeit jetzt frei konfigurierbar sind. Leider haben wir es jedoch nicht mehr geschafft das Persistenz-Framework einzubinden und zu evaluieren" (Szenario aus der Entwicklung eines Prototypen).
- **Muster-Euphorie:** „Zu Beginn haben wir das Kommando-Entwurfsmuster nur für die Kommunikation zwischen zwei Komponenten eingesetzt. Das hat so gut funktioniert, dass wir auch innerhalb einer Komponente jetzt nicht mehr mit festen Methoden arbeiten. Wir setzen jetzt überall das Kommando-Entwurfsmuster ein und arbeiten durchgängig mit generischen Methoden." (Szenario aus einem evolutionären Prototyp-Entwicklung. Die Entwicklung kam mehr oder weniger zum Stillstand, da der Programmcode zum Schluss nicht mehr lesbar war und Seiteneffekte sich mehr und mehr ausbreiteten. Die Änderung eines Parameters konnte ungeahnte Auswirkungen haben.)
- **Muster-Dekoration:** „Wir haben das bestehende Design noch verbessert. Da das neue Release unserer ABC-Entwicklungsumgebung eine hervorragende Entwurfsmuster-Unterstützung besitzt, konnten wir ohne großen Aufwand noch zusätzliche Entwurfsmuster hinzufügen." (Gerade hierin liegt die Gefahr der Entwurfsmuster-Unterstützung von modernen Entwicklungswerkzeugen.)
- **Muster-Rekord:** „Unsere Anwendung hat eine sehr hohe Qualität. Wir haben *alle* Entwurfsmuster des GoF-Buches eingebaut" (kein Kommentar).

All diese Beispiele zeigen, dass der falsch verstandene Einsatz von Entwurfsmustern gravierende Folgen haben kann. Einfache, dem Problem und dessen Kontext angemessene Entwürfe können bis zur Unkenntlichkeit aufgebläht werden. Die Produktivität sinkt, die Software wird fehleranfällig und schwer wartbar.

Ähnliche Phänomene werden im Anti-Pattern „Kompatible austauschbare Entwickler" beschrieben (vgl. [WikiPCIEn]). Beim Kochen kommt es eben nicht nur auf die Zutaten an, sondern auch auf die Köche.

Mythos 4: Diese Anforderungen können nicht (weiter) priorisiert werden

Die fachlichen Anforderungen sind aufgenommen, Anwendungsfallbeschreibungen und zugehörige Diagramme existieren. Darüber hinaus wurde bereits eine Vielzahl nicht-funktionaler Anforderungen an die Architektur aufgenommen. Das Team stellt die ersten Architekturentwürfe zusammen, alles läuft bestens, die Stimmung ist gut. Aber auf einmal gerät der Fortschritt ins Stocken, die Teilentwürfe passen nicht zusammen. Ein Auszug aus den Systemanforderungen könnte ungefähr so klingen:

(1) „Die Anwendung muss hoch verfügbar sein. Der Austausch geänderter Komponenten muss deshalb zur Laufzeit erfolgen."
(2) „Jede Benutzeraktivität muss protokolliert werden."
(3) „Die Anwender fordern zu Recht kurze Antwortzeiten. Eine Suchabfrage muss in weniger als einer Sekunde ausgeführt werden."
(4) „Es muss gewährleistet sein, dass die aktuellen Daten jederzeit im Zugriff sind. Pessimistisches Locking ist deshalb unverzichtbar."
(5) „Wir arbeiten mit sensiblen Daten. Die Berechtigungsprüfungen müssen bis auf Feldebene gehen."
(6) „Unser System hat eine langjährige Nutzungsdauer. Wir wissen nicht, was kommt. Aus diesem Grund dürfen im System nur frei konfigurierbare Schnittstellen verwendet werden."
(7) „Einige Anwender arbeiten im Offline-Modus. Sie müssen eine lokale Replik der Datenbank erhalten, die dann später wieder synchronisiert wird."

Wollen Sie alle diese Anforderungen gleich gut berücksichtigen, so stehen Sie vor einer unlösbaren Aufgabe. Beispielsweise stehen die Anforderungen (2) und (6) als „Performance-Killer" bereits in krassem Widerspruch zum Wunsch nach kurzen Antwortzeiten (3). Und wie passen die Anforderungen (4) und (7) – pessimistischen Locking einerseits, und die Arbeit auf lokalen Repliken mit anschließender Synchronisation andererseits – zusammen?

Die oben geschilderten Anforderungen sind keineswegs besonders exotisch; statt dessen begegnet man der einen oder anderen Teilmenge in fast jedem Softwareentwicklungsprojekt. Selbstverständlich ▸

Abb. 3: Beispiel für schlechte deutsche Lokalisierung

ließe sich die Liste noch beliebig fortsetzen.

Konkurrierende Anforderungen müssen gewichtet und gemäß Ihrem Nutzen priorisiert werden. Nur so kann die Anwendung überhaupt umgesetzt werden und später eine hohe Akzeptanz erfahren. Je größer die Projekte sind, desto schwerer fällt diese Priorisierung. Liegt es an der Vielzahl der Beteiligten, der Unmenge von Anforderungen oder den unzähligen Abstimmungsrunden?

Auf alle Fälle eigenen sich Großprojekte besonders gut dazu, eine Priorisierung einfach zu unterlassen. Da werden Entscheidungen vertagt und man entwirft und entwickelt an einer Ecke schon mal weiter. Wenn es schon insgesamt nicht zusammen passt, ist wenigstens diese eine Komponente so richtig rund. Und falls das Projekt scheitert, sind eben die anderen Teilprojekte daran schuld. Sehr treffend beschrieben wird dieses Phänomen in den *Anti-Patterns* „Gelähmte Analyse" (vgl. [WikiAnPar]) und „Schleichende Feature-Hölle" (vgl. [WikiCrFeat]).

Das Fazit bleibt leider immer das gleiche: Das System kann entweder gar nicht umgesetzt werden oder nur mit einem äußerst unbefriedigenden Ergebnis. Wenn Sie Ihr Projekt erfolgreich abschließen wollen, müssen Sie auf einer sinnvollen Gewichtung der Anforderungen bestehen.

Mythos 5: Internationalisierung erledigen Entwicklungswerkzeuge

Beim Stichwort Internationalisierung denken viele zunächst an den Austausch von Oberflächenbeschriftungen und Fehlermeldungen – möglichst zur Laufzeit, auf Knopfdruck, per Menüpunkt. (Hatten Sie schon einmal das Bedürfnis, mitten in der Arbeit mit einer Anwendung die Sprache zu wechseln? Wahrscheinlich nicht. Genau deshalb reicht es in der Regel aus, die Sprache im Zuge der Anwendungsinitialisierung festzulegen, z. B. bei der Anmeldung des Benutzers.)

Diese Anforderungen können Sie normalerweise mit einem gewissen Fleiß zügig umsetzen. Entwicklungswerkzeuge unterstützen Sie beim Extrahieren der zu übersetzenden Zeichenketten und lagern diese in so genannte „RessourceBundle" aus. Erste Probleme tauchen zwar hier und da durch die Tatsache auf, dass Softwareentwickler in der Regel keine Übersetzer sind. Bei entsprechenden Budget lässt sich dieses Problem jedoch leicht lösen, indem die Übersetzung dieser extrahierten Ressourcen an Profis delegiert wird.

Die wirklich kritischen Stellen internationaler Anwendungen liegen aber in folgenden Bereichen:

- *Formatierung von Daten:* Dies betrifft meist die Darstellung von Datum, Zeit, Währung und Gleitkommazahlen.
- *Währung:* Zu berücksichtigen sind die Aktualität der Kurse, die Auszeichnung einer Referenzwährung sowie der Zeitpunkt der Umrechnung (zur Laufzeit oder redundante Datenhaltung).
- *Mehrsprachige Anwendungsdaten:* Ist die Oberfläche erst einmal mehrsprachig, so dürfen die Inhalte von Combo-Boxen natürlich nicht mehr einsprachig sein. Was nützen mir ein Internet-Shop, in dem ich die verschiedenen Versandarten nicht verstehe, weil sie in z. B. in spanisch sind? Noch schlimmer wird dies natürlich, wenn Daten wie Artikelbeschreibungen, Farben, Größen etc. nicht an lokale Gegebenheiten angepasst sind. Im B2C-E-Business haben derartige Unterlassungen fatale Folgen: Die Akzeptanz ausländischer Nutzer geht gegen Null, neue Märkte bleiben verschlossen.
- *Shortcuts:* Tastenkürzel für Buttons und Menüeinträge müssen mit den sprachlich verschiedenen Beschriftungen korrespondieren (insbesondere die <Alt>-Kombinationen, in denen ein Buchstabe der Button- oder Menü-Beschriftung unterstrichen wird).
- *Zeichensätze:* Werden Sprachen aus unterschiedlichen Sprachräumen unterstützt, so müssen entsprechende Zeichensätze zur Verfügung stehen und von allen beteiligten Komponenten unterstützt werden (z. B. auf dem Client zur Darstellung und in der Datenbank zur persistenten Speicherung von Daten[3]).
- *Zeitzonen:* läuft die Anwendung zeitgleich in verschiedenen Zeitzonen, so müssen für zeitrelevante Funktionen spezielle Dienste zur Verfügung gestellt werden.
- *Anwendungslogik:* Die Anwendungslogik darf selbstverständlich nicht zeichenkettenbasiert auf die Oberflächen-Beschriftungen bezogen sein. Diese Forderung klingt trivial, wird aber häufig von generierten Anwendungsteilen verletzt. Dann ist eine aufwändige manuelle Nachbearbeitung erforderlich.

Alle diese Dinge werden oftmals bei der Entscheidung für eine internationale Anwendung vergessen. Abbildung 3 zeigt hierfür ein häufiges Beispiel: In einem deutschen Adresseingabefeld ist die Frage nach dem „Staat" verwirrend und überflüssig. Die Maskenbeschriftungen wur-

[3]) *Dies führte z. B. zu erheblichen Schwierigkeiten, als ein Archiv-System für Radiosender aus einer deutschen Version in eine griechische Version lokalisiert wurde. Alles lief bestens bei Titeln aus der internationalen Popmusik. Erst als die ersten griechischen Volkslieder archiviert werden sollten, stellte man plötzlich fest, dass die Datenbank keine Unterstützung für griechische Zeichensätze bot.*

den angepasst, aber das Attribut wurde ungeprüft aus einem anderem Format (z.B. dem amerikanischen Adressformat) übernommen. Weitere sehr anschauliche Beispiele finden Sie in der „Interface Hall of Shame" (vgl. [IntHShm]). Wenn Sie es besser machen wollen, empfehle ich Ihnen zu diesem Thema das Buch von Czarnecki und Deitsch ([Cza01]).

Internationalisierung geht über Mehrsprachigkeit weit hinaus. Erhöhte Aufwände und Kosten entstehen in allen Phasen des Softwareentwicklungsprozesses und bei fast allen Aktivitäten – von der Analyse, über das Design und die Entwicklung bis hin zum Test. Aus diesem Grund sollten Sie bei der Aufnahme der Anforderung „Mehrsprachigkeit" sehr genau nachfragen, welche Wünsche an eine umfassende Internationalisierung damit verbunden sind.

Fazit

Die Lage ist ernst, aber nicht hoffnungslos. Glücklicherweise werden Sie in kaum einem Projekt allen diesen Legenden auf einmal begegnen. Und natürlich gibt es auch für jedes der beschrieben Szenarien die sinnvolle Anwendung im geeigneten Kontext.

Welche Erfahrungen haben Sie gemacht? Haben Sie einige Situationen aus Ihren Projekten wieder erkannt? Oder können Sie noch neue Mythen ergänzen? Über ein Feedback von Ihnen würde ich mich freuen. ■

Literatur & Links

[Ale99] C. Alexander et. al., A Pattern Language, Oxford University Press, 1977

[Bec99] K. Beck, Extreme programming explained, Addison-Wesley, 1999

[Bre90] R.W. Brednich, Die Spinne in der Yucca-Palme – Sagenhafte Geschichten von heute, C.H. Beck Verlag, 1990

[Cza01] D. Czarnecki, A. Deitsch, Java Internationalization, O'Reilly 2001

[Gam95] E. Gamma, R. Helm, R. Johnson, J Vlissides, Design Patterns, Addison-Wesley, 1995

[IntHShm] Interface Hall of Shame, siehe: digilander.libero.it/chiediloapippo/Engineering/iarchitect/global.htm

[WikiAnPar] Wiki – AntiPattern „Analysis Paralysis", siehe: c2.com/cgi/wiki?AnalysisParalysis

[WikiAntPat] Wiki – Übersicht und Katalog von „AntiPatterns", siehe: c2.com/cgi/wiki?AntiPatterns

[WikiArCod] Wiki – AntiPattern „Architects don't code", siehe: c2.com/cgi/wiki?ArchitectsDontCode

[WikiCrFeat] Wiki – AntiPattern „Creeping featuritis", siehe: c2.com/cgi/wiki?CreepingFeaturitis

[WikiPCIEn] Wiki – AntiPattern „PlugCompatibleInterchangeableEngineers", siehe: c2.com/cgi/wiki?PlugCompatibleInterchangeableEngineers

[WikiRInWh] Wiki – AntiPattern „Reinventing the wheel", siehe: c2.com/cgi/wiki?ReinventingTheWheel

[WikiSwArm] Wiki – AntiPattern „Swiss Army Knife", siehe: c2.com/cgi/wiki?SwissArmyKnife

[WikiYAGNI] Wiki – AntiPattern „You aren't gonna need it", siehe: c2.com/cgi/wiki?YouArentGonnaNeedIt