mitp

Sam
Newman

Microservices
Konzeption und Design

Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Sam Newman

Microservices

Konzeption und Design

Übersetzung aus dem Amerikanischen
von Knut Lorenzen

mitp

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-95845-082-0
1. Auflage 2015

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Authorized German translation of the English edition of *Building Microservices*
ISBN 9781491950357 © 2015 Sam Newman. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Lektorat: Sabine Schulz
Sprachkorrektorat: Maren Feilen
Coverbild: © Irochka, fotolia.de
Satz: III-satz, Husby, www.drei-satz.de

Inhaltsverzeichnis

Einleitung

Microservices sind ein Ansatz für verteilte Systeme, die die Nutzung feingranularer Services mit eigenen Entwicklungszyklen fördern, die sich gegenseitig zuarbeiten. Da Microservices vornehmlich im geschäftlichen Umfeld Anwendung finden, werden die bei herkömmlichen abgestuften Architekturen auftretenden Schwierigkeiten umgangen. Microservices nutzen außerdem die während des letzten Jahrzehnts entwickelten Technologien und Verfahren und vermeiden dadurch die Fallstricke, die mit vielen serviceorientierten Architekturen einhergehen.

Dieses Buch enthält eine Reihe konkreter Beispiele dafür, wie Microservices weltweit eingesetzt werden, etwa in Unternehmen wie Netflix, Amazon, Gilt und der REA-Gruppe, die allesamt festgestellt haben, dass ihnen die erhöhte Unabhängigkeit dieser Architektur große Vorteile bringt.

Wer sollte dieses Buch lesen?

Der Anwendungsbereich dieses Buches umfasst ein breites Spektrum, ebenso wie auch die Auswirkungen einer feingranularen Microservice-Architektur vielgestaltig sind. Es soll Leser ansprechen, die an verschiedenen Aspekten des Designs, der Entwicklung, des Deployments, des Testens und der Wartung dieser Systeme interessiert sind. Diejenigen Leser, die bereits damit begonnen haben, sich mit feiner unterteilten Architekturen zu beschäftigen – sei es nun einer vollkommen neuen Anwendung oder der Aufteilung eines bereits vorhandenen, eher monolithischen Systems –, werden viele praktische Ratschläge finden. Und auch denjenigen Lesern, die im Grunde nur wissen möchten, was der ganze Rummel eigentlich soll, wird geholfen, damit sie entscheiden können, ob Microservices für ihre Zwecke geeignet sind.

Der Grund für dieses Buch

Als es vor vielen Jahren zu meinen Aufgaben gehörte, anderen dabei zu helfen, Software schneller fertigzustellen, fing ich an, mich mit Anwendungsarchitekturen zu befassen. Mir war klar, dass automatisierte Infrastrukturen, Tests und kontinuierliche Weiterentwicklungen zwar durchaus hilfreich sind, man aber bald an die Grenzen des Machbaren stößt, wenn das grundlegende Design eines Systems es nicht erlaubt, schnell und einfach Modifizierungen daran vorzunehmen.

Zur selben Zeit experimentierten viele Unternehmen mit feiner unterteilten Architekturen, um vergleichbare Ergebnisse zu erzielen. Gleichzeitig sollten aber auch eine verbesserte Skalierbarkeit, eine größere Unabhängigkeit der Entwicklerteams oder eine vereinfachte Übernahme neuer Technologien ermöglicht werden. Sowohl meine eigenen Erfahrungen als auch die meiner Kollegen bei ThoughtWorks und anderen Unternehmen bestätigten die Tatsache, dass eine größere Zahl eingesetzter unabhängiger Services mit eigenen Entwicklungszyklen unweigerlich zu weiteren Problemen führt, mit denen man sich auseinandersetzen muss. Dieses Buch soll in gewisser Weise eine Art zentrale Anlaufstelle sein und helfen, die breite Palette von Themen, die zum Verständnis von Microservices nötig ist, zu beschreiben. So etwas hätte mir seinerzeit wirklich außerordentlich geholfen!

Zum Stand der Dinge

Das Thema »Microservices« ist einem ständigen Wandel unterworfen. Obwohl die Idee an sich nicht neu ist (auch wenn der Begriff es ist), haben die Erfahrungen der Nutzer auf der ganzen Welt zusammen mit dem Aufkommen neuer Technologien maßgeblichen Einfluss auf die Verwendungsweise. Aufgrund der schnellen Fortentwicklung in diesem Bereich habe ich versucht, mich in den folgenden Kapiteln weniger auf bestimmte Technologien als vielmehr auf die grundlegenden Konzepte zu konzentrieren, und zwar wohlwissend, dass sich Details der Implementierung stets schneller ändern als die dahinterstehenden Ideen. Dessen ungeachtet erwarte ich absolut, dass wir in einigen Jahren noch besser verstehen werden, wann der Einsatz von Microservices angebracht ist und wie sie vernünftigerweise eingesetzt werden.

Aufbau des Buches

Der Aufbau dieses Buches orientiert sich vornehmlich an den behandelten Themen. Sie können daher direkt zu einem bestimmten Thema springen, das Sie am meisten interessiert. Ich habe mich bemüht, wichtige Begriffe und Konzepte gleich in den ersten Kapiteln zu erläutern und gehe davon aus, dass selbst Leser, die sich als recht erfahren einschätzen, in jedem Kapitel noch etwas von Interesse entdecken. Grundsätzlich empfehle ich Ihnen, sich Kapitel 2 anzusehen, das einen Eindruck von der Tiefe des Themas vermittelt und umreißt, wie ich im weiteren Verlauf des Buches vorgehe, falls Sie sich mit den nachfolgenden Inhalten eingehender beschäftigen möchten.

Ich hoffe, ich habe die Kapitel für Leser, denen das Thema neu ist, in der richtigen Reihenfolge angeordnet, damit das Buch in sinnvoller Weise von vorn bis hinten durchgelesen werden kann.

Hier ein Überblick über den Inhalt des Buches:

Kapitel 1 – Microservices Wir beginnen mit einer Einführung in das Thema Microservices, in der die wesentlichen Vorteile, aber auch einige der Nachteile dargestellt werden.

Kapitel 2 – Der fortentwickelte Systemarchitekt In diesem Kapitel kommen die Schwierigkeiten zur Sprache, denen man als Systemarchitekt gegenübersteht, weil man Kompromisse eingehen muss. Außerdem wird erörtert, was bei der Verwendung von Microservices alles zu beachten ist.

Kapitel 3 – Gestaltung von Services Hier werden die Grenzen der Microservices erkundet. Um uns auf das Wesentliche zu konzentrieren, kommen dabei Verfahren des vom Anwendungsbereich geprägten Designs (*Domain-Driven Design*, DDD) zum Einsatz.

Kapitel 4 – Integration An dieser Stelle beschäftigen wir uns eingehender mit bestimmten Auswirkungen der Technologien und erörtern, welche Arten der Zusammenarbeit von Services am nützlichsten sind. Des Weiteren werden wir auf die Themen Benutzerschnittstelle und Integration vorhandener und seriengefertigter Produkte (Commercial off-the-shelf , COTS) eingehen.

Kapitel 5 – Die Aufspaltung des Monolithen In vielen Fällen richtet sich das Interesse auf die Microservices, um sie in großen, nur schwer änderbaren monolithischen Systemen sozusagen als Gegenmittel einzusetzen. Genau dieser Ansatz wird in diesem Kapitel ausführlich untersucht.

Kapitel 6 – Deployment Das Buch ist zwar weitgehend theoretischer Natur, allerdings wurde kaum ein anderes der behandelten Themen so sehr durch die jüngsten technologischen Neuerungen beeinflusst wie das Deployment. Dieser Aspekt wird hier eingehender betrachtet.

Kapitel 7 – Testen Dieses Kapitel geht dem Thema Testen auf den Grund – einem Bereich, der gerade beim Deployment mehrerer eigenständiger Services von Bedeutung ist. Besonders interessant ist hier die Rolle, die *Consumer-Driven Contracts* (CDCs) für die Gewährleistung der Qualität unserer Software spielen.

Kapitel 8 – Monitoring Die vor der Auslieferung durchgeführten Tests helfen uns nicht weiter, wenn Probleme erst auftreten, nachdem die Software bereits online gestellt wurde. In diesem Kapitel wird untersucht, wie sich verteilte Systeme überwachen lassen und wie man die bei solchen Systemen auftretende Komplexität handhabt.

Kapitel 9 – Sicherheit Hier betrachten wir die Sicherheitsaspekte von Microservices und untersuchen, wie Authentifizierung und Autorisierung zwischen Benutzer und Service bzw. zwischen Services gehandhabt werden. Sicherheit ist ein sehr wichtiges Thema, das allzu leicht vernachlässigt wird. Ich bin zwar keineswegs ein

Sicherheitsexperte, hoffe jedoch, dass dieses Kapitel Ihnen dabei hilft, beim Aufbau Ihrer Systeme bedeutsame Sicherheitsaspekte in Betracht zu ziehen – insbesondere, wenn es sich um Microservice-Systeme handelt.

Kapitel 10 – Conways Gesetz und Systemdesign Dieses Kapitel widmet sich dem Zusammenspiel zwischen Organisationsstruktur und Systemarchitektur. Wie viele Unternehmen bereits feststellen mussten, führt es zu Problemen, wenn diese beiden Faktoren nicht aufeinander abgestimmt sind. Wir werden versuchen, die Ursachen dieses Dilemmas zu erörtern und betrachten verschiedene Möglichkeiten, das Systemdesign an die Struktur des Entwicklerteams anzugleichen.

Kapitel 11 – Microservices skalieren Hier werden wir uns ansehen, wie Microservices skalieren, damit wir die bei einer großen Zahl von Services und hohem Datenaufkommen wachsende Wahrscheinlichkeit eines Systemausfalls handhaben können.

Kapitel 12 – Auf den Punkt gebracht Das letzte Kapitel bemüht sich, die Besonderheiten von Microservices hervorzuheben. Es enthält eine Liste von sieben für Microservices geltende Prinzipien und arbeitet die Kernpunkte des Buches heraus.

Konventionen dieses Buches

In diesem Buch gelten folgende typografische Konventionen:

- Neue Begriffe, Dateinamen und Dateinamenerweiterungen sind *kursiv* gedruckt.
- URLs und E-Mail-Adressen sind im `Hyperlink-Format` dargestellt.
- Für Programmlistings oder im Fließtext vorkommende Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter wird eine `nicht-proportionale Schrift` verwendet.
- Für vom Benutzer einzugebende Befehle und Texte wird eine **`fette nicht-proportionale Schrift`** benutzt.
- Texte, die vom Benutzer durch eigene Eingaben oder aus dem Kontext ersichtliche Werte ersetzt werden sollen, sind in einer *`kursiven nicht-proportionalen Schrift`* gedruckt.

Danksagungen

Ohne Lindy Stephens würde es dieses Buch, das ihr gewidmet ist, nicht geben. Sie hat mich ermuntert, es in Angriff zu nehmen, hat mich bei dem oftmals anstrengenden Entstehungsprozess unterstützt und ist die beste Partnerin, die man sich nur wünschen kann. Ich möchte dieses Buch auch meinem Vater Howard Newman widmen, der immer für mich da war. Für euch beide ist dieses Buch.

Besonderer Dank gilt Ben Christensen, Vivek Subramaniam und Martin Fowler, die mir beim Schreiben dieses Buches ausführliche Rückmeldungen gaben und das vorliegende Werk dadurch mitgestalteten. Ich möchte auch James Lewis danken, mit dem ich bei der Besprechung der im Buch vorgestellten Ideen so manches Bier getrunken habe. Ohne eure Hilfe und Beratung wäre dieses Buch nur ein Schatten seiner selbst.

Außerdem haben viele andere mitgeholfen und die ersten Versionen des Buches kommentiert. Ich möchte insbesondere (in wahlloser Reihenfolge) folgenden Personen danken: Kane Venables, Anand Krishnaswamy, Kent McNeil, Charles Haynes, Chris Ford, Aidy Lewis, Will Thames, Jon Eaves, Rolf Russell, Badrinath Janakiraman, Daniel Bryant, Ian Robinson, Jim Webber, Stewart Gleadow, Evan Bottcher, Eric Sword, Olivia Leonard, sowie allen anderen Kollegen bei ThoughtWorks und in der Branche, die mir dabei geholfen haben, so viel zu erreichen.

Schließlich möchte ich auch der Belegschaft von O'Reilly USA danken, insbesondere Mike Loukides, der mich angeheuert hat, meinem Lektor Brian MacDonald sowie Rachel Monaghan, Kristen Brown, Betsy Waliszewski und all den anderen Leuten, die auf mir unbekannte Weise an der Entstehung dieses Buches beteiligt waren.

Über den Autor

Sam Newman ist als Technologe bei ThoughtWorks tätig, wo er einerseits als Berater für Klienten arbeitet und andererseits auch als Systemarchitekt für ThoughtWorks' interne Systeme verantwortlich ist. Im Rahmen seiner Beratertätigkeit arbeitete er mit zahlreichen internationalen Unternehmen aus den unterschiedlichsten Geschäftsbereichen zusammen, wobei er oft mit einem Bein im Lager der Entwickler und mit dem anderen im Lager des IT-Betriebs stand. Wenn Sie ihn nach seiner Tätigkeit befragen, würde er sagen: »Ich helfe den Leuten dabei, bessere Softwaresysteme zu entwickeln.« Er veröffentlicht Artikel, hält Vorträge auf Fachtagungen und arbeitet hin und wieder auch an Open-Source-Projekten mit.

Kapitel 1

Microservices

Seit vielen Jahren erkunden wir inzwischen immer bessere Methoden für die Systementwicklung. Wir haben aus den Erfahrungen der Vergangenheit gelernt, neue Technologien adaptiert und erleben nun, wie eine neue Generation von Technologieunternehmen ganz verschiedene Ansätze bei der Errichtung von IT-Systemen verfolgt, die nicht nur ihre Kunden, sondern auch die eigenen Entwickler zufriedener stellen.

Eric Evans Buch *Domain-Driven Design* (Addison-Wesley) hat uns gelehrt, wie wichtig es ist, die reale Welt in unserem Code widerzuspiegeln und bessere Möglichkeiten aufgezeigt, unsere Systeme zu modellieren. Das Konzept *Continuous Delivery* führt vor, wie es gelingen kann, die Software effektiver und effizienter zur Serienreife zu bringen und schärft uns den Grundgedanken ein, jede einzelne Version wie einen zur Veröffentlichung geeigneten *Release Candidate* zu behandeln. Unsere Einsicht in die Funktionsweise des Webs hat uns dazu gebracht, bessere Methoden zur Kommunikation zwischen Computern zu entwickeln. Alistair Cockburns Konzept der hexagonalen Architektur (`http://alistair.cockburn.us/Hexagonal+architecture`) wies uns den Weg fort von abgestuften Architekturen, in denen sich die Anwendungslogik verbergen konnte. Virtualisierungsplattformen erlaubten es uns, Maschinen beliebiger Größe bereitzustellen und diese dank automatisierter Infrastrukturen in großem Maßstab einzusetzen. Einige große und erfolgreiche Unternehmen wie Amazon und Google befürworten die Ansicht, dass kleine Entwicklerteams für den vollständigen Entwicklungszyklus ihrer Services zuständig sein sollten. Und erst in jüngster Zeit hat Netflix uns den Aufbau robuster Systeme von einer Größe vorgeführt, die vor nur zehn Jahren kaum vorstellbar war.

Domain-Driven Design. Continuous Delivery. Virtualisierung nach Bedarf. Automatisierte Infrastrukturen. Kleine, eigenständige Entwicklerteams. Skalierbare Systeme. Aus diesem Umfeld sind die Microservices hervorgegangen. Sie wurden nicht vorab entwickelt oder geplant, sondern entstanden vielmehr aus den bei der praktischen Anwendung zu beobachtenden Tendenzen oder Mustern heraus. Microservices existieren also im Grunde genommen nur, weil es all die anderen genannten Dinge gibt. Ich werde im weiteren Verlauf des Buches immer wieder Verbindungen zu diesen vorausgehenden Entwicklungen aufzeigen, so dass sich schlussendlich beim Aufbau, der Verwaltung und der Fortentwicklung von Microservices ein geschlossenes Gesamtbild ergibt.

Viele Unternehmen haben nach der Einführung feingranularer Microservice-Architekturen festgestellt, dass ihre Software schneller zur Serienreife gelangt und selbst neue Technologien leicht übernommen werden können. Microservices gestatten uns eine deutlich größere Entscheidungsfreiheit, um auf die unvermeidlichen Änderungen reagieren zu können, die uns alle betreffen.

1.1 Was sind Microservices?

Microservices sind kleine, eigenständige Services, die kollaborieren bzw. sich gegenseitig zuarbeiten. Lassen Sie uns diese Definition im Folgenden etwas genauer fassen und die besonderen Eigenschaften der Microservices betrachten.

1.1.1 Klein und darauf spezialisiert, eine bestimmte Aufgabe richtig gut zu erledigen

Zur Ergänzung neuer Funktionalitäten schreiben wir zusätzlichen Code und erweitern damit zwangsläufig auch die Codebasis. Im Laufe der Zeit kann es allerdings schwierig werden, die Stelle zu finden, an der eine Modifikation erforderlich ist, weil die Codebasis so umfangreich geworden ist. Trotz des Strebens nach einer klar strukturierten, modularen monolithischen Codebasis werden die willkürlich gezogenen Grenzen nur allzu oft überschritten. Der zu ähnlichen Funktionen gehörende Code beginnt sich überall auszubreiten und erschwert es, Fehler zu beheben oder Änderungen vorzunehmen.

Bei monolithischen Systemen kämpfen wir dagegen an, indem wir versuchen, zusammenhängenden Code zu schreiben, nicht selten durch Abstrahierung oder das Erstellen von Modulen. Diese Geschlossenheit, also das Bestreben, zusammengehörigen Code auch zusammenzuhalten, spielt bei Microservices eine wichtige Rolle. Das wird auch durch das von Robert C. Martin definierte *Prinzip einer einzigen Zuständigkeit* bestätigt, das besagt: »Fasse Dinge zusammen, die aus demselben Grund geändert werden, und trenne Dinge, die aus unterschiedlichen Gründen geändert werden.«

Microservices verfolgen denselben Ansatz bezüglich voneinander unabhängiger Services. Wir beschränken unsere Services auf eng begrenzte Geschäftsvorgänge, damit offensichtlich ist, wo sich der zu einer bestimmten Funktionalität zugehörige Code befindet. Durch diese Fokussierung eines Services auf eine explizite Schnittstelle entgeht man der Versuchung, den Code zu groß werden zu lassen – und damit auch all den anderen Schwierigkeiten, die damit einhergehen können.

Man stellt mir oft die Frage: *Wie klein ist klein?* Hier eine Anzahl von Codezeilen anzugeben, ist problematisch, weil manche Programmiersprachen ausdrucksstärker sind als andere und daher dieselben Aufgaben in weniger Codezeilen verrichten können. Außerdem ist zu berücksichtigen, dass wir möglicherweise mehrfache Abhängigkeiten von anderem Code einbringen, der seinerseits aus

vielen Zeilen besteht. Darüber hinaus sind manche Teile Ihrer Anwendung möglicherweise aus gutem Grund komplex und erfordern daher mehr Code. Jon Eaves (`http://RealEstate.com.au`) beschreibt einen Microservice als etwas, das in zwei Wochen neu geschrieben werden kann – eine Faustregel, die bei seinen besonderen Rahmenbedingungen tatsächlich sinnvoll ist.

Eine weitere, etwas banale Antwort auf obige Frage lautet: *Klein genug, aber nicht zu klein*. Als Redner auf Fachtagungen stelle ich fast immer die Frage: *Wer betreibt ein zu großes System und würde es gern aufspalten?* Nahezu jeder der Anwesenden hebt dann die Hand. Wir scheinen einen ausgeprägten Sinn dafür zu besitzen, etwas zu Großes zu erkennen. Folglich könnte man argumentieren, dass Code, der nicht mehr zu umfangreich *erscheint*, wahrscheinlich kompakt genug ist.

Bei der Beantwortung der Frage *Wie klein?* spielt es eine wichtige Rolle, in welchem Ausmaß der Service auf die Struktur des Entwicklerteams abgestimmt ist. Wenn die Codebasis zu groß ist, um von einem kleinen Entwicklerteam gehandhabt zu werden, ist es sinnvoll, sie aufzuspalten. Diese Abstimmung auf betriebliche Gegebenheiten wird später noch zur Sprache kommen.

Wenn es darum geht, wie klein denn nun klein genug ist, stelle ich mir das gern folgendermaßen vor: Je kleiner ein Service ist, desto stärker kommen die Vor- und Nachteile der Microservice-Architektur zum Tragen. Kleinere Services ziehen größere Vorteile aus der wechselseitigen Abhängigkeit. Gleichzeitig steigt aber auch die Komplexität, die sich aus dem Vorhandensein von immer mehr veränderlichen Systembestandteilen ergibt, was wir im weiteren Verlauf des Buches immer wieder untersuchen werden. Sobald man die Komplexität besser in den Griff bekommt, kann man auch immer kleinere Services anstreben.

1.1.2 Eigenständigkeit

Unser Microservice ist ein eigenständiges Gebilde. Dabei kann es sich um einen isolierten PaaS-Service (*Platform-as-a-Service*, Bereitstellung einer Computerplattform in der Cloud) handeln oder um einen eigenen Betriebssystemprozess. Wir versuchen zu vermeiden, mehrere Services auf derselben Maschine zu betreiben, auch wenn der Begriff *Maschine* heutzutage ziemlich vage ist. Diese Isolierung kann zwar etwas Verwaltungsaufwand verursachen, aber die damit einhergehende Vereinfachung trägt sehr dazu bei, das verteilte System besser zu verstehen. Außerdem werden viele der bei dieser Form des Deployments auftretenden Probleme durch neue Technologien deutlich entschärft.

Die Kommunikation der Services untereinander erfolgt durch Aufrufe über das Netzwerk. Dadurch wird die Isolierung der Services betont und man geht den Gefahren einer engen Kopplung aus dem Weg.

Die Services müssen unabhängig voneinander geändert und erneut bereitgestellt werden können und ohne Änderungen aufseiten der Consumer auskommen. Wir

müssen uns Gedanken darüber machen, welche Informationen die Services preisgeben sollten und welche sie verbergen dürfen. Wenn zu viele Informationen gemeinsam genutzt werden, führt das zu einer Kopplung der Consumer an die interne Darstellung. Das vermindert die Eigenständigkeit, weil dadurch bei Änderungen eine zusätzliche Abstimmung mit den Consumern erforderlich wird.

Unser Service bietet eine API (*Application Programming Interface*, Programmierschnittstelle) an, über die die anderen mit unserem Service kollaborierenden Services mit ihm kommunizieren. Wir müssen außerdem darüber nachdenken, welche Technologie am besten geeignet ist, damit durch die API selbst keine Kopplung an Consumer entsteht. Dazu kann es erforderlich sein, technologieunabhängige APIs einzusetzen, damit die Auswahl der eingesetzten Technologien nicht beschränkt wird. Wir werden im Verlauf des Buches immer wieder auf die Bedeutung einer guten entkoppelten API zurückkommen.

Ohne Entkopplung fällt alles wie ein Kartenhaus in sich zusammen. Die Gretchenfrage lautet: Können Sie eine Änderung an einem Service vornehmen und ihn erneut deployen, ohne irgendetwas anderes zu ändern? Lautet die Antwort Nein, werden sich viele der im Buch erörterten Vorteile nur schwer erzielen lassen.

1.2 Die wichtigsten Vorteile

Microservices besitzen zahlreiche und vielgestaltige Vorteile. Viele davon bringt jedes verteilte System ohnehin mit sich, allerdings schöpfen Microservices das Potenzial dieser Vorteile tendenziell vor allem deswegen besser aus, weil sie bei der Umsetzung der Konzepte verteilter Systeme und serviceorientierter Architekturen (*Service-Oriented Architecture*, SOA) viel weiter gehen.

1.2.1 Verschiedenartige Technologien

Bei einem aus mehreren kollaborierenden Services bestehendem System können bei den einzelnen Services unterschiedliche Technologien zum Einsatz kommen. Dadurch ist es möglich, für jede Aufgabe das am besten geeignete Werkzeug auszuwählen, anstatt sich mit standardisierten Allzwecklösungen zufriedengeben zu müssen, die sich oftmals als kleinster gemeinsamer Nenner herausstellen.

Falls ein bestimmter Teil unseres Systems beschleunigt werden soll, könnten wir uns dazu entschließen, eine andere Technologie einzusetzen, die besser dafür geeignet ist, die erforderliche Geschwindigkeit zu erzielen. Es wäre ebenfalls möglich, die in verschiedenen Teilen des Systems anfallenden Daten auf unterschiedliche Weise zu speichern. Bei Anwendungen eines sozialen Netzwerks beispielsweise könnten wir die Interaktionen der Benutzer in einer grafikfähigen Datenbank speichern, um die hochgradig verknüpften Verbindungen in Form

eines *Social Graph* widerzuspiegeln. Die Nachrichten der Benutzer hingegen könnten in einem dokumentorientierten Datenspeicher abgelegt werden, was zu einer heterogenen Architektur wie der in Abbildung 1.1 gezeigten führt.

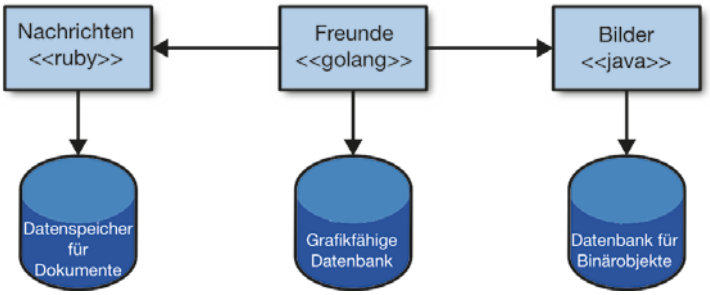

Abb. 1.1: Microservices gestatten eine einfachere Übernahme unterschiedlicher Technologien.

Microservices ermöglichen außerdem eine schnellere Übernahme von Technologien, und wir können uns gegebenenfalls auch Verbesserungen zunutze mache. Die größten Hürden beim Ausprobieren und bei der Übernahme neuer Technologien sind die damit verbundenen Risiken. Wenn ich bei einer monolithischen Anwendung eine neue Programmiersprache, eine andere Datenbank oder ein Framework ausprobieren möchte, sind bei jeder Änderung weite Teile meines Systems betroffen. Bei einem aus mehreren Services bestehenden System stehen mir zum Austesten neuer Technologien hingegen gleich mehrere Möglichkeiten zur Verfügung. Ich kann mir einen unkritischen Service aussuchen, dessen Ausfall nur mit geringem Risiko verbunden wäre, und die Technologie dort einsetzen, wohlwissend, dass ich nur sehr begrenzten Schaden anrichten kann. Viele Unternehmen betrachten diese Möglichkeit, neue Technologien einzubinden, als echten Vorteil.

Die Integration mehrerer Technologien ist natürlich ohne einen gewissen Mehraufwand nicht machbar. Manche Unternehmen schränken etwa die Auswahl der Programmiersprache ein. Netflix und Twitter beispielsweise verwenden als Plattform vornehmlich die *Java Virtual Machine* (JVM), weil sie über ein sehr gutes Verständnis der Zuverlässigkeit und der Leistungsfähigkeit dieses Systems verfügen. Beide Unternehmen entwickeln auch Bibliotheken und Hilfswerkzeuge für die JVM, was den Betrieb im großen Maßstab sehr erleichtert, aber den Einsatz nicht auf Java beruhender Services oder Clients erschwert. Jedoch beschränken sich selbst Twitter und Netflix nicht für sämtliche Aufgaben nur auf diese eine Technologie. Ein weiterer Kontrapunkt hinsichtlich der Bedenken, Technologien miteinander zu vermischen, ist auch die Größe der Codebasis: Wenn ich meinen Microservice tatsächlich in zwei Wochen komplett neu programmieren kann, sind die mit der Übernahme neuer Technologien verbundenen Risiken weitgehend entschärft.

Sie werden bei der Lektüre des Buches feststellen, dass es bei vielen Dingen, die Microservices betreffen, immer wieder darum geht, das richtige Gleichgewicht zu finden. Die Auswahl passender Technologien wird in Kapitel 2 erläutert, das sich vornehmlich mit der Fortentwicklung der Systemarchitektur beschäftigt. In Kapitel 4, das die Integration zum Thema hat, werden Sie erfahren, wie Sie Ihre Services voneinander unabhängig weiterentwickeln können, ohne sie übermäßig miteinander zu koppeln.

1.2.2 Belastbarkeit

Bei der Gewährleistung einer hohen Belastbarkeit spielt das Abschotten der Services voneinander eine entscheidende Rolle: Wenn eine Komponente des Systems ausfällt, ohne die anderen in Mitleidenschaft zu ziehen, können Sie die betreffende Komponente isolieren und der Rest des Systems läuft weiter. Es liegt nahe, die Abschottung den Aufgaben der Services entsprechend vorzunehmen. Wenn bei einer monolithischen Anwendung ein Service ausfällt, funktioniert gar nichts mehr. Man kann monolithische Systeme auf mehreren Maschinen betreiben, um die Ausfallwahrscheinlichkeit zu verringern – bei der Verwendung von Microservices ist es allerdings möglich, Systeme zu entwickeln, deren Funktionalität bei einem Totalausfall von Services nur schrittweise abgebaut wird.

Wir sollten aber dennoch vorsichtig bleiben. Man muss die neuen Fehlerquellen, mit der verteilte Systeme vornehmlich zu kämpfen haben, genau verstehen, damit gewährleistet ist, dass unser Microservice-System Vorteile aus der erhöhten Belastbarkeit zieht. Netzwerke können nun einmal ausfallen – und das tun sie auch. Gleiches gilt für die Maschinen, auf denen das System läuft. Wir müssen daher wissen, wie Ausfälle zu handhaben sind und welche Auswirkungen, sofern vorhanden, diese Ausfälle für den Endbenutzer der Software haben.

In Kapitel 11 werden wir uns noch eingehender mit der Verbesserung der Belastbarkeit und der Handhabung von Fehlerzuständen beschäftigen.

1.2.3 Skalierung

Bei einem großen, monolithischen Service muss alles gleichzeitig skaliert werden. Wenn auch nur ein kleiner Teil des Gesamtsystems nicht hinreichend leistungsfähig ist, dieser Teil aber in einer riesigen monolithischen Anwendung »eingesperrt« ist, muss das System als Ganzes skaliert werden. Bei der Verwendung kleinerer Services hingegen reicht es aus, nur die Leistungsfähigkeit der betroffenen Services zu erhöhen, was es ermöglicht, andere Teile des Systems auf weniger leistungsstarker Hardware zu betreiben (Abbildung 1.2).

Der Online-Modehändler Gilt hat Microservices aus genau diesem Grund eingeführt. Als Gilt 2007 seine Geschäftstätigkeit aufnahm, setzte das Unternehmen eine monolithische Rails-Anwendung ein. Ab 2009 konnte das System die zuneh-

mende Last jedoch nicht mehr bewältigen. Nach der Aufspaltung wesentlicher Teile des Systems war Gilt dann aber wieder in der Lage, den aufkommenden Datenverkehr auch in Stoßzeiten zu handhaben. Heute laufen dort mehr als 450 Microservices, jeder davon auf mehreren unabhängigen Maschinen.

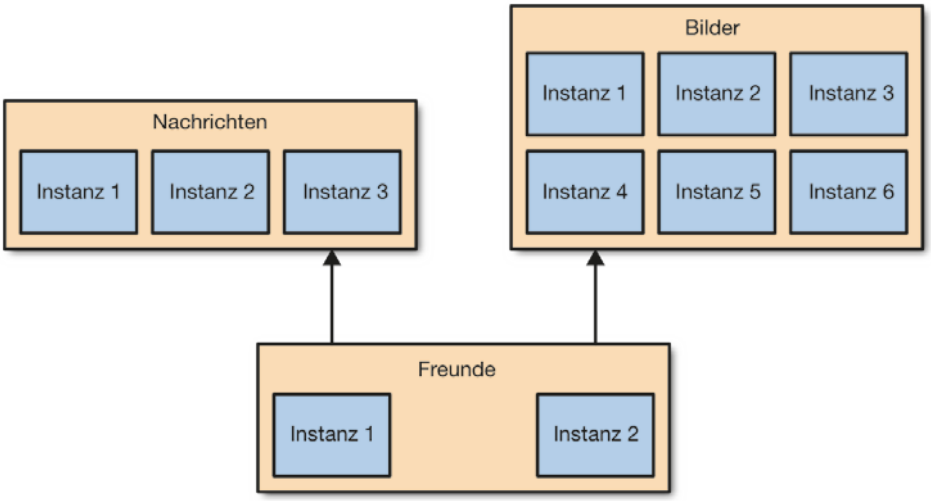

Abb. 1.2: Es reicht aus, nur die Leistungsfähigkeit betroffener Microservices zu erhöhen.

Beim Einsatz von Systemen, die Ressourcen nach Bedarf bereitstellen, wie z.B. Amazon Web Services (AWS), ist es sogar möglich, auch die Skalierung nur nach Bedarf vorzunehmen, nämlich nur für die Systembestandteile, bei denen es erforderlich ist. Dadurch wird es möglich, die anfallenden Kosten wesentlich effektiver zu steuern. Es kommt nur selten vor, dass ein architektonischer Ansatz so eng mit nahezu sofortigen Kosteneinsparungen verknüpft ist.

1.2.4 Komfortables Deployment

Wenn bei einer aus Millionen Codezeilen bestehenden monolithischen Anwendung auch nur eine Zeile geändert wird, muss das Deployment für die gesamte Anwendung erneut durchgeführt werden, um diese Änderung zur Verfügung zu stellen. Das kann allerdings erhebliche Auswirkungen nach sich ziehen und birgt Risiken. In der Praxis werden solche risikobehafteten Deployments mit weitreichenden Auswirkungen aufgrund der verständlichen Sorgen, die man sich in diesem Zusammenhang macht, nur selten durchgeführt. Das bedeutet aber leider auch, dass sich die Modifikationen bis zum nächsten Release anhäufen und tatsächlich veröffentlichte Versionen somit jede Menge Anpassungen und Modifikationen aufweisen. Und je größer die Unterschiede zwischen den Versionen sind, desto höher ist auch das Risiko, Fehler zu begehen!

Bei der Verwendung von Microservices können wir dagegen Änderungen an einem einzelnen Service vornehmen und diesen dann unabhängig vom übrigen System deployen. Das ermöglicht es uns, den neuen Code schneller zu deployen. Falls ein Problem auftritt, kann es schnell einem bestimmten Service zugeordnet werden, was es wiederum erlaubt, Änderungen gegebenenfalls auch schnell wieder rückgängig zu machen. Darüber hinaus kann man auf diese Weise dem Kunden neue Funktionalitäten schneller zur Verfügung stellen. Das ist einer der Hauptgründe dafür, dass Unternehmen wie Amazon und Netflix diese Architektur verwenden – um zu gewährleisten, dass es bei der Veröffentlichung neuer Software so wenig Hindernisse wie möglich gibt.

Die auf diesem Gebiet eingesetzte Technologie hat sich in den letzten Jahren drastisch verändert. In Kapitel 6 werden wir das Deployment im Microservices-Umfeld noch genauer in Augenschein nehmen.

1.2.5 Betriebliche Abstimmung

Viele von uns wissen aus eigener Erfahrung um die Schwierigkeiten, die große Entwicklerteams und eine große Codebasis bereiten können. Diese Probleme können sich bei verteilten Teams noch verschärfen. Außerdem ist bekannt, dass kleinere Teams, die mit einer kleineren Codebasis arbeiten, tendenziell produktiver sind.

Microservices ermöglichen eine bessere Abstimmung der Architektur auf die Unternehmensstruktur und helfen uns dabei, die Anzahl der an einer bestimmten Codebasis arbeitenden Entwickler zu minimieren und so ein optimales Verhältnis zwischen Teamgröße und Produktivität zu erreichen. Außerdem können die Zuständigkeiten der Entwicklerteams umverteilt werden, damit die an einem bestimmten Service arbeitenden Entwickler am selben Ort tätig sind. Wir werden dieses Thema noch genauer betrachten, wenn in Kapitel 10 Conways Gesetz erörtert wird.

1.2.6 Modularer Aufbau

Man verspricht sich vom Einsatz verteilter Systeme und serviceorientierter Architekturen insbesondere, dass dadurch Möglichkeiten zur Wiederverwendung von Funktionalitäten eröffnet werden. Im Fall der Microservices ist vorgesehen, dass die Funktionalitäten auf verschiedene Weise für unterschiedliche Aufgaben nutzbar sind. Das kann besonders wichtig sein, wenn wir darüber nachdenken, wie die Nutzer unsere Software verwenden. Die Zeiten, in denen wir unsere Überlegungen entweder auf die Websiteversion für Desktoprechner oder Mobilgeräte beschränken konnten, sind lange vorbei. Heutzutage sind eine Unzahl von Kombinationen der Möglichkeiten des Webs, nativer Anwendungen, des mobilen Webs sowie von Tablet-Anwendungen oder tragbaren Geräten (»Wearables«) zu berücksichtigen. Während die Unternehmen dabei sind, sich von diesen ein-

engenden Kommunikationskanälen zu verabschieden und sich eher ganzheitlichen Konzepten zur Kundenbindung zuwenden, benötigen wir Architekturen, die damit schritthalten können.

Mit Microservices öffnen wir unsere Systeme gewissermaßen einen Spaltbreit und machen einen Teil davon Außenstehenden zugänglich. Wenn sich die Umstände ändern, können wir unser System zur Anpassung anders aufbauen. Bei monolithischen Anwendungen steht mir als Außenstehendem oft nur ein vergleichsweise grobgranularer Zugang zur Verfügung. Wenn ich diesen Zugang weiter aufspalten möchte, um etwas Nützlicheres zu erhalten, brauche ich einen Vorschlaghammer! In Kapitel 5 werden wir erörtern, wie man bereits vorhandene monolithische Systeme aufspaltet und sie in hoffentlich wiederverwendbare und neu zusammensetzbare Microservices umwandelt.

1.2.7 Austauschbarkeit

Wenn Sie in einem Unternehmen mittlerer Größe oder sogar in einer großen Organisation tätig sind, stehen die Chancen nicht schlecht, dass dort ein großes, garstiges, veraltetes System in irgendeiner Ecke ein trauriges Dasein fristet. Das System, mit dem niemand etwas zu tun haben will. Das System, das für die Arbeitsweise Ihres Unternehmens unentbehrlich ist, aber in einem seltsamen Fortran-Dialekt programmiert ist und nur auf einer Hardware läuft, die besser vor 25 Jahren ausrangiert worden wäre. Warum wurde es nicht schon längst ersetzt? Sie ahnen es schon: Es wäre zu umständlich und zu riskant.

Bei einzelnen, kleinen Services ist der Aufwand für das Ersetzen einiger oder sogar aller Services durch eine verbesserte Implementierung viel leichter zu handhaben. Wie oft haben Sie schon an einem einzigen Tag mehr als hundert Codezeilen verworfen, ohne sich dabei allzu große Sorgen gemacht zu haben? Bei der Verwendung von Microservices, die oft von vergleichbarer Größe sind, fällt es nicht besonders schwer, sich dazu durchzuringen, einen Service neu zu schreiben oder vollständig zu entfernen.

Entwicklerteams, die den Microservice-Ansatz verfolgen, haben überhaupt kein Problem damit, bei Bedarf Services komplett neu zu programmieren oder überflüssige Services zu löschen. Wenn die Codebasis nur einige hundert Zeilen lang ist, wird wohl kaum jemand eine emotionale Bindung dazu aufbauen, und der Aufwand für das Ersetzen eines Services ist ziemlich gering.

1.3 Was ist mit serviceorientierten Architekturen?

Bei serviceorientierten Architekturen (*Service-Oriented Architecture*, SOA) handelt es sich um einen Designansatz, bei dem mehrere Services kollaborieren, um letzten Endes einen Satz an Funktionalitäten bereitzustellen. Hier ist mit dem Begriff »Service« typischerweise ein eigenständiger Betriebssystemprozess gemeint. Die

Kommunikation zwischen diesen Services erfolgt über das Netzwerk, nicht durch Funktionsaufrufe innerhalb eines Prozesses.

SOA wurde entwickelt, um den Herausforderungen großer monolithischer Anwendungen etwas entgegenzusetzen. Dieser Ansatz zielt darauf ab, die Wiederverwendbarkeit von Software zu fördern. Beispielsweise könnten zwei oder mehr Programme für Endanwender dieselben Services nutzen. Die Wartung oder das Umschreiben von Software soll dadurch erleichtert werden, denn theoretisch könnten wir einen Service durch einen anderen ersetzen, ohne dass irgendjemand das bemerkt, sofern sich die Verwendungsweise des Services nicht allzu sehr ändert.

Im Grunde genommen ist SOA eine sehr vernünftige Idee, allerdings gibt es – trotz einer ganzen Reihe von Anläufen – keinen echten Konsens, wie SOA *richtig* umzusetzen ist. Meiner Ansicht nach betrachtet ein Großteil der Branche das Problem nicht ganzheitlich genug und präsentiert eine verlockende Alternative zur Sichtweise einiger Anbieter in diesem Umfeld.

Viele der Probleme, die SOA angelastet werden, hängen eigentlich mit anderen Dingen zusammen: Kommunikationsprotokolle (z.B. SOAP, *Simple Object Access Protocol*), Software zum Datenaustausch (*Middleware*) verschiedener Hersteller, fehlende Orientierungshilfe bei der Aufteilung der Services oder falsche Empfehlungen zur Auswahl der Stelle, an der ein System aufgespalten werden sollte. Ein Zyniker würde sagen, dass die Hersteller den SOA-Trend für sich vereinnahmt (und in einigen Fällen gefördert) haben, um die Verkäufe ihrer Produkte anzukurbeln, und dass ebendiese Produkte die SOA-Ziele letzten Endes untergraben.

Die landläufige Meinung über SOA trägt nicht zum Verständnis bei, wie etwas Großes in etwas Kleines aufgespalten wird. Es geht nicht darum, wie groß zu groß ist. Es geht hingegen zu wenig um realistische, praxisnahe Methoden, die gewährleisten, dass Services nicht zu eng gekoppelt werden. Die Summe der ungesagten Dinge ist der Ursprung für viele der Fallstricke, die mit SOA in Verbindung gebracht werden.

Der Microservice-Ansatz ist aus praktischen Anwendungen hervorgegangen und baut dabei auf unser besseres Verständnis der Systeme und Architekturen auf, um SOA richtig umzusetzen. Stellen Sie sich also Microservices lieber als einen bestimmten SOA-Ansatz vor, in derselben Weise, wie XP (*Extreme Programming*) oder Scrum bestimmte Ansätze der agilen Softwareentwicklung sind.

1.4 Weitere Verfahren zur Aufspaltung

Wenn man der Sache auf den Grund geht, stellt man fest, dass viele der Vorteile einer Microservice-basierten Architektur auf zwei Dingen beruhen: der Aufteilung der Services und der Tatsache, dass eine viel größere Auswahl an möglichen Herangehensweisen für Problemlösungen zur Verfügung steht. Aber könnten auch andere vergleichbare Verfahren zur Aufspaltung dieselben Vorteile bringen?

1.4.1 Programmbibliotheken

Ein Standardverfahren zur Aufspaltung, das in praktisch jeder Programmiersprache vorkommt, ist die Aufteilung der Codebasis in mehrere Bibliotheken. Solche Bibliotheken können von Drittherstellern stammen oder in Ihrem Unternehmen erstellt worden sein.

Bibliotheken ermöglichen es, Funktionalitäten mit anderen Entwicklerteams oder anderen Services zu teilen. Ich könnte beispielsweise einen Satz nützlicher Hilfsprogramme zur Handhabung von Objektsammlungen erstellen oder vielleicht eine wiederverwendbare Statistikbibliothek programmieren.

Entwicklerteams können sich entsprechend solcher Bibliotheken organisieren, und die Bibliotheken selbst sind wiederverwendbar. Es gibt aber auch einige Nachteile.

Erstens geht die echte Nutzung verschiedenartiger Technologien verloren, denn die Bibliothek muss typischerweise in derselben Sprache wie der Code programmiert sein, zumindest aber auf derselben Plattform laufen. Zweitens wird die Leichtigkeit beschnitten, mit der Sie Teile Ihres Systems voneinander unabhängig skalieren können. Drittens können Sie keine neue Bibliothek deployen, ohne auch den gesamten Prozess erneut zu deployen, sofern Sie keine dynamisch eingebundenen Bibliotheken verwenden. Dadurch schränken sich Ihre Möglichkeiten ein, isolierte Änderungen vorzunehmen. Das eigentliche Problem ist aber vermutlich, dass es nun keinen naheliegenden Zugang mehr gibt, über den Sie die architektonischen Sicherheitsmaßnahmen (siehe Kapitel 11) zur Aufrechterhaltung der Belastbarkeit des Systems einrichten können.

Dennoch haben Programmbibliotheken ihre Berechtigung. Sie werden für häufig anfallende Aufgaben, die unabhängig von einem bestimmten Geschäftsfeld sind, vermutlich Code erstellen, den Sie im gesamten Unternehmen wiederverwenden möchten – hier liegt es auf der Hand, eine wiederverwendbare Bibliothek einzusetzen. Seien Sie aber dennoch vorsichtig: Gemeinsam genutzter Code, der zur Kommunikation zwischen Services eingesetzt wird, kann zu einer Kopplung führen. Mehr dazu in Kapitel 4.

Services können auf Bibliotheken von Drittherstellern zugreifen und sollten auch regen Gebrauch davon machen, um häufig vorkommenden Code wiederzuverwenden. Die alleinige Lösung ist das aber nicht.

1.4.2 Module

Manche Programmiersprachen stellen eigene Verfahren zur Aufspaltung zur Verfügung, die die Möglichkeiten einfacher Bibliotheken übertreffen. Dabei kann die Lebensdauer solcher Module gesteuert werden, indem sie aktiviert und deaktiviert werden, was es ermöglicht, sie laufenden Prozessen bereitzustellen. Auf diese

Weise können Sie Änderungen vornehmen, ohne den ganzen Prozess beenden zu müssen.

Hier ist insbesondere die *Open Source Gateway Initiative* (OSGI) als Ansatz zur Modularisierung zu nennen. Java an sich kennt noch keine richtigen Module und wir werden mindestens auf die Java-Version 9 warten müssen, bis die Sprache dahingehend erweitert wird. OSGI wurde in Form eines Frameworks entwickelt, das es ermöglicht, in der Java-IDE Eclipse Plug-ins zu installieren und wird nun dazu verwendet, in Java über eine Bibliothek ein Modulkonzept nachzurüsten.

Das Problem ist, dass OSGI Funktionalitäten wie die Steuerung der Lebensdauer von Modulen zu erzwingen versucht, ohne dass die Sprache selbst dies hinreichend unterstützt. Das führt dazu, dass die Modulautoren mehr Arbeit aufwenden müssen, um die Module ordnungsgemäß zu isolieren. Außerdem gerät man innerhalb eines einzelnen Prozesses allzu leicht in Versuchung, die verschiedenen Module zu sehr zu koppeln, was wiederum die verschiedensten Probleme hervorrufen kann. Ich weiß aus eigener Erfahrung, die mit derjenigen von Kollegen aus der Branche übereinstimmt, dass OSGI selbst bei guten Entwicklerteams schnell zu einer Komplexität führen kann, die von den Vorteilen nicht aufgewogen wird.

Die Programmiersprache Erlang verfolgt einen anderen Ansatz, bei dem Module zu einem Bestandteil der Laufzeitumgebung werden. Erlang ist somit eine sehr ausgereifte Verfahrensweise zur Modularisierung. Erlang-Module lassen sich problemlos stoppen, neu starten und aktualisieren. Diese Programmiersprache gestattet es sogar, mehrere Versionen eines Moduls gleichzeitig laufen zu lassen, wodurch es möglich ist, Module auf elegante Weise zu aktualisieren.

Die Fähigkeiten der Erlang-Module sind tatsächlich beeindruckend, aber selbst wenn wir das Glück haben, eine Plattform mit diesen Fähigkeiten zu verwenden, verbleiben immer noch die Unzulänglichkeiten, die wir von herkömmlichen Programmbibliotheken kennen. Unsere Möglichkeiten, neue Technologien einzusetzen, sind sehr beschränkt: Wir können nur in begrenztem Maße unabhängig skalieren, laufen Gefahr, zu stark koppelnde Integrationsverfahren zu verwenden, und es fehlt der Zugang zu architektonischen Sicherheitsmaßnahmen.

Es gibt an dieser Stelle aber noch eine weitere erwähnenswerte Überlegung. Rein technisch sollte es möglich sein, innerhalb eines einzelnen monolithischen Prozesses ordnungsgemäß aufgeteilte, unabhängige Module zu erstellen. Dessen ungeachtet sieht man dergleichen nur selten. Die Module selbst werden schon bald eng mit dem übrigen Code gekoppelt sein und geben damit einen ihrer wichtigsten Vorteile preis. Eine deutliche Trennung der Prozesse zwingt in dieser Hinsicht zu »sauberer« Programmierung (oder erschwert es zumindest, etwas falsch zu machen!). Ich will natürlich nicht behaupten, dass dies der entscheidende Antrieb zur Trennung von Prozessen sein sollte, aber es ist interessant festzustel-