Methoden des Software-Engineerings

4

In Teil 4 diskutieren wir Aspekte der Software-Wiederverwendbarkeit und Methoden des Software-Engineerings. Leser mit Vorkenntnissen bezüglich der Grundlagen des Projektmanagements und der Prinzipien des objektorientierten Entwurfs und der Programmierung sind bei diesem Kapitel im Vorteil.

- Kapitel 4.1, »*Software-Wiederverwertung?*«, argumentiert, dass Software-Wiederverwendbarkeit nicht ein eigenständiges Unternehmensziel ist. Stattdessen muss Wiederverwendbarkeit im Kontext des Software-Engineering und der Qualitätssicherung gesehen werden und sich Zielen wie Flexibilität, Time-to-Market und Kosteneffektivität unterordnen. Verschiedene Entwicklungen in der Softwaretechnologie, insbesondere im Bereich der Programmiersprachen, werden in diesem Kontext betrachtet.
- Kapitel 4.2, »*Eine Wiederverwendungs-Methodologie*«, diskutiert die Software-Wiederverwendung unter besonderer Berücksichtigung von komponentenbasierten Architekturen. Dabei entwickeln wir Richtlinien, um die Wiederverwendung von Komponenten möglichst nutzbringend zu machen.
- Kapitel 4.3, »*Der Software-Entwicklungszyklus*«, führt Software AGs SELC ein, ein inkrementelles und iteratives Lifecycle-Modell, das insbesondere für objektorientierte Electronic Business-Anwendungen geeignet ist.

4.1 Software-Wiederverwertung?

Seit Programmierer Software für Computer schreiben, wenden sie das gleiche Prinzip an wie die Herrscher des Römischen Reiches: *Divide et Impera*. D&I war sogar lange vor der Computerprogrammierung ein Generalprinzip aller Ingenieurskunst: Wenn ein Problem zu komplex ist, um es auf einen Schlag zu lösen, teilt man es in mehrere kleinere Unterprobleme und nimmt sich diese getrennt vor.

Wenden wir dieses Prinzip wiederholt an, so finden wir schnell heraus, dass einige Unterprobleme immer wieder auftauchen, und dass wir anfangen, das Rad neu zu erfinden. Teilen wir z.B. das Problem der Wohnzimmerreinigung in die Einzelprobleme Teppichreinigung und Fensterputzen, so werden wir schnell merken, dass das Fensterputzen im Wohnzimmer dem Fensterputzen in der Küche sehr ähnelt, dass wir hier also ganz ähnliche Prozeduren anwenden können.

Viel mehr gibt es auch über die Wiederverwendung von Software nicht zu berichten. *Software-Reuse* handelt von der Wiederverwendung existierender Entwürfe, Programme und Prozeduren mit dem Ziel, einen Computer ähnliche Vorgänge in *verschiedenen* Kontexten ausführen zu lassen. Das verlangt, dass wiederverwendbare Prozeduren an andere Gegebenheiten anpassbar sind. Um bei unserem Beispiel zu bleiben: Die Prozedur für das Fensterputzen sollte sich an verschiedene Fenstergrößen und -formen anpassen lassen, um eine maximale Wiederverwendbarkeit zu erreichen.

Wiederverwendbarkeit ein geschäftliches Ziel?

Software-Wiederverwendbarkeit ist jedoch kein geschäftliches Ziel an sich, sondern nur das Mittel zum Zweck. Dabei ist die Software-Wiederverwendbarkeit nur ein Mittel unter vielen.

Als geschäftliche Ziele seien genannt:

- Die Neugestaltung von Geschäftsprozessen mit dem Zweck höherer Effektivität im Unternehmen.
- Minimierung der Kosten für Software-Entwicklung und -Wartung.
- Die Bewältigung einer Software-Krise in einem Unternehmen.
- Die Verkürzung des Zeitraums bis zur Markteinführung eines Produktes.
- Die Maximierung von Kundenzufriedenheit und -treue.

Diese Ziele sind oft widersprüchlich. Insbesondere der vierte Punkt kann später viel Ärger bereiten. Zeitdruck führt oft zu *Quick-and-Dirty*-Lösungen, die dann später hohe Wartungskosten nach sich ziehen. Normalerweise schnell implementiert, um ein dringendes Problem zu lösen, entpuppen sich diese »Lösungen« als außerordentlich langlebig. Das Y2K-Problem war teilweise das Ergebnis solcher »Ingenieurskunst«.

Die schnelle Entwicklung des Internets und des Electronic Business haben zu ähnlichen Problemen geführt. Um Unternehmen schnell »auf's Web« zu bekommen, werden Abkürzungen genommen. Ingenieurmäßig sauber konstruierte Webseiten sind die Ausnahme, nicht die Regel.

Software-Engineering für Internet-Anwendungen?

Im Juni 1998 hielt *IEEE Software [Pressman1998]* ein Roundtable-Gespräch ab unter dem Titel *»Can Internet-based Applications Be Engineered?«*.

Allein die Tatsache, dass diese Frage gestellt wurde, lässt vermuten, dass viele Websites sich in einen chaotischen Dschungel bestehend aus Webseiten, CGI Skripts, Java Applets, ActiveX Controls, Active Server Pages, Dynamic HTML Seiten, Plug-ins u.a.m. entwickelt haben. Das Ergebnis sind praktisch unwartbare Sites: Wartungsarbeiten, für die ein paar Stunden angesetzt waren, erstrecken sich mitunter über Wochen.

Bei einer Wachstumsrate von 70% pro Jahr ist es abzusehen, dass Electronic Business-Anwendungen in kurzer Zeit die Mehrzahl aller kommerziellen Anwendungen ausmachen werden. Deshalb ist es heute umso dringender, Ingenieurpraktiken für den Entwurf und die Implementierung von Electronic Business-Applikationen zu diskutieren, wenn wir nicht in ein paar Jahren mit Problemen konfrontiert sein wollen, die das Y2K-Problem eher marginal erscheinen lassen.

Die richtigen Werkzeuge

Wir argumentieren also für einen ingenieurmäßigen Ansatz für Electronic Business-Applikationen. An die benutzten Werkzeuge und Programmiersysteme stellen wir die folgenden Forderungen:

- Applikationen müssen *schnell* erstellt werden können.

  Das heißt nicht notwendigerweise, dass eine neue Applikation schnell kodiert werden muss. Die Erstellung einer neuen Applikation umfasst mehrere Phasen, von denen das Schreiben des Programmcodes nur 10-20% der Gesamtzeit in Anspruch nimmt. Zusätzliche Anstrengungen, die während der frühen Entwurfs- und Implementierungsphasen investiert werden, zahlen sich gewöhnlich während der Test- und Wartungsphasen aus.

  Alle Programme sollten gründlich entworfen und dokumentiert werden. Fehler sollten so früh wie möglich gefunden werden, wenn möglich schon in der Entwurfsphase oder bei der Kompilierung. Das Programmiersystem sollte typische Standardvorgänge bereits als Sprachelemente anbieten, da Sprachelemente im Gegensatz zu Bibliotheksfunktionen vom Compiler geprüft werden können.

- Die erzeugten Applikationen müssen robust sein.

  Konstrukte, die in einer Testumgebung nur schwierig fehlerfrei zu machen sind, sollten vermieden werden. Typische Kandidaten sind hier: Speicherplatzverwaltung auf unterer Ebene, Prozess- und Thread-Synchronisierung auf unterer Ebene und Datenbanktransaktionen auf unterer Ebene.

Gleichermaßen ist es für Electronic Business Applikationen wesentlich, dass nicht nur funktionale Tests durchgeführt werden, sondern auch »Crash«-Tests mit hohem Datenvolumen, um die Skalierbarkeit zu testen.

- Die erzeugten Applikationen müssen flexibel sein.

  Es muss möglich sein, Applikationen rasch an sich ändernde Erfordernisse anzupassen. Dabei sollte es nicht zu Überraschungen kommen.

  Der Programmcode muss leicht zu lesen und einfach zu verstehen sein; die verwendeten Konstrukte sollten intuitiv sein. Die verschiedenen Belange in einer Applikation sollten in verschiedenen Code-Einheiten erscheinen (*Separation of Concerns*). Änderungen an Komponenten sollten keine Fernwirkungen auf andere Komponenten haben.

- Es muss möglich sein, existierende Entwürfe und Komponenten, inklusive der Komponenten von Fremdanbietern und von existierenden (*legacy*) Systemen zu nutzen.

  Programmiersysteme müssen Methoden bereitstellen, externe Komponenten in eine Applikation zu integrieren. Auch muss es möglich sein, Komponenten für die Zwecke einer Applikation anzupassen.

Wenn es so aussieht, als würden wir uns von unserem ursprünglichen Thema (Software-Wiederverwendung) wegbewegen, so stimmt dieser Eindruck. Wir betrachten die Software-Wiederverwendung als zum erweiterten Bereich des Software-Engineering gehörig. Einige der Techniken, die vordem unter dem Banner der Software-Wiederverwendung angepriesen wurden wie z.B. Objektorientierung und Vererbung sind in Wirklichkeit Techniken, deren Bedeutung mehr im Bereich der Qualitätssicherung wie Korrektheit, Stabilität und Flexibilität liegt.

In den letzten Jahre sind komponentenbasierte Softwareentwicklungssysteme Realität geworden. Hier sehen wir die Chancen für eine Software-Wiederverwendung, die sich auch lohnt. Nicht nur ein paar Zeilen Code werden hier wiederverwendet, sondern komplette, schlüsselfertige (*plug-and-play*) und anpassbare Softwarekomponenten.

4.1.1 Einfache Sachen zuerst

Es tut uns fast leid, aber erwähnen müssen wir es: Die Lesbarkeit einer Programmiersprache hat direkte Wirkung auf die Qualität der Software, die damit implementiert wird. Kann der Programmcode leicht gelesen und verstanden werden, so ist es auch leichter in dem Programm Fehler zu finden, es zu ändern und erneut zu benutzen. David Parnas Geheimnisprinzip *[Parnas1972]* bezieht sich auf Modularisierungstechniken – nicht gemeint ist damit die Lesbarkeit des Codes.

In der Tat ist es eine der effektivsten Techniken, um korrekten Code zu erzielen, die Coderevision durch Kollegen (*peer review*), also die Programme noch von einer zweiten Person Korrektur lesen zu lassen. Einige der erfolgreichsten Softwaresysteme beruhen auf offenem Quellcode. Je mehr Leute den Code lesen und verstehen können, umso besser. Tausend Augen sehen mehr als zwei.

Hier sind ein paar Ratschläge für lesbaren Code:

- Prägnanz ist keine Tugend. Nicht in der Programmierkunst.

  COBOL versus APL

  Die erste kommerzielle Programmiersprache war COBOL, eine Sprache, die alles andere als prägnant ist. Anweisungen wie

  ```
  ADD a TO b GIVING c
  ```

  lassen uns heutzutage über so viel Beredsamkeit lächeln, machten aber zur Zeit, als COBOL eingeführt wurde, Sinn. Zu dieser Zeit schrieben Programmierer ihren Code noch auf Papier (mit einem Bleistift!). Der Kodierbögen wurden dann an die Lochkartenabteilung übergeben und dort von Datentypisten auf Lochkarten gestanzt. Die Lochkarten wurden ausgedruckt und vom Programmierer Korrektur gelesen, bevor sie für die Kompilierung freigeben wurden. Bei Intervallen von Stunden oder Tagen zwischen Kompilierungen waren Tippfehler fatal. Eine Programmiersprache wie COBOL, die alles in volle Worte fasste, war leichter zu tippen und leichter Korrektur zu lesen, was das Risiko von Tippfehlern gering hält.

  Dann kam APL. APL erschien zu einer Zeit als auch Time-Sharing-Systeme aktuell wurden. Programmierer bekamen eigene Endgeräte wie die IBM-Kugelkopfschreibmaschine, ein feinmechanisches Wunderwerk. APL konnte auf solchen Time-Sharing-Systemen laufen und war auch eine der ersten interpretativen Sprachen. Eine Kompilierung war überflüssig, man konnte das Programm eintippen, sofort ausführen und augenblicklich die Ergebnisse. Welch ein Fortschritt!

  Allerdings gab es ein kleines Problem. Die meisten Programmierer und Wissenschaftler konnten nicht tippen! Die extreme Prägnanz von APL (gewöhnlich ein Zeichen für eine Funktion) machte es populär – da gab es schließlich nicht viel zu tippen. Und da APL auch noch griechische Zeichen verwendete (man brauchte einen speziellen Kugelkopf), hätte eine trainierte Datentypistin auch ihre Schwierigkeiten gehabt. So aber sah man mit dem Zweifingersystem nicht allzu dämlich aus.

  Die Folge war, dass APL-Programme sehr schwierig zu lesen waren – sogar für den Autor – und APL errang schnell den Ruf einer WOL (*Write Only Language*). Der Sprache gelang es nie, größere Bereiche in der kommerziellen Programmierung zu erobern, trotz der Anstrengungen von IBM in den frühen Siebzigern, APL als kommerzielle Programmierspra-

che zu vermarkten. APL ist heute immer noch in Gebrauch, aber seine Bedeutung für die Entwicklung geschäftskritischer Applikationen ist gleich Null. Inzwischen sind Programmierer auch etwas flüssiger im Umgang mit der Tastatur geworden – die extreme Prägnanz von APL wird deshalb auch nicht mehr als so überaus 'cool' angesehen.

- Intuitive Syntax

Gleichheit und Zuweisung

Eine Programmiersprache sollte Operatoren und Kommandos so verwenden, wie man es erwartet. Bei einem Elektroherd z.B. wird ja auch nicht die höchste Stufe mit »0« und Aus-Stellung mit »3« gekennzeichnet.

Nehmen wir als Beispiel die Zuweisung:

Aus der Schule wissen wir alle, dass »=« der Vergleichsoperator ist. Wir wissen auch, dass mit `a = b` gemeint ist, dass die Inhalte der Variablen a und b gleich sind. Wir wissen auch, dass wenn `a = b` gilt, so gilt umgekehrt `b = a`.

Dann erschien FORTRAN. FORTRAN verwendete das Gleichheitszeichen für eine ganz andere Operation. In FORTRAN bedeutet `a = b`: Weise den Inhalt der Variablen `b` der Variablen `a` zu, wobei der frühere Inhalt von `a` zerstört wird. Offensichtlich ist die Bedeutung von `a = b` hier völlig verschieden von der Bedeutung von `b = a`. Gar nicht gut. FORTRAN hatte damit allerdings eine Tradition begonnen. C, C++ und Java benutzen alle das Gleichheitszeichen für die destruktive Zuweisung.

COBOL geht in dieser Beziehung überhaupt kein Risiko ein, indem es laut und deutlich sagt was Sache ist: `MOVE b TO a`. ALGOL drückte die verschiedenen Semantiken von Gleichheit und Zuweisung dadurch aus, dass `a := b` für die Zuweisung verwendet wird, und dieser Tradition folgen Pascal, Delphi, Modula, Eiffel, Natural und Bolero. Die asymmetrische Form des Operators macht auch klar, dass die Operation selbst nicht symmetrisch ist.

Operatorenüberfrachtung

Operatorenüberfrachtung *(Operator overloading)* war und ist Gegenstand einer hitzigen Debatte in der objektorientierten Welt. Es ist aber auch ein oft missverstandener Term. Operatorenüberfrachtung bezieht sich eben nicht nur auf Operatoren, sondern generell auf Methoden. (Der Begriff *Operatorenüberfrachtung* hat historische Gründe.) Operatorenüberfrachtung bedeutet, dass der gleiche Methodenname für verschiedene Methodenimplementierungen verwendet werden kann, also ein Name für verschiedene Semantiken steht.

So adressieren die beiden Methodenaufrufe

```
a.add(b)  // b Integer
a.add(c)  // c BigDecimal
```

verschiedene Implementierungen und könnten sehr verschiedene Semantiken haben. In diesem Beispiel sind allerdings die Methoden auch formal verschieden. (Methoden werden durch Signaturen identifiziert, bestehend aus Methodennamen und den Typen der Parameter.)

Zusätzlich hängt die tatsächlich ausgeführte Methode auch noch vom aktuellen *Inhalt* der Variablen `a` ab. So kann z.B. das Feld `a` einen *Subtyp* des deklarierten, statischen Typs enthalten. Wurde bei der Definition dieses Subtyps die Methode `add()` überschrieben, so wird die überschreibende Methode des Subtyps ausgeführt und nicht die Originalmethode des statischen Typs von `a`.

Bei den Argumenten auf der rechten Seite (`b` und `c`) hängt die Methodenauswahl *nicht* vom Inhalt der Variablen ab, sondern ausschließlich von der statischen Definition der Variablen.

Das klingt vielleicht etwas verwirrend, und das ist auch der Grund warum Operatorenüberfrachtung in OO-Kreisen kontrovers diskutiert wird. Operatorenüberfrachtung verlangt hohe Disziplin vom Programmierer, Methoden mit gleichen Namen auch nur mit Implementierungen zu versehen, die intuitiv das gleiche bedeuten, auch wenn die konkreten Semantiken unterschiedlich sind. So ist es durchaus sinnvoll, Methoden, die zu Kollektionen etwas hinzufügen, mit `add` zu bezeichnen, obwohl die Implementierung für Listen, Bäume, Mengen und Arrays völlig unterschiedlich aussehen kann. Sehr verwirrend wäre es allerdings, Methoden, die etwas aus einer Kollektion entnehmen, mit `add` zu bezeichnen.

Operatorenüberfrachtung ist nicht sinnvoll bei Sprachen, die eine Mehrfachvererbung implementieren wie C++ oder Eiffel. In diesen Sprachen können Methoden von einer Vielzahl von Vorfahren ererbt werden, was in Zusammenhang mit der Operatorenüberfrachtung recht unübersichtlich werden kann.

In Sprachen mit Einfachvererbung dagegen wie in Java oder Bolero, halten wir Operatorenüberfrachtung für nützlich – den gleichen Namen für gleichartige Methoden zu verwenden, macht Programme verständlicher.

Infix-Operatoren

Benutzerdefinierbare Infix-Operatoren sind ebenfalls nützlich, um die Lesbarkeit von Programmen zu verbessern, besonders in objektorientierten Sprachen. Warum sollten wir gezwungen werden, gleichartige Funktionen mit verschiedenen Notationen auszudrücken?

So wird die Addition in Java mit '+' ausgedrückt, wenn sie sich auf `int` Variablen bezieht:

```
a + b//  a is Integer
```

Wollen wir dagegen Zahlen vom Typ `BigDecimal` addieren, so ist das '+ ' nicht möglich, da `BigDecimal` eine Bibliotheksklasse ist:

```
a.add(b)//  a is BigDecimal
```

Um die Verwirrung komplett zu machen, lässt sich das '+' hingegen wieder dazu verwenden, um zwei Zeichenketten miteinander zu verketten:

```
a + b //   a is String
```

Eigentlich würden wir das '+' gern in all diesen Fällen gebrauchen. Aus diesem Grund erlaubt es Bolero, Infix-Operatoren generell als Methodennamen zu verwenden wie in Bibliotheksklassen und benutzerdefinierte Klassen.

C-Syntax

C wurde entwickelt, um ein Betriebssystem zu implementieren, nämlich UNIX. C wurde aus Effizienzgründen möglichst nahe an der Hardware entworfen. So gibt es spezielle Operatoren wie z.B. << oder ++, die sich direkt auf elementare Hardware-Operationen wie *shift* und *increment* beziehen. So gelang es C, Assembler aus vielen Systemanwendungen zu verdrängen und so den Weg für plattformunabhängiges Programmieren zu ebnen. C war allerdings nicht für das kommerzielle Programmieren entworfen worden.

Java adoptierte die etwas kryptische Syntax von C. Aus gutem Grunde: Es gab eine riesige Zielgruppe frustrierter C++-Programmierer, die für Java gewonnen werden sollten. So sieht z.B. die `for`-Anweisung in Java genauso aus wie in C:

```
for (int i=0; i<8; i++) ...
```

Das mehr an der Zielgruppe der kommerziellen Programmierer orientierte Bolero spricht dagegen Klartext:

```
for i type Integer in 0..7 do
                ...
end for
```

Wir bemerken hier auch einen Unterschied in der Semantik der `for`-Kontrollstruktur: Während C und Java mittels eines Algorithmus alle möglichen Werte der Variable beschreiben (mit 0 initialisieren; fortsetzen solange der Wert kleiner als 8 ist, nach jedem Durchlauf um 1 inkrementieren), benutzt Bolero ein aktives Objekt (in diesem Falle vom Typ `Range`), dass selbsttätig durch alle möglichen Werte iteriert. So können in einer `for`-Schleife auch andere Objekte verwendet werden, die einen `Iterator` implementieren wie z.B. Arrays, Kollektionen oder die Ergebnisse von Datenbankabfragen.

- Klare Kennzeichnung von Kontrollstrukturen

  Sprachen wie C, C++ und Java verfügen eigentlich über explizite Kontrollstrukturen. Stattdessen gibt es Kontrollanweisungen wie `if` oder `for`, die in Kombination mit einer weiteren Anweisung oder einem Anweisungsblock verwendet werden. Anweisungsblöcke sind dabei neutral und werden von neutralen Begrenzungszeichen { } umschlossen.

  Geschweifte Klammern oder Schlüsselworte?

  In komplexen Programmroutinen führt dies oft zu einer Abfolge von Begrenzungszeichen wie z.B.:

```
            }
         }
      }
```

  Der Nachteil ist, dass wir den geschweiften Klammern nicht ansehen, wie weit der Bereich (*scope*) einer Kontrollanweisung denn reicht. Programmierer helfen sich dabei oft mit Kommentaren:

```
            }  // if
         }  // for
      }  // method
```

  Hier halten wir Programmiersprachen, die echte Kontrollstrukturen bieten, für eleganter und sicherer. Dabei erwarten wir, dass die Endbegrenzung auch anzeigt, welche Kontrollstruktur denn da geschlossen wird. (Diese Technik wird z.B. auch in XML angewandt.) So kann der Compiler die Korrektheit der Kontrollstrukturen prüfen. Hier folgt ein entsprechendes Codesegment in Bolero:

```
method m1

         ....

if  a < b then
      for i type Integer in 1..10 do
         ....
      end for
   end if

end method m1
```

4.1.2 Teile und herrsche

So sehr Ingenieure sich auch über dieses Prinzip einig sind, so sehr sind sie unterschiedlicher Auffassung, wie es am besten in die Praxis umzusetzen sei. In der kurzen Geschichte der Informatik haben sich viele verschiedene Strategien für die Implementierung von *Divide et Impera* entwickelt. Wir zählen hier nur die wichtigsten davon auf:

Prozedurales versus funktionales Programmieren

- *Prozedurales Programmieren* zerlegt den informationsverarbeitenden Prozess in mehrere Schritte, Prozeduren genannt. Meist teilen Prozeduren Datenbereiche mit anderen Prozeduren. Diese Datenbereiche dienen als Zwischenspeicher und zum Zwecke der Kommunikation mit anderen Prozeduren. Prozeduren sind so voneinander abhängig. Das macht es nicht nur schwierig, Prozeduren in anderen Zusammenhängen wiederzuverwenden, sondern auch eine gegebene Prozedur zu ändern, ohne die andern Prozeduren dabei zu beeinflussen. Deshalb haben sich in der Geschichte des Prozeduralen Programmierens bestimmte Prinzipien wie Datenkapselung (*Data Encapsulation*) und das Geheimnisprinzip (*Information Hiding*) *[Parnas1972]* herausgebildet. Diese Prinzipien haben schließlich zur objektorientierten Programmierung geführt.
- *Funktionales Programmieren* betrachtet ein gesamtes Computerprogramm als eine einzige mathematische Funktion. Diese Funktion wird dann in kleinere Unterfunktionen zerlegt, bis die Ebene der primitiver, vom System bereitgestellter Funktionen erreicht ist. Funktionen sind zustandslos, d.h., sie benötigen keine internen Datenbereiche. Folglich können sie keine Datenbereiche überschreiben. Daten treten in der funktionalen Programmierung nur in Form von Parametern und Ergebnissen auf. Das erlaubt es, Funktionen auch leicht in anderen Zusammenhängen wiederzuverwenden, und die resultierenden Programme sind sehr robust. In der kommerziellen Programmierung konnte funktionales Programmieren allerdings kaum Terrain erobern, vermutlich weil das Konzept der zustandslosen Funktion für die datengetriebenen Programme der kommerziellen Programmierung zu fremdartig ist.

  Allerdings haben einige Konzepte, die im Rahmen der funktionalen Programmierung entwickelt wurde wie *generische Datentypen*, auch Eingang in moderne objektorientierte Sprachen gefunden.

4.1.3 Der objektorientierte Ansatz

Objektorientiertes Programmierung hat sich aus der prozeduralen Programmierung entwickelt. Dabei werden Prozeduren, die sich einen bestimmten Datenbereich teilen, zu einem Objekt zusammengefasst. Die gemeinsame Datenmenge definiert dabei den Zustand des Objektes, die Prozeduren, die nun *Methoden* heißen, definieren das Verhalten des Objektes. Da so konstruierte Objekte ein ähnliches Reiz-/Reaktionsverhalten (*stimulus/response behavior*) wie der Rest des Tierreichs entwickeln, sind sie recht intuitiv. So können wir in der kommerziellen Welt Geschäftsobjekte wie Kunden, Produkte, Aufträge und Rechnungen als Software-Objekte abbilden, die wir *Business Objects* (Geschäftsobjekte) nennen. Objekte lassen sich gut wiederverwenden, da sie klar definierte Schnittstellen haben und nicht von ihrer Umgebung abhängen.

Kommando und Abfrage

Eine übliche Metapher für ein Objekt in der objektorientierten Programmierung ist die der *Maschine [Meyer1997]*. Dabei wird der interne Zustand der Maschine von *Kommando*-Methoden verändert, während *Abfrage*-Methoden den Zustand der Maschine ermitteln können. Abfrage-Methoden sollten den Zustand eines Objekts nicht verändern, d.h., die Abfrage sollte keine Seiteneffekte haben.

Der Grund für die Trennung der Methoden in Kommando-Methoden und Abfrage-Methoden liegt in der besseren Wiederverwendbarkeit der Objektklassen. Außerdem wird die Klassenspezifikation lesbarer, und es ist sogar möglich, Beweisverfahren zur Programmverifikation anzuwenden. Datenbankmanagementsysteme verwenden übrigens den gleichen Ansatz. Bei den Datenbankzugriffsmethoden wird strikt zwischen Kommandos und Abfragen unterschieden: *Kommandos* modifizieren die Daten, während *Abfragen* Daten von der Datenbank abfordern, jedoch nicht verändern.

Vererbung

In einer großen Applikation werden wir feststellen, dass viele Objektklassen gemeinsame Merkmale mit anderen Objektklassen haben. Objektorientierte Programmiersprachen erlauben daher die Definition von Klassenhierarchien. Die Klassen an der Spitze der Hierarchie enthalten die allgemeinsten Merkmale. Subklassen ererben diese Merkmale und können die Klassendefinition um eigene Merkmale erweitern oder ererbte Merkmale überschreiben.

Ältere objektorientierte Sprachen wie *Simula* oder *Smalltalk* unterstützen hier die Einfachvererbung: Jede Klasse kann nur eine Elternklasse haben. Das wurde oft als nicht ausreichend angesehen: Klassen haben oft mehrere Facetten, d.h., ihr Typ kann als eine Kombination mehrerer unterschiedlicher Eigenschaften gesehen werden. Z.B. könnte eine Klasse die Typen `Printable`, `Observable` und `Serializable` implementieren.

Mehrfachvererbung?

Neuere Sprache so wie C++ oder Eiffel implementieren deshalb die Mehrfachvererbung: Eine Klasse kann Merkmale von mehreren Elternklassen erben. Besonders in C++ kommt es zu Problemen, wenn Methodenimplementierungen von mehreren Elternklassen ererbt werden. Bei der Mehrfachvererbung besteht grundsätzlich das Problem des Namenskonfliktes. Der Programmierer muss dann entscheiden, wie der Konflikt aufgelöst werden soll: Soll die Methode einer bestimmten Elternklasse vorgezogen werden, sollen Methoden umbenannt werden, oder sollen Methoden gleichen Namens kombiniert werden, und wenn ja, in welcher Reihenfolge? So werden zusätzliche Kopplungen zwischen Elternklassen und Kindklassen eingeführt, was Applikationen komplexer werden lässt, deren Änderung also schwieriger macht *[Sakkinen1988]*.

Klasse und Typ

Java und damit auch *Bolero* haben aus dieser Erfahrung gelernt. In beiden Sprachen wird das Konzept von Klasse (der Implementierung) und Typ (der Schnittstelle) getrennt. Bei Vererbung zwischen Klassen ist nur die Einfachvererbung erlaubt, jedoch können Klassen Schnittstelleneigen-

schaften von mehreren Schnittstellendefinitionen (*interface*) erben. Auch zwischen den Schnittstellendefinitionen ist eine Mehrfachvererbung möglich.

Das bedeutet, dass eine Klasse nur eine einzige Implementierung pro Methode erben kann, dass andererseits aber die Klasse ein Subtyp mehrerer Elterntypen sein kann, wodurch der gewünschte Polymorphismus gegeben ist. Diese Lösung behält wesentliche Vorteile der Mehrfachvererbung bei, vermeidet jedoch auf der anderen Seite eine zu enge Kopplung zwischen Eltern- und Kindklassen.

Wiederverwendung durch Vererbung

In der Vergangenheit wurde die objektorientierte Vererbung oft als wesentliches Konzept für die Softwarewiederverwendung angepriesen. Inzwischen hat sich herausgestellt, dass eine übermäßige Verwendung der Vererbung sich kontraproduktiv auswirkt: Die entstehenden komplexen Vererbungshierarchien verstoßen direkt gegen das Prinzip »*Divide et Impera*«. Die Lesbarkeit der Programme leidet, der Begriff »Jo-Jo-Effekt » (die Klassenhierarchie auf und ab wandern, um so zu verstehen, um was es eigentlich geht) hat hier seinen Ursprung *[Taenzer1989]*. Also geben wir den Rat – trotz gegenteiliger Meinung in manchen Veröffentlichungen zum Thema objektorientiertes Programmieren – Vererbungspfade möglichst kurz zu halten.

Die Prinzipien der Vererbung, wie sie im objektorientierten Programmieren verwendet werden, wurden von dem taxonomischen System des schwedischen Biologen Carl Linné (1701-1778) übernommen. Allerdings hatte Linné sein System zur Klassifizierung von existierendem Material verwendet, nicht als ein Hilfsmittel für Konstruktionsaufgaben. Bertrand Meyer [Meyer1997] gibt in seinem Buch einen schönen Drei-Seiten-Überblick über die Geschichte der Taxonomie – und natürlich auch über alle anderen Fragen objektorientierter Programmierkunst auf den restlichen 1254 Seiten.

4.1.4 Sichere Software

Typsicherheit

Fast jede Programmiersprache hat einen Mechanismus, um die Kompatibilität der an einer Operation beteiligten Datentypen zu prüfen oder nur bestimmte Daten als Parameter in einem Funktionsaufruf zuzulassen. Besonders in objektorientierten Sprachen, wo jede Klasse und jede Schnittstelle einen neuen Datentyp einführt, ist Typsicherheit besonders geboten:

Dynamische und statische Datentypen

- Sprachen wie Smalltalk führen den Quellcode einer Klasse direkt über einen Interpreter zur Ausführung. Ein Kompilierungsschritt in einen Zwischencode oder Maschinencode entfällt. Die Konsequenz ist, dass erst bei der Ausführung auf Typkompatibilität hin geprüft werden kann. Folglich braucht in solchen Sprachen auch gar nicht erst der Typ

einer Variable deklariert zu werden, es ist der Inhalt der Variable, der zählt. Das ist nun genau das, was wir unter dynamischen Datentypen (*dynamic typing*) verstehen. Die Typsicherheit in solchen Sprachen ist niedrig, da Inkompatibilitäten erst in der Testphase gefunden werden können.

- Sprachen wie Eiffel, C++, Java oder Bolero kompilieren Programme, bevor sie ausgeführt werden können. Das erlaubt es dem Compiler, die Kompatibilität von Datentypen zu überprüfen. Alle diese Sprachen sehen deshalb vor, dass der Typ von Variablen im Programm deklariert wird. Statische Datentypen (*static typing*) machen solche Prüfungen möglich.

Type casting

Nicht in allen Fällen ist es möglich, der strikten Typalgebra der statischen Datentypen zu genügen. Insbesondere, wenn Daten von externen Quellen wie Dateien und Datenbanken bezogen werden, wenn Daten mit externen Modulen wie DCOM-Komponenten ausgetauscht werden oder wenn wir generische Konstrukte wie Kollektionen (*siehe Kapitel 3.4.2*) verwenden, ist es oft notwendig, Inhalte Variablen zuzuweisen, die *formal* nicht kompatibel miteinander sind. Es sind deshalb Typumwandlungen (*type casting*) nötig, die die Typüberprüfung durch den Compiler außer Kraft setzen und so ein Element der Unsicherheit in die Programmierung einführen.

Generische Typen

Eine typsichere Programmiersprache sollte deshalb mit möglichst wenigen Typumwandlungen auskommen. Eine anerkannte Strategie dabei ist, generische Datentypen zu verwenden (*siehe Kapitel 3.4.2*). Während generische Typen die erforderliche Flexibilität für Kollektionen und ähnliche Konstrukte bieten, erlauben sie trotzdem noch die Typüberprüfung durch den Compiler, was Typumwandlung unnötig macht.

Automatische Typabbildung

Eine andere Strategie zur Reduktion von Typfehlern ist es, die Zugriffsmethoden auf externe Datenquellen und Komponenten in die Sprache zu integrieren. Das wurde zuerst von den Sprachen der vierten Generation (4GL) praktiziert, die Datenbankzugriffsmethoden auf Sprachebene abbildeten. Bolero geht den gleichen Weg: Mit der *objektrelationalen Abbildung* und der *Relational-Objekt-Abbildung* werden die Strukturen und Datentypen der Datenbank automatisch auf Bolero-Klassen und Bolero-Datentypen abgebildet (*siehe Kapitel 3.4.6*). Die bei JDBC-Datenbankzugriffen nötige Typumwandlung entfällt hier, was wiederum einige Fehlermöglichkeiten ausschließt.

Das gilt auch für den Import von DCOM- und CORBA-Komponenten oder von CICS-Transaktionen in das *Bolero Component Studio*. Datentypen von diesen Komponentenmodellen werden automatisch auf Bolero-Datentypen abgebildet, eine explizite Typumwandlung durch den Programmierer entfällt *(siehe Kapitel 3.4.4 und 3.4.12)*.

Semantische Sicherheit

Softwarekonstrukte verlassen sich oft auf gewisse Annahmen über den Kontext in dem sie eingesetzt werden. Das kann die Wiederverwendung dieser Konstrukte gefährlich machen, wenn die Randbedingungen plötzlich nicht mehr stimmen:

Bertrand Meyer [Meyer1997], der Kontraktprogrammierung im Rahmen von Eiffel erstmals in die objektorientierte Programmierung einführte, erzählt die Geschichte der Ariadne5-Rakete, die kurz nach ihrem Start infolge eines Softwarefehlers explodierte. Eine numerische Routine verursachte eine nicht vorhergesehene und deshalb auch nicht abgefangene Ausnahmebedingung. Die Programmautoren gingen bei der Implementierung von durchaus korrekten Randbedingungen aus, jedoch galten diese nur für Ariadne4. Bei der Wiederverwendung für Ariadne5 waren die Randbedingungen jedoch andere.

Kontrakte Kontraktprogrammierung (*Programming by Contract*) (*siehe Kapitel 3.4.2*) macht eine Software-Einheit nicht generischer oder wiederverwendungsfähiger, vielmehr macht sie wiederverwendungsfähige Software-Einheiten sicherer. Wird eine Softwarekomponente in einem anderen Kontext wiederverwendet, so könnte sie Parameterwerte erhalten, die beim Entwurf der Komponente nicht vorgesehen waren. Der Kontrakt stellt nun sicher, dass eine Ausnahmebedingung erzeugt wird, bevor eine unvorhergesehene Wertekombination Schaden verursacht.

Die Formulierung eines Kontraktes hat dreierlei Wirkungen:

- Der Programmierer wird sich über die Randbedingungen der jeweiligen Softwareeinheit klar.
- Der Kontrakt dokumentiert späteren Benutzern der Softwareeinheit, welche Randbedingungen einzuhalten sind und welche Ergebnisse garantiert werden.
- Eventuelle Fehler bei der Verletzung von Randbedingungen erzeugen sofort eine Ausnahmebedingung und nicht erst dann, wenn falsche Werte über eine lange Wirkungskette zum Absturz des Programms führen. Durch die frühe Meldung von Fehlern wird denn auch die Fehleranalyse erleichtert, und die Wahrscheinlichkeit, dass Fehler bereits in der Testphase gefunden werden, wächst.

4.1.5 »Separation of Concerns«

In *Kapitel 2.5* hatten wir bereits Djikstras berühmte These der *Separation of Concerns* (Trennung von Belangen) *[Hürsch1994]* im Zusammenhang mit *Aspect Oriented Programming* (AOP) diskutiert.

Softwaresysteme sind oft mit einer Vielzahl von Belangen befasst. Dazu zählen Persistenz, Transaktionskontrolle, Fehlerbehandlung, Nebenläufigkeit, Einsatzort (welches Objekt lokal ist und welches nicht), Performanz,

Robustheit, Protokollierung, Internationalisierung, Präsentation an der Benutzerschnittstelle u.a. m.

Ein Beispiel ist der Programmcode von *Kapitel 3.4.2*:

```
class Produkt
 is public and is persistence capable with population

instance field Name type String is public

instance field Preis type BigDecimal is public
  contract
    precondition newValue >= 0
    postcondition result >= 0
  end contract
  value 0

instance method berechneMwSt is public
  parameter Steuersatz type Decimal
  result type BigDecimal
  throws MwStFehler, SteuersatzFehler, PreisFehler
  contract
    precondition Steuersatz >= 0
                           else throw SteuersatzFehler()
    precondition Preis >= 0 else throw PreisFehler()
    postcondition result >= 0 else throw MwStFehler()
  end contract

  implementation
   result := Preis*Steuersatz/100
  end implementation

end method berechneMwSt
end class Produkt
```

Abbildung 4.1 Code für Kontrakte gemischt mit Geschäftslogik.

Die eigentliche Geschäftlogik diese Beispiels passt in eine einzige Zeile:

```
result := Preis*Steuersatz/100
```

der Rest ist Code, der nicht direkt auf die Geschäftlogik bezogen ist, sondern Kontrakte und Methodenparameter definiert o.Ä.

Oft vermischt sich dieser zusätzliche Code mit dem eigentlichen Code, der die Geschäftslogik des Programms implementiert. Das erschwert es, den eigentlichen Zweck eines Programms unmittelbar zu erkennen, und erschwert das Lesen der Programme, deren Änderung, Wiederverwertung und deren Wartung *[Lopez1997]*.

»Separation of Concerns« fordert nun, dass die einzelnen Aspekte eines Programms nicht die Geschäftlogik zersplittern und verdecken sollen, sondern möglichst separat gehandhabt werden sollen. Auf der Ebene von Software-Entwurfstechniken ist »Separation of Concerns« ein anerkanntes Prinzip und wird gern praktiziert. Auf der Ebene der Implementierung fehlten bis jetzt die Programmiersprachen, die es erlauben, bestimmte Aspekte eines Programms separat zu formulieren. Allerdings gibt es einige interessante Neuentwicklungen, die entlang des Prinzips der aspektorientierten Programmierung entworfen wurden so z.B. *AspectJ* von Xerox *[Kiczales1997a]*. Die verschiedenen Aspekte eines Softwarekonstrukts werden hier in getrennten Programmsektionen formuliert. Diese Aspekte werden später zu einem Programm zusammenge”webt«. Das kann durch einen Programmgenerator, durch einen Präprozessor oder durch einen Compiler geschehen.

Business Class

Das klingt eigentlich nicht nach objektorientierter Programmierung, oder? Das objektorientierte Paradigma fordert doch, dass alle zu einem Objekt gehörigen Funktionen auch Teil der Objektdefinition sein sollen. Aber es gibt Objekte und Objekte, so wie es Leute und Leute gibt. Da gibt es den berühmten kleinen Mann, der alles selbst erledigen muss: Termine ausmachen, das Auto reparieren und am Wochenende den Rasen mähen. Und dann gibt es den reichen Geschäftsmann: die Sekretärin ist für die Termine zuständig, um das Auto kümmert sich der Chauffeur und der Rasen wird vom Gärtner gepflegt. Der gute Mann kann sich also ohne Ablenkung voll ums Geschäft kümmern.

Mit den Objekten ist es genau dasselbe. Das Durchschnittsobjekt muss sich um alles selbst kümmern: um Fehlerbehandlung, Transaktionskontrolle und die Koordination mit anderen Objekten. Das Geschäftsobjekt dagegen wird umsorgt und kann sich voll auf seine Geschäftslogik konzentrieren. Eine Klassengesellschaft? Aber ja!

Als kommerzielles Programmiersystem unterstützt Bolero »Separation of Concerns« auf zwei Ebenen:

- Zunächst schlägt Bolero eine grundlegende Methodologie bei der Implementierung kommerzieller Applikationen vor:

  Die Triade von Geschäftsobjekt, Geschäftsvorgang und Geschäftsprozess (abgebildet durch persistente Objekte, transaktionskontrollierende Objekte und Lange Transaktionen) *(siehe auch Kapitel 3.4.9)* trennt schon einige der Belange voneinander:

Einheit	Belange
Geschäftsobjekt (persistenzfähige Klasse)	Geschäftslogik Persistenz Objekt-relationale Abbildung
Geschäftsvorgang (transaktionskontrollierende Klasse)	Kontrolle von Datenbanktransaktionen Beziehungen zwischen Geschäftsobjekten
Geschäftsprozess (Lange Transaktion)	Koordination zwischen Geschäftsvorgängen Stornieren Wiederanlauf Protokollierung

Abbildung 4.2 »Separation of Concerns« in Bolero.

- Außerdem präsentiert das *Bolero Component Studio* die verschiedenen Aspekte der Programmlogik auf verschiedenen Seiten eines Notizbuches, wobei jedes Notizbuch alle Definitionen von Software-Einheiten wie Klassen, Schnittstellen, Adapter, Methoden, Felder usw. enthält.

  So werden beispielsweise auf der Ebene der Klassen Aspekte wie Objektpersistenz und Transaktionskontrolle, Objekt-relationale Abbildung, Eigenschaften für den Einsatz als DCOM- oder EJB-Komponente auf getrennten Seiten präsentiert und so aus dem Programmcode herausgehalten.

Abbildung 4.3 Blatt für die allgemeinen Eigenschaften der Klasse `Produkt`

- Auf der Ebene der Methoden und Felder gibt es eine ähnliche Trennung der Aspekte. Parameter, Kontrakte, Ausnahmebedingungen u.a. werden vom Code ferngehalten. So ist die eigentliche Geschäftslogik der Methoden einfacher zu erkennen und zu verstehen. Programme, die gut zu verstehen sind, können auch leichter an andere Zwecke angepasst

werden. So ist es möglich, bestimmte Aspekte wie Transaktionskontrolle oder Komponentenmodelle zu ändern, ohne die eigentliche Geschäftslogik auch nur anzurühren.

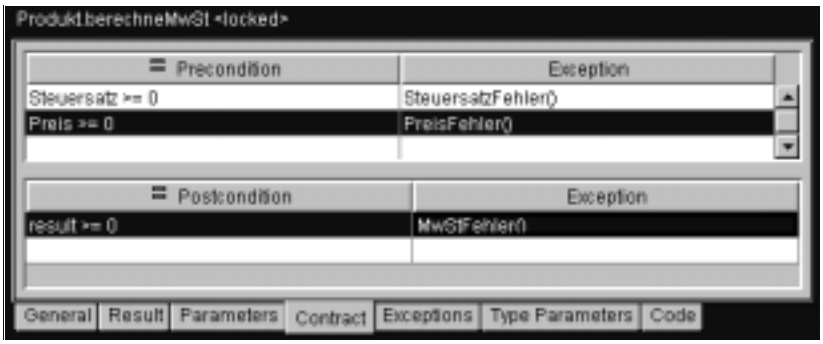

Abbildung 4.4 Kontrakte-Blatt für die `berechneMwSt`-Methode der Klasse `Produkt`. Hier werden vor und Nachbedingungen der Methode definiert.

Abbildung 4.5 Code-Blatt für die `berechneMwSt`-Methode der Klasse `Produkt`. Die Geschäftslogik könnte kaum deutlicher sein.

4.1.6 Fehlerbehandlung

Wenn die Geschäftlogik die Regel ist, was ist dann die Ausnahme?

Ausnahmebedingungen (*exceptions*) sind Situation, die mit den normalen Programmstrukturen der Geschäftslogik nicht behandelt werden können:

- Hardwarefehler: kein Plattenplatz, unterbrochene Datenübertragung.
- Netzwerkfehler: falsche URL, keine Antwort vom Server.
- Betriebssystemfehler: zu viele Fenster geöffnet, nicht genug Ressourcen, Dateilesefehler.
- Programmfehler: Fehler im Code, fehlende Module, Versionsfehler, Typfehler.
- Absichtlich von einer Applikation herbeigeführte Ausnahmebedingungen wie:
  - Ausführung einer `throw`-Anweisung
  - von einem Kontrakt erzeugte Ausnahmebedingung

- von einer Langen Transaktion erzeugte Ausnahmebedingung, z.B. wenn eine *Event Reaction Condition* eine ungültige Situation entdeckt.

Dagegen sind vom *Endbenutzer* falsch eingegebene Daten nicht die Ausnahme, sondern die Regel. Die Validierung von Benutzereingaben obliegt der normalen Programmlogik und sollte keine Ausnahmebedingungen hervorrufen.

Zwiebelhäute

In den meisten modernen Programmiersprachen folgt das Modell für die Behandlung von Ausnahmebedingungen dem mit ADA eingeführten Modell der Zwiebelhäute (*onion skin model*): Wird eine Ausnahmebedingung in einem inneren Programmblock erzeugt, versucht sie nach außen zu gelangen. Auf jeder Ebene können *Exception Handler* versuchen, die Ausnahmebedingung abzufangen. Ausnahmebedingungen, die es bis nach draußen schaffen, bringen die Applikation zum Absturz.

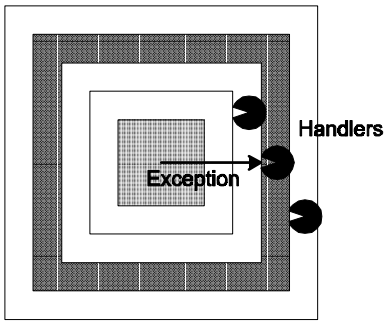

Abbildung 4.6
Das Zwiebelhautmodell für das Exception Handling.

Bürger erster Klasse

In den meisten objektorientierten Sprachen sind Ausnahmebedingungen vollgültige Objekte (*first-class citizens*), normalerweise Subtypen der Klasse `Exception`. Die Objekte können zusätzliche Daten mit detaillierter Information über Grund, Ort und Kontext der Ausnahmebedingung mit sich führen und damit die Diagnose der Fehlerursache erleichtern. Das ist eine beträchtliche Erleichterung im Vergleich zu nicht-objektorientierten Programmiersprachen, wo oft nur eine schlichte Fehlernummer zurückgegeben wird.

In Java und Bolero wird eine saubere Fehlerbehandlung vom Compiler erzwungen. Ein Programmblock, der eine Ausnahmebedingung erzeugt oder eine Methode aufruft, die dies tut, muss entweder auch die entsprechende Fehlerbehandlung bereitstellen oder der Block muss deklarieren, dass er die entsprechende Ausnahmebedingung erzeugt. Das ermöglicht es dem Compiler, durch alle »Zwiebelhäute« hindurch die Fehlerbehandlung zu prüfen und das Entschlüpfen einer Ausnahmebedingung nach draußen wirksam zu verhindern.

In Bolero steht dabei die Fehlerbehandlung immer am Ende eines Blockes:

Abbildung 4.7 Exception Handling in Bolero.

```
begin
   connect("DataBase", "User", "Password")
...
   something.dangerous()
...

on exception e
   case IOException
       System.out.println("IO error")

   case OverflowException
       System.out.println("Overflow")

   default
       System.out.println("Exception" + e)
end exception

finally
      disconnect
end finally
end
```

In diesem Beispiel prüft die `on exception`-Klausel den dynamischen Typ der Ausnahmebedingung `e`. Der `default`-Fall wird ausgeführt, wenn sonst kein Fall zutrifft. Die `finally`-Klausel wird immer ausgeführt und kann dazu benutzt werden, um Aufräumarbeiten, die beim Verlassen des Blokkes notwendig sind, durchzuführen.

4.1.7 Entwurfsmuster

Design Patterns

Innovationen entstehen äußerst selten in der Isolation, besonders beim Schreiben von Software. Programmierer sehen sich gewöhnlich an, wie eine ähnliche Aufgabe von anderen gelöst wurde, bevor sie daran gehen, die Erfordernisse für eine neue Anwendung oder eine Implementierungsstrategie festzulegen. »*Make it look like a Macintosh*«, so wird berichtet, war das Entwurfsziel, dass schließlich zum Windows Betriebssystem führte.

Das heißt keineswegs, dass eine existierende Lösung einfach kopiert wird. Die bestehende Lösung dient lediglich als eine Art Vorgabe und Orientierung. Im Verlauf des Entwurfs und der Implementierung werden dann Merkmale der Referenzlösung akzeptiert, modifiziert oder verworfen, oder es werden neue Merkmale hinzugefügt. Während des Prozesses entwi-

ckeln sich dann oft innovative Lösungen, die nicht aufgetaucht wären, hätte es keinen Diskurs mit dem Stand der Kunst gegeben.

Grafische Designer z.B. benutzen ganz ähnliche Methoden. Bei einem neuen Projekt ist es eine der ersten Maßnahmen, Kataloge mit Vorlagen schon existierender, erfolgreicher und preisgekrönter Designs durchzublättern. Auf der Basis dieser Designs oder von Ideen, die beim Durchblättern entstehen, werden dann neue Designs geschaffen.

Auch bei Architekten ist das ähnlich. So gibt einen vitalen Markt für Architekturmagazine und -bücher, der Architekten über die Werke der Meister, neue Trends und Problemlösungen informiert.

Ursprung der Design Patterns

Es sollte denn auch ein Architekt sein, der diesen Prozess der Innovation in eine formale Methode abbildete:

Das Konzept der *Design Patterns* (Entwurfsmuster) wurde zuerst von dem Architekten Christopher Alexander formalisiert. Alexander kannte sich auch in der Mathematik aus und veröffentlichte in den späten 1970ern über Stadtplanung und Stadtarchitektur.

In ihrer ursprünglichen Bedeutung beschreiben Design Patterns die *Beziehung* zwischen einem *Problem*, dem *Kontext* des Problems und der *Lösung* des Problems. Dabei wird diese Beziehung derart beschrieben, dass es möglich wird, den Lösungsweg auf andere Kontexte zu übertragen. Design Patterns werden nicht erfunden: Sie werden entdeckt, wenn ähnlich Lösungen für ähnliche Probleme in verschiedenen Kontexten existieren.

Anders als objektorientierte Vererbung, bei der Wissen vertikal übergeben wird (von Elternklasse auf Kindklasse), erfolgt der Wissenstransfer mit Design Patterns horizontal: Die neue Lösung (B), die mit Hilfe der Lösung (A) gefunden wurde, ist nach dem Transfer (A) keineswegs unterstellt, sondern ist unabhängig.

Im Bereich des Software-Engineerings werden Design Patterns seit den späten Achtzigern in der Entwurfsphase angewandt. Bahnbrechend war hier die Arbeit der »Viererbande« bestehend aus Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides *[Gamma1995]*. Von da an wurden Patterns auch im Bereich der Software-Entwicklung populär.

Design Patterns erklärt

Sehen wir uns zum Beispiel die Beziehung zwischen den Geschäftsobjekten Kunde und Auftrag an: `Customer:Order`. Wir könnten die Beziehung zwischen diesen beiden Klassen `Customer` und `Order` mit einem Feld in der Klasse `Order` implementieren, das den Eigentümer des Auftrags referenziert (`Customer`).

Nach einigen weiteren Erfahrungen finden wir heraus, dass es noch einige weitere Probleme gibt, die ganz ähnliche Lösungen erfordern wie z.B. die Beziehung zwischen Lieferant und Rechnung: `supplier:invoice` oder zwischen Abteilung und Mitarbeiter: `department:employee`.

Diesen Problemen ist gemeinsam, dass in allen Fällen eine *1:n*-Beziehung (`OneMany`) vorliegt.

Um nun ein Design Pattern zu formulieren, müssen wir von den jeweiligen konkreten Szenarien abstrahieren. Anstatt von einer konkreten *1:n*-Beziehung wie `customer:order` zu sprechen, verwenden wir verallgemeinerte Rollenbezeichnungen: `roleB:roleA`. So können wir das Problem und den Lösungsweg in allgemeiner Form, basierend auf `roleB` und `roleA` formulieren.

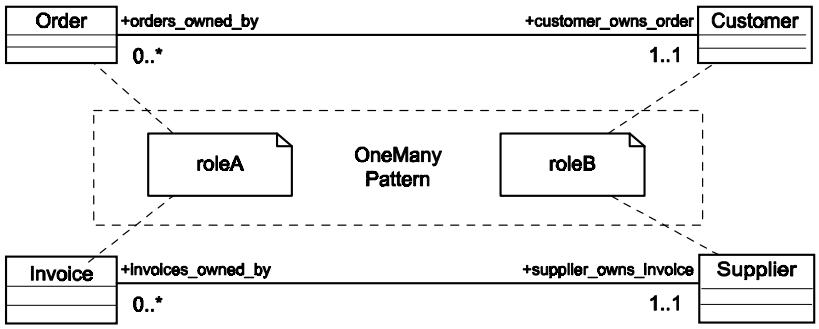

Abbildung 4.8 Übertragung einer Lösung mithilfe eines Pattern.

Nachdem das neue `OneMany`-Entwurfsmuster nun so verallgemeinert wurde, können wir es auf konkrete Szenarien anwenden und dabei die allgemeinen Rollen mit konkreten Klassen wie `supplier` und `invoice` instanziieren. Das Pattern wird uns dann eine konkrete Lösung für das konkrete Problem liefern.

Design Patterns in Bolero

In *Bolero* werden Design Patterns im *Bolero Component Studio* unterstützt. Dabei erfolgt die Anwendung eines Patterns auf ein konkretes Problem automatisch und die so erzeugte Lösung kann sofort ausgeführt und getestet werden.

Bolero Design Patterns bestehen aus:

- Einer *Pattern-Klasse* so z.B. aus der `OneMany` Pattern-Klasse, die bereits in Boleros Pattern-Bibliothek vorhanden ist. Die abstrakten Rollen (in unserem Fall `roleA` und `roleB`) sind Eigenschaften (öffentliche Felder) dieser Klasse.
- Einem Ursprungsprojekt (*source*), das aus allen Klassen besteht, die den Kontext der Ausgangslösung (*Source Context*) festlegen wie in unserem Beispiel die Klassen `Customer` und `Order`. Dazu kommen noch alle Klassen, Felder und Methoden, die zu der Lösung dieses spezifischen Problems gehören, in unserem Falle des `Customer:Order`-Problems.

- Einer Instanz der Pattern-Klasse (in unserem Beispiel der `OneMany`-Klasse), die wir *Source Pattern Instance* nennen. Diese Instanz enthält die Beziehung zwischen Ursprungskontext und abstraktem Pattern. So sind bei dieser Instanz die Felder, die die abstrakten Rollen implementieren, mit den jeweils konkreten Rollen des Ursprungskontextes instanziiert. In unserem Beispiel gilt: `Order` ⇔ `roleA`, `Customer` ⇔ `roleB`.
- Einem Zielprojekt (*Target*), das die Klassen enthält, die die Rollen des Zielkontextes repräsentieren, hier `Supplier` und `Invoice`. Es müssen hier nur die Klassendefinitionen für diese Rollen präsent sein. Diese Klassendefinition können leer sein, da Felder, Methoden und Hilfsklassen bei der Anwendung des Patterns automatisch generiert werden.

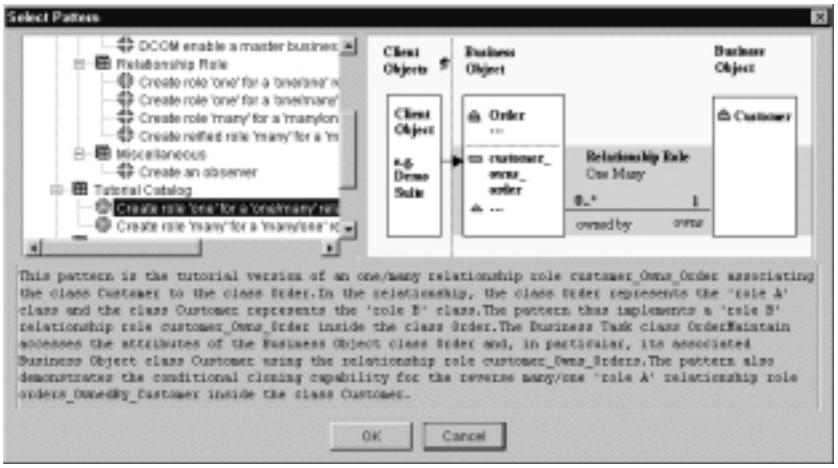

Abbildung 4.9 Design Patterns in Bolero. Vordefinierte Patterns übertragen bereits vorhandene Lösungen auf neue Szenarien.

Projekte klonen

Mit diesen Zutaten ist es in drei Schritten möglich, die Lösung vom Ursprungskontext auf den Zielkontext zu übertragen:

- Im ersten Schritt wird eine *Target Pattern Instance* angelegt, in unserem Fall eine zweite Instanz der `OneMany`-Pattern-Klasse, nur dass hier die Rollen des *Zielkontextes* den abstrakten Rollen zugeordnet werden: `Invoice` ⇔ `roleA`, `Supplier` ⇔ `roleB`.
- Unter der Kontrolle dieser beiden Pattern-Instanzen wird die Lösung, die im Ursprungskontext vorhanden ist, auf den Zielkontext *geklont*. Das Pattern agiert im Rahmen diese Klonens als Transportmechanismus, der alle Klassendefinitionen, Felder und Methoden vom Ursprungsprojekt zum Zielprojekt kopiert und dem Zielkontext entsprechend anpasst.
- Die resultierenden Klassen im Zielprojekt können sofort kompiliert und getestet werden. Wenn das Ergebnis nicht den Erwartungen entspricht, kann das Klonen rückgängig gemacht werden, d.h., das Zielprojekt wird in den alten Zustand zurückgesetzt.

Der Transfer einer vorhandenen Lösung auf einen neuen Kontext ist natürlich nur der erste Schritt in einem Konstruktionsprozess. Am erzeugten Zielprojekt können anschließend Änderungen vorgenommen werden. (Spätere Änderungen am Ursprungsprojekt haben keinen Einfluss auf Zielprojekte.) Sowohl das Ursprungsprojekt als auch das modifizierte Zielprojekt können wiederum Ausgangspunkte neuer Pattern-Anwendungen werden.

Es ist wichtig, sich klar zu machen, dass nicht das Pattern eine Lösung für ein Problem bereitstellt. Die Lösung ist im Ursprungsprojekt enthalten. Das Pattern dient nur als Transfermechanismus.

Die Erzeugung von Zielprojekten mit Hilfe von Patterns sollte wie jede Entwurfs- und Implementierungsentscheidung dokumentiert werden.

Patterns können in verschiedener Granularität angewandt werden. Es ist möglich, Patterns auf sehr kleine Probleme (wie oben) anzuwenden, aber auch auf große Probleme mit Hunderten von Klassen, bis hin zum *Application Framework*. Hätten wir z.B. in unserem Beispiel die Geschäftsobjekte `Order` und `Customer` noch mit grafischen Benutzeroberflächen ausgestattet, so hätte das Pattern äquivalente Benutzeroberflächen für `Invoice` und `Supplier` mitgeneriert. Obendrein lassen sich Patterns auch noch schachteln, aber hier verlässt uns die Vorstellungskraft.

Standard-Patterns

Bolero enthält bereits eine Bibliothek mit vordefinierten Design Patterns:

- Patterns für Beziehungen wie *One-one*, *One-many*, *Many-One*.
- Patterns für Geschäftsvorgänge wie *Single-platform-elementary-object*, *Single-platform-aggregate-object*, *Distributed-platform-elementary-object*.
- Das Publisher-Subscriber-Pattern.

4.1.8 Komponenten

»Objekte eignen sich gut für die Wiederverwendung, da sie über klar definierte Schnittstellen verfügen und nicht von ihrem Kontext abhängen.« So oder ähnlich schrieben wir, als wir die objektorientierte Programmierung diskutierten, und so wurde auch die objektorientierte Programmierung angepriesen.

Daran ist sicherlich wahr, dass Objekte klar definierte Schnittstellen haben und dass sie nicht unter dem Hauptproblem prozeduraler Programmierung leiden, nämlich dem gegenseitigen Überschreiben von Variablen in gemeinsamen Speicherbereichen.

Erfahrungen mit der Wiederverwendung von Software

Die in den letzten Jahren gemachten Erfahrungen mit einer großen Zahl von kommerziellen Softwareprojekten zeigen, dass man diesem Anspruch jedoch nicht gerecht wird. Zwar sind Objekte unabhängig von ihrem Kontext, jedoch nur in einem physischen Sinn (bei der Variablenverwendung).

Auf semantischer Ebene, insbesondere in ihrem Verhalten, hängen die meisten Objekte sehr wohl von ihrem Umfeld ab, weil sie mit anderen Objekten in diesem Umfeld interagieren, also in ihrem Verhalten auf das der anderen Objekte abgestimmt sein müssen. Außerhalb dieses Umfelds ist ihre Brauchbarkeit begrenzt.

Selbstverständlich gibt es Objekte, die von allgemeinem Interesse und Nutzen sind. Die meisten dieser Objekte sind allerdings bereits in *Software Development Kits* (SDK) und Standardbibliotheken enthalten. In Sachen Wiederverwendbarkeit haben objektorientierte Techniken inzwischen teilweise einen Sättigungspunkt erreicht.

Also wurde es Zeit für ein neues Konzept in Sachen Softwarewiederverwendbarkeit. Das neue Konzept heißt Komponenten (*siehe Kapitel 2.6*). Komponenten sind per Definition wiederverwendbar:

Komponenten definiert

> *»Eine Komponente ist ein Typ, eine Klasse oder irgend ein anderes Arbeitsprodukt, das spezifisch für die Wiederverwendung konstruiert wurde.« [Jacobson1997]*

Der Möglichkeiten, Komponenten wiederzuverwenden, sind viele. So gibt es viele Geschäftsfunktionen, die von Sparte zu Sparte identisch oder sehr ähnlich sind, deshalb immer wieder in Electronic Business Applikationen auftauchen. Zum Beispiel werden Kunden, die Dienste eines Unternehmens anfordern, immer wieder mit den gleichen Fragen konfrontiert: Name? Geburtsdatum? Telefonnummer? E-Mail-Adresse? usw. In einem Unternehmen werden derartige Funktionen von den verschiedensten Applikationen immer wieder implementiert. Hier gemeinsame, wiederverwendbare Komponenten zu identifizieren, kann beträchtliche Vorteile für die Erstellung und Wartung von Applikationen mit sich bringen.

Vorteile von Komponenten

Der Einsatz von Komponententechnologie hat unter anderem die folgenden Vorteile:

- *Mehr Produktivität für den Entwickler*. Vorgefertigte und vorgetestete wiederverwendbare Komponenten können vom Entwickler einfach in die Anwendung eingeklinkt (plug-in) werden und reduzieren so die nötige Anstrengung und Zeit um neue Applikationen zu entwickeln und existierende Applikationen zu warten.
- *Konsistente und akkurate Verarbeitung*. Dies wird dadurch erreicht, dass sich nur eine Softwareinstanz um eine gegebene Funktion kümmert, auch wenn diese Funktion in verschiedenen Anwendungen auftaucht.
- *Einfacheres Austesten*. Wurde eine Komponente gründlich getestet, so wird für gewöhnlich keine extensive Testphase benötigt, wenn die Komponente in einem anderen Kontext (einer anderen Applikation) eingesetzt wird, vorausgesetzt, ihre Schnittstellen wurden entsprechend abgesichert (*siehe Kontrakte, Kapitel 4.1.4 und 3.4.2*).

Um die angesprochenen Vorteile zu erzielen, ist eine geeignete Wiederverwendungsstrategie erforderlich:

- Verwendung einer Wiederverwendungsmethodologie (*siehe Kapitel 4.2*), die konsistent von allen Anwendungsprogrammierern angewandt wird.
- Ein Expertengremium, dessen Aufgabe es ist, die Wiederverwendung von Komponenten durch die Begutachtung von Projekten zu fördern und bei der Einführung von Wiederverwendungstechniken zu assistieren.
- Dokumentation für jede Komponente. Dies schließt die exakte Definition der Eingabe- und Ausgabeparameter für jede Komponentenschnittstelle mit ein.
- Eine Bibliothek oder Repositorium, das Informationen über wiederverwendbare Komponenten enthält.
- Integrierte Fehler- und Ausnahmebehandlung, die jede Komponente in die Lage versetzen, unabhängig von anderen Komponenten und Applikationen zu agieren.

Es wäre höchst nachlässig von uns, würden wir im Zusammenhang von Komponententechnologie die Frage existierender (*legacy*) Applikationen vergessen. Diese Applikationen repräsentieren oft den Wert von zehn bis zwanzig Jahren fortgesetzten Investments. Ernsthafte Softwarewiederverwendung heißt deshalb auch, eine Strategie zu entwickeln, um existierende Applikationsfunktionen im Kontext von Electronic Business wiederzuverwenden.

Application Mining

Die Technik, wiederverwendbare Komponenten innerhalb existierender Applikationen zu finden, wird auch als »harvesting« oder »application mining« bezeichnet.

Wie im echten Bergbau gibt es dabei zwei Schritte:

- Eine **Explorationsphase**, in der die existierenden Applikationen untersucht werden, mit dem Ziel bestehende Geschäftsfunktionen und -objekte als eigenständige Komponenten zu isolieren.
- Eine **Ausbeutungsphase**, in der die ausgewählten Geschäftfunktionen und -objekte als Objektklassen bereitgestellt werden. Die hier verwendeten Techniken heißen »wrapping« oder »Einkapselung«. Dazu wird Code geschrieben (oder automatisch erzeugt), der existierende Geschäftsfunktion oder -objekte mit einer Programmierschnittstelle (API) umgibt.

4.2 Eine Wiederverwendungs-Methodologie

Die erfolgreiche Implementierung einer Multi-Tier-Architektur mit wiederverwendbaren Komponenten hängt nicht ausschließlich von der Fähigkeit ab, wiederverwendungsfähige Komponenten zu erstellen. Der Erfolg hängt auch von der Bereitstellung geeigneter Werkzeuge und von einem entsprechenden Management für die Wiederverwendung von Komponenten ab.

Datenmodell des Unternehmens

Im Zusammenhang mit der Komponentenwiederverwendung haben Unternehmen, die über ein unternehmensweites Datenmodell verfügen, einen Vorteil. Unternehmensweite Daten sind Daten, die sowohl für einzelne Geschäfteinheiten zur Verfügung stehen, jedoch auch über das Gesamtunternehmen hinweg zwischen mehreren Geschäftseinheiten oder Tochtergesellschaften geteilt und ausgetauscht werden. Applikationscode, der auf unternehmensweite Daten zugreift oder diese pflegt, sollte ebenfalls in allen Abteilungen wiederverwendet und gemeinsam von verschiedenen Einheiten genutzt werden. Gibt es dagegen mehrere Programminstanzen, die auf die gleiche Datenstruktur zugreifen, so besteht immer das Risiko, dass die verschiedenen Programminstanzen sich bei der Änderungen der Daten verschieden voneinander verhalten, und so die Integrität der Daten verletzen können.

Strategie für Wiederverwendung

Das Schlüsselelement für Erfolg im Wiederverwendungsgeschäft ist eine solide Strategie. Wird eine komponentenbasierte Architektur nicht explizit im Hinblick auf Wiederverwendung entworfen und aktiv betreut, so wird im Ergebnis die Softwareentwicklung nicht erleichtert, sondern erschwert.

Die Schlüsselelemente eines Wiederverwendungsprogramms sind:

- Bestandsaufnahme
- Katalog
- Wiederverwendungsadministrator
- Methodologie
- Entwurfsrichtlinien und -prinzipien
- Bewertungsverfahren
- Qualitätssicherung
- Leistungsanreize

Programme für die Wiederverwendung sollten die Wiederverwendungsmethodologie unternehmensweit installieren. Dabei sollte eine Wiederverwendungsmethodologie mit dem Systementwicklungszyklus integriert sein.

Ein Expertenteam (*component review board*) sollte Projekte begutachten, beim »harvesting« von Komponenten aus Altanwendungen (*siehe Kapitel 4.1.8*) und bei der Implementierung von Komponenten assistieren. Die Experten sollten dabei aus wichtigen Benutzergruppen aus allen Unternehmensbereichen kommen. Die Aufgabe des Expertensystems ist es, das Programm für die Komponentenwiederverwendung im ganzen Unternehmen anzuwenden. Damit das Programm Erfolg hat, ist es erforderlich, dass die Mitglieder des Expertenteams die nötige Autorität haben, um die Definition wiederverwendbarer Komponenten aushandeln zu können *[Jacobson 1997]*.

4.2.1 Techniken für die Komponentenwiederverwendung

Softwarewiederverwendung ist wirklich kein neues Thema. Applikationsprogrammierer haben immer schon Code wiederverwendet. Wiederverwendungstechniken bei komponentenbasierter Architektur baut auf den bekannten Techniken auf:

- **Copycode.** Hier gibt es zwei Spielarten für die Wiederverwendung von Quellcode:

  1. Das Kopieren von Quellcode direkt von einem Programm in den Code eines anderen Programms (*cut&paste*).
  2. Die Einbettung ganzer Programmdateien und Codesegmente mithilfe von INCLUDE-Anweisungen und von *Copybooks* in ein Programm.

  Allerdings entsteht bei diesem Prozess ein Code, der schwierig an neue Aufgaben anzupassen und deshalb teuer in der Wartung ist. Implementiert ein Code-Segment eine bestimmte fachliche Regel (*Business Rule*), und ändert sich diese Regel, so muss die Änderung in allen Programmen, die dieses Codesegment importiert haben, vorgenommen werden. Mindestens müssen bei der Verwendung von INCLUDE-Anweisungen und *Copybooks* alle diese Programme neu kompiliert und getestet werden.

- **Link Libraries.** Das Binden von Programmen aus vorkompilierten Bibliotheksmodulen wird entweder statisch nach dem Kompilieren durchgeführt oder beim Laden eines Programms dynamisch kurz vor der Ausführung mit Hilfe von *Dynamic Link Libraries* (DLL). Diese Methode der Codewiederverwendung ist besser, als Quellcode von einem Programm ins andere zu kopieren, da hier die implementierte Fachlogik physisch nur einmal existiert. Allerdings muss, wenn sich ein Modul in der Bibliothek ändert, jedes Programm, das dieses Modul verwendet, identifiziert, ggf. neu gebunden und neu getestet werden.

- **Service Request.** Hier wird das benötigte Dienstprogramm nicht als Modul in die Applikation eingebunden, sondern die jeweilige Dienstfunktion wird durch das Versenden einer Nachricht von einem lokalen oder entfernten Server abgefordert. So verwenden z.B. die Dienste von Betriebssystemen diese Methode. Der Vorteil dieser Methode ist, dass die Applikation vom Dienstprogramm unabhängig ist. Beide können in unterschiedlichen Adressräumen oder sogar auf verschiedenen Maschinen ablaufen.

Service Requests sind heute die bevorzugte Methode für den Funktionsaufruf bei wiederverwendetem Code und sind die empfohlene Technik für komponentenbasierte Architekturen. Diese Methode unterstützt auch *Multi-Tier*-Architekturen wie sie für kommerzielle Applikationen empfohlen werden (*siehe Kapitel 3.4.5*).

4.2.2 Von Komponenten bereitgestellte Dienste

Wiederverwendbare Komponenten können in die folgenden Kategorien eingeordnet werden:

- **Applikationsdienste.** Diese Komponenten umfassen Geschäftsobjekte (*Business Objects*), Geschäftsvorgänge (*Business Tasks*) und Geschäftsprozesse (*Business Processes*).
- **Dienste für die Benutzerschnittstelle.** Diese Dienste umfassen Navigationsfunktionen, Datenansichten (*views*), Funktionen für die Darstellung von Daten und Funktionen für die Interaktion mit dem Benutzer. Ein solcher Dienst könnte z.B. bestehende Altanwendung webfähig machen, indem die ursprünglichen 3270-Oberfläche durch eine HTML-Oberfläche ersetzt wird. Das Ziel ist hier, das System so zu implementieren, dass es keinen Unterschied macht von welcher Benutzerschnittstelle aus das Anwendungsprogramm betrieben wird: Der Informationsfluss in der Altanwendung bleibt identisch.

  Typische Benutzerschnittstellen umfassen:

  - Grafische Benuterzeroberflächen (GUI)
  - Grün-schwarze Bildschirme (z.B. UNIX- oder 3270-Endgeräte)
  - Web-Browser
  - Point-of-sales Geräte (z.B. Kassen)
  - Mobile Geräte (WAP)
  - Sprachein- und -ausgabe wie das gute alte Telefon
- **Unterstützende Dienste.** Dies sind Dienste, die betriebssystemsartige Funktionen bieten wie z.B. Drucken, Faxen oder Bildverarbeitung. Normalerweise werden diese Dienste fertig als Pakete gekauft. Dabei ist darauf zu achten, dass sich die Dienste gut in eine *Multi-Tier*-Umgebung integrieren lassen.

- **Kerndienste.** Diese Komponenten stellen die grundlegende IT-Infrastruktur in einem Unternehmen bereit. Dazu gehören Sicherheitsdienste (*security*), Namens- und Verzeichnisdienste sowie Nachrichtentransportdienste. Diese Dienste werden normalerweise in Form gekaufter Middleware bereitgestellt.

4.2.3 Richtlinien für Komponentensysteme

Hier sind einige Richtlinien, die beim Entwurf oder Kauf von Komponenten, die für anpassbare, verteilte Multi-Tier-Anwendungen geeignet sind, helfen sollen:

- Das Ziel der komponentenbasierten Architektur ist die Verbesserung des geschäftlichen Erfolgs. Eine komponentenbasierte Entwicklungsstrategie ermöglicht adaptive Systeme, die sich den wechselnden Erfordernissen des Geschäftslebens und des sich ständig ändernden technologischen Umfelds leicht anpassen können. Eine komponentenbasierte Entwicklungsstrategie hilft, die Informationstechnologie mit den fachlichen Anforderungen besser zu synchronisieren.

Komponenten über das ganz Unternehmen wiederverwenden

- Die Komponentenarchitektur erlaubt es, Komponenten über das ganze Unternehmen hinweg wiederzuverwenden. Wiederbenutzbare Komponenten verbessern die Produktivität der Anwendungsentwicklung in den einzelnen Entwicklungsabteilungen eines Unternehmens. Die gemeinsame Nutzung bestimmter Komponenten erhöht die Fähigkeit des Gesamtsystems, sich an ändernde Erfordernisse anzupassen.

  In Applikationsfamilien statt in Einzelapplikationen zu denken, erhöht dabei die Wiederverwendbarkeit von Komponenten *[Parnas1972]*.

  Wiederverwendbare Komponenten müssen aus jeder Anwendung heraus aufrufbar sein. Die Wiederverwendung dieser Komponenten eliminiert Dopplungen bei der Entwicklung, beim Testen und der Wartung. Die Wiederverwendung von Komponenten eliminiert Inkonsistenzen in der Informationsverarbeitung, da fachliche Regeln (*Business Rules*) nur einmal implementiert werden müssen. So entfallen möglicherweise voneinander abweichende Mehrfachimplementierungen. Die Zeit für die Entwicklung und Wartung von Applikationen wird damit verkürzt.

Sprachneutral und plattformunabhängig

- Komponenten sollten so entworfen und implementiert werden, dass der aufrufende Prozess nicht an eine bestimmte Programmiersprache oder Umgebung gebunden ist.

  Neue Komponenten sollten plattformunabhängig implementiert werden, so dass Komponenten auf jeder unterstützten Plattform eingesetzt werden können. Eine Komponente sollte von jeder unterstützten Programmiersprache und von jeder unterstützten Plattform aus aufrufbar

sein. Wenn sich die fachlichen Anforderungen ändern und ggf. eine neue Computerplattform zum Einsatz kommen soll, können die Komponenten problemlos auf die neue Plattform übertragen werden.

Kaufen oder selber machen?

- Wenn möglich, sollten Komponenten nicht selbst implementiert, sondern gekauft werden. Gekaufte Komponenten sollten in der Lage sein, in einer serverbasierten Multi-Tier-Architektur abzulaufen, d.h., sie sollten über eine angemessene Programmierschnittstelle (API) verfügen. Auch Entwicklungskomponenten wie Klassenbibliotheken können gekauft werden. Das erlaubt es den Anwendungsprogrammierern, sich auf die Implementierung der fachlichen Regeln zu konzentrieren.

  Der nächste Schritt: In Zukunft werden größere Komponenten wie Datenbanken oder Office-Produkte in vielen Fällen nicht mehr gekauft werden, sondern entsprechende Dienstleistungen werden vom Internet-Provider oder vom Betreiber eines Extranets gegen eine Gebühr angeboten. Gerade für Internet-Provider, die durch die fallenden Kommunikationspreise unter Druck geraten, bieten sich derartige *value-added services* an. *[McNealy1999]*

Repositorium

- Ein Repositorium sollte alle Information über vorhandene wiederverwendbare Komponenten enthalten. Das Repositorium sollte den Anwendungsprogrammierern zugänglich gemacht werden und ein wichtiges Werkzeug für ihre Arbeit bilden. Im Repositorium wird auch die Dokumentation für die Programmierschnittstellen (APIs) der Komponenten gespeichert.

Qualität planen

- Komponenten sollten so entworfen sein, dass sie vollständig selbstgenügsam (*self contained*) sind. Funktionen für die Validierung, für die Erkennung und Behandlung von Ausnahmebedingungen, Berichtsfunktionen, Protokollierung, Diagnose und Fehlerbehebung, Überwachungsfunktionen, Warnfunktionen und Funktion für die Systemverwaltung müssen von jeder Komponente bereitgestellt werden, um den Betrieb, die Administration und die Wartung der Komponente zu unterstützen.

  Eine Komponente sollte eine einzelne fachliche Regel (*Business Rule*), eine Funktion oder eine kleine Menge aufeinander bezogener Regeln und Funktionen wie z.B. einen Geschäftsvorgang implementieren. Maximale Wiederverwendungsfähigkeit wird erreicht, wenn jede Komponente nur eine einzige Regel oder Funktion implementiert.

  Es sollten Richtlinien für die Optimierung der Performanz aufgestellt werden. Richtlinien für die Länge von Nachrichten bei Anfragen und Antworten dienen dazu, unnötigen Netzwerkverkehr und damit Performanzprobleme zu vermeiden.

  Jede Komponente muss mindestens eine veröffentlichte Programmierschnittstelle (API) besitzen. Jedes veröffentlichte API definiert eine Ein-/Ausgabe-Schnittstelle für eine Komponente oder einen Dienst. Die Do-

kumentation sollte die Ein- und Ausgabeparameter vollständig wiedergeben: welche Parameter benötigt werden, welche Parameter optional sind sowie Typen und Längen von Parametern. Das API sollte dem Komponentenrepositorium beigegeben werden, das jedem Entwickler zugänglich ist.

Wiederbenutzbare Testsuiten sollten für jede Komponente entwickelt werden. Eine Testsuite enthält spezielle Programme, die für den Aufruf einer Komponente benötigt werden, darüber hinaus Eingabedaten, die für die Tests benötigt werden, und vorgegebene Ausgabedaten, mit denen die Testergebnisse verglichen werden können. Die Testsuiten werden wie jede andere wiederverwendungsfähige Komponente gewartet und gepflegt.

Methodologie

Wiederverwendungsmethodologien für die Identifizierung und Implementierung von wiederverwendungsfähigen Komponenten sollten eingesetzt werden. Dazu gehören effektive Methodologien für die Verwaltung von Komponenten, inklusive der Werkzeuge für Komponentenwiederverwendung. Gerade in einer verteilten Umgebung muss es eine Methodologie geben, mit der die vorhandenen Komponenten über Plattformen hinweg verwaltet werden können. Die Methodologie muss die notwendigen Schritte enthalten, um wiederverwendbare Komponenten identifizieren, definieren und entwickeln zu können. Wird eine solche Methodologie nicht eingesetzt, ist es sehr schwierig, die Wiederverwendung von Komponenten effektiv zu organisieren.

Eine Wiederverwendungsmethodologie besteht aus folgenden Schritten:

1. Die fachlichen Anforderungen (*Business Requirements*) nach Dienstkategorie (Anwendung, Benutzeroberfläche, Unterstützung, Kerndienste) klassifizieren.
2. Das Repositorium nach wiederverwendbaren Komponenten absuchen, die die gegebenen fachlichen oder funktionalen Anforderung abdecken.
3. Die möglichen Kandidaten daraufhin prüfen, ob sie der Anforderung vollständig genügen.
4. Die ausgewählten Komponenten in eine neue oder überarbeitete Applikation unter Benutzung von Standard-Programmierschnittstellen (API) einbringen.
5. Komponenten aus existierenden Applikation herauslösen (*harvest*). Altapplikationen sind gute Quellen für den Aufbau eines Komponentenrepositoriums. Sogenannte *Legacy*-Applikationen bestehen oft aus reifer, robuster und effizienter Software (freilich gibt es auch überalterte und unwartbare Exemplare!). Auf jeden Fall besteht hier

nicht die Notwendigkeit, das Rad neu zu erfinden und eine schon existierende Funktionalität noch einmal neu zu implementieren. Wenn möglich, sollte die Altfunktion als Komponente eingepackt, also mit einem API versehen werden, das zu dem jeweiligen Dienst eine Programmierschnittstelle definiert. Damit werden Altapplikationen ohne größere Änderungen und Aufwand zu wiederverwendbaren Komponenten.

Die Wiederverwendungsmethodologie sollte in den Softwareentwicklungszyklus integriert werden (*siehe Kapitel 4.3*).

Eigentümer benennen

- Komponenten sollten designierte Eigentümer und Verantwortliche für die Wartung haben. Die Verantwortung liegt dabei bei dem Team, das die Komponente im Rahmen der Anwendungsentwicklung erstellt hat, oder bei einem solchen, das sich auf Komponentenentwicklung spezialisiert hat. Oft gibt es verschiedene Teams für verschiedene Komponentenkategorien. Die *fachliche* Verantwortung für die Definition einer Komponente sollte bei der Geschäftseinheit liegen, die auch mit der entsprechenden fachlichen Funktion betraut ist.

Expertenteam

- Die Aufstellung eines Expertenteams für Komponentenwiederverwendung (*component review board*) ist nötig, um gemeinsam zu nutzende Komponenten zu identifizieren. Komponenten, die von mehreren Geschäftseinheiten verwendet werden, müssen auch von allen Benutzern verstanden und referenziert werden. Die Komponentenentwicklung kann dabei im Rahmen von Projektarbeiten betrieben werden. Das Expertenteam sollte mit kleinen, durchführbaren und extrem strategischen Projekten beginnen. Um wiederverwendbare Komponenten zu erstellen, ist eine Zusammenarbeit zwischen den Eigentümern der jeweiligen Geschäftsprozesse unabdinmgbar. Dazu wird ein gewisser Organisationsrahmen benötigt:
  - Zentralisierte Verwaltung von wiederverwendbaren Komponenten, die sich für die gemeinsame Nutzung eignen.
  - Begutachtung der Entwürfe von neuen und existierenden Projekten, um derartige Komponenten zu identifizieren.
  - Zugang zu den Information über wiederverwendbare Komponenten auf Unternehmensebene.

  Der Entwurf von Komponenten sollte in allen laufenden Projekten regelmäßig begutachtet werden. Dabei muss bestimmt werden, ob die fachlichen Anforderungen von den existierenden Komponenten abgedeckt werden. Falls nicht, muss entschieden werden, ob bestehende Komponenten erweitert werden können, um die Anforderungen abzudecken, allerdings ohne dabei die Wiederverwendbarkeit der Komponente zu gefährden.

4.3 Der Software-Entwicklungszyklus

Die Informationstechnologie scheint sich mit exponentiell ansteigender Rate zu verändern. Produktentwicklungszyklen haben sich zunächst von 10 auf 5 Jahre reduziert, dann auf 1 Jahr, auf 6 Monate und werden nun in »Web-Zeit« gemessen – 3 Monate bis 6 Wochen. Die Folge davon ist, dass die »best practices« von heute zu den schlimmsten Albträume von morgen werden können.

Fachliche Architektur versus Technische Architektur

Organisationen, die ihre fachlichen Anforderungen an eine spezifische technischen Implementierung binden, haben deshalb die Aussicht, dass sie ihre fachlichen Regeln ständig neu implementieren müssen, um der sich ändernden Technologie zu folgen. Ein typisches Beispiel für die Verquikkung von Fachfunktion und Implementierung ist EDI / EDIFACT (*siehe Kapitel 3.4.11*). Ein sicherer Ansatz ist es, die Spezifikation der fachlichen Funktionalität soweit wie möglich unabhängig von der zugrunde liegenden Technologie zu machen, also die fachliche von der technischen Architektur zu trennen.

Allerdings fordern wir nicht, die Fachfunktionalität in kompletter Ignoranz der technologischen Möglichkeiten zu definieren. Schließlich wird ja der Einsatzbereich einer Applikation grundsätzlich von der jeweils existierenden Technologie definiert: Kein Electronic Business ohne das Internet!

Beim Planen einer neuen Anwendung ist jedoch das klare Verständnis der fachlichen Anforderungen und der Architektur des Geschäftsmodells der erste Schritt. Dieser Schritt definiert denn auch die ersten Phasen im Lebenszyklus eines Projekts *[Yourdon1995]*.

4.3.1 Phasen eines objektorientierten Projektzyklus

SELC

Software AGs *Software Engineering Lifecycle Model* (SELC), das wir hier präsentieren, unterstützt einen solchen Ansatz. Die Schritte in SELC sind:

❶ Konzeptphase

Das SELC deckt auch eine dem eigentlichen Entwicklungsprozess vorangestellte Konzeptphase mit ab, in der mittels Modellierung der Geschäftsprozesse – *Business Process Modeling* (BPM) – versucht wird, den Anwendungsbereich zu verstehen, das Projekt zu visualisieren und den Umfang des Projektes zu bestimmen.

❷ Analyse der Anforderungen

Was

Requirements Analysis Modeling ist die grundlegende Aktivität für den gesamten Projektzyklus. Die *Assets* und *Artefakte*, die in dieser Phase entwickelt werden, bilden die Grundlage für spätere Entwicklungsaktivitäten, und erlauben die etwaige Rückverfolgung (*traceability*) von Entwurfsentscheidungen.

Ein *Artefakt* ist ein greifbares Arbeitsergebnis, das notwendig ist, um die Funktionalität zu definieren oder zu erweitern, so z.B. Diagramme, Projektpläne oder Schemata.

Ein *Asset* ist ein Artefakt, der auch einen Zweck außerhalb des jeweiligen Projektes hat. Beispiele von Assets sind Artefakte, die wiederverwendbare Komponenten definieren, Pläne und Zeitpläne, die dazu benutzt werden, um das Projekt dem höheren Management darzustellen, und funktionale Beschreibungen für den Vertrieb und das Marketing.

Die Analyse der Anforderungen ergibt einen externen Blick auf das System aus der Perspektive des Fachklienten. Diese Phase erfasst und entwickelt die inhärente Fachfunktionalität und die fachlichen Regeln. Alle Aktivitäten und Arbeitsergebnisse werden in der Semantik des Fachbereiches ausgedrückt, also in der Sprache und den Begriffen des fachlichen Klienten. Alle in dieser Phase erzielten Arbeitsergebnisse sind für alle fachlichen Benutzer unmittelbar verständlich.

Kandidaten

Einige der während der Anforderungsanalyse erforschten Konzepte können möglicherweise nicht in ein Produktionssystem abgebildet werden. Bestimmte fachliche Interaktionen, die notwendig sind, fachliche Regeln auszuarbeiten und zu beurteilen, können außerhalb des Bereichs eines bestimmten Produktionssystems liegen. Die Konsequenz ist, dass viele dieser Konstrukte einen Kandidatenstatus erhalten *(candidate Use Cases*, *candidate Business Concepts*). Zu bestimmen, welche Kandidaten schließlich in das Produktionssystem überführt werden, ist Zweck der folgenden Entwicklungsphase.

❸ Analytisches Modell

Die Phase der Systemanalyse umfasst Entscheidungen, die den Problembereich des Gesamtsystems betreffen.

Analysis Modeling übersetzt die Ergebnisse der *Requirements Analysis* in ein objektorientiertes Format. Dabei ist jede objektorientierte Notation angemessen, sofern diese Notation integrierte statische und dynamische Aspekte unterstützt. Im Kontext unserer Diskussion setzen wir UML (*Unified Modeling Language*) als objektorientierte Modellierungssprache voraus *[Rumbough1998]*.

Die fachlichen Anforderungen (*Business Requirements*) werden in *Use Cases* übersetzt: Jede mögliche Art, in der ein Endbenutzer das System benutzt, definiert einen *Use Case*. Jeder *Use Case* definiert so eine Anzahl von Interaktionen mit dem System *[Jacobson1993]*.

Mittels einer formalen Methode wie UML transferiert die Systemanalyse das Wissen, das in der Anforderungsanalyse erworben wurde, vom fachlichen Bereich in den Bereich der Entwickler. Insofern kennzeichnet

die Erstellung das analytischen Modells einen 'Crew'-Wechsel: Die formalen Spezifikationen werden vom Systemanalytiker den Entwicklern übergeben. Der Systemanalytiker stellt dabei sicher, dass das analytische Modell die fachlichen Anforderungen reflektiert.

Der formale Charakter der benutzten Methoden erlaubt die Rückverfolgung (*traceability*) von Entscheidungen im Entwicklungsprozess: Spätere Artefakte können bis zu ihren Wurzeln, nämlichen den fachlichen Anforderungen zurückverfolgt werden und umgekehrt.

Das analytische Modell ist auch die Phase, in der *Analysis Patterns* identifiziert werden. Im Unterschied zu *Design Patterns [Gamma1995]* (*siehe Kapitel 4.1*), die gemeinsame Implementierungskonstrukte adressieren, beziehen sich Analysis Patterns auf fachliche Konstrukte. Analysis Patterns führen keine neue Funktionalität in ein Modell ein, vielmehr identifizieren sie bestehende Lösungen.

❹ Entwurfsmodell

Wie

Nun ist es Zeit, die Technologie einzuführen und die technische Architektur zu definieren. Das Entwurfsmodell (*Design Model*) ist die Anwendung der Technischen Architektur auf das analytische Modell. In der Anforderungsanalyse und der Systemanalyse hatten wird das »Was« adressiert; nun bringen wir das »Wie« zum Ausdruck.

Die technische Architektur enthält den kompletten Satz von technischen Konstrukten, die benötigt werden, das analytische Modell zu implementieren – Konstrukte und Techniken, wie sie in den vorigen Kapiteln dieses Buches diskutiert wurden. Das schließt die Aufteilung der Applikation in Teilbereiche (Kommentensysteme, *Packages*) und die Definition von Anwendungsschichten mit ein.

❺ Implementierungsmodell

Wo

Die Erstellung des Implementierungsmodells ist die Phase in der wir die Aspekte des Einsatzes (*deployment*) in der technischen Architektur adressieren. Wenn Anforderungsanalyse und Systemanalyse das »Was« behandelten und das Entwurfsmodell das »Wie«, so definiert das Implementierungsmodell das »Wo«. Das Implementierungsmodell ist eine entscheidende Aktivität für komponentenbasierte und verteilte Systeme (*siehe auch Kapitel 2.6*). Im Gegensatz dazu ist das Implementierungsmodell für monolithische Applikationen trivial: Die Frage des »Wo« ist dort leicht zu beantworten.

In dieser Phase behandeln wir auch die Frage der Integration von Altanwendungen und von Systemen von Fremdherstellern.

❻ Kodieren und Zusammenbau

In dieser Phase kommen wir zur Produktion des ausführbaren Codes. Das schließt gewöhnlich die folgenden Aktivitäten ein: Code schreiben, Applikationsteile aus bereits existierenden Komponenten zusammenstellen, Verpackungstechniken (*wrapping*) auf Altsoftware anwenden u.s.w.

❼ Qualitätssicherung und Testen

Auch in dieser Methodologie haben Qualitätssicherung und Testen dieselbe Funktion wie auch in anderen Methodologien: Die Verifikation und Validierung der technischen und fachlichen Funktionalität über alle Phasen des Projektzyklus hinweg.

Bei den objektorientierten Entwurfs- und Implementierungsmethoden schließt die Teststrategie für eine Applikation die folgenden Schritte mit ein:

- **Klassentest.** Klassen sind die kleinsten Einheiten innerhalb einer objektorientierten Teststrategie. Der Entwickler stellt sicher, dass die Methoden der Klasse sich korrekt ausführen lassen und dass der interne Zustand der Klasse und die veröffentlichen Eigenschaften (Felder) korrekt gesetzt werden. Erinnern wir uns, dass der interne Zustand eines Objektes sich während des Lebenszyklus des Objektes ändert und dass die Methoden ihr Verhalten abhängig vom Status des Objektes ebenfalls ändern können.
- **Szenariotest.** Hier testen die Entwickler die Interaktion zwischen Klassen, und zwar auf Grundlage von Szenarien und Mustern (*patterns*), die in den Modellierungsphasen entwickelt wurden. In dieser Phase sind die Details der Codierung noch sichtbar und zugänglich.
- **Use Case Test.** Bei der Validierung der *Use Cases* liegt der Schwerpunkt auf der Verifikation der fachlichen Funktionalität. Verantwortlich ist hier das Qualitätssicherungsteam, das keinen Zugang zum Programmcode hat.
- **Package Test.** *Packages* fassen aufeinander bezogene Klassen zu größeren Einheiten zusammen, die die Funktionalität der Applikation in kohärente Untermengen zerlegen. Der Schwerpunkt in dieser Testphase liegt auf den Interaktionen zwischen diesen *Packages*.
- **System Test.** Die letzte Aktivität im Qualitätssicherungsprozess. Der Schwerpunkt liegt hier auf dem Gesamtsystem und seinen Interaktionen mit anderen Systemen. Hier sind nicht nur funktionelle Themen wichtig, sondern auch Themen wie Performanz und Skalierbarkeit.

Zusätzliche Qualitätssicherungsmethoden ergänzen das Testen. Ein oft wiederholter Satz ist der, dass Testen nur das Vorhandensein von Fehlern feststellen kann, nicht aber deren Abwesenheit. Wir wiederholen diesen Satz gerne noch einmal.

Peer Review

Unter dem Begriff des *Peer Review* verbirgt sich eine der effektivsten Methoden der Qualitätssicherung. *Peer Review* bedeutet, dass Arbeitsergebnisse auf allen Ebenen von unabhängigen Kollegen überprüft werden. Das fängt bei der Anforderungsanalyse an, kulminiert im *Code Review* und endet mit dem Überprüfen der Teststrategien.

❽ Einsatz

Der letzte Schritt ist der Einsatz (*deployment*) der Applikation. Dieser Schritt schließt nicht nur die Verteilung der Software über die Server mit ein, sondern umfasst auch korrelierte Aktivitäten: Die Verteilung der Systemdokumentation, das Benutzertraining, den Aufbau von Support-Kanälen, usw. In Electronic Business-Szenarien ist das Trainieren der Endbenutzer und die Verteilung von Benutzerhandbüchern oft nicht möglich – hier werden effektive Support-Kanäle noch wichtiger.

❾ Wartung

Im Ansatz des SELC wird die Wartung einfach zu einer erneuten Anwendung der Phasen des Projektzyklus – ein neuer Zyklus in einem iterativen Prozess. Einer der wichtigsten Belange der Wartung ist es, herauszufinden, wann die eingegangenen Änderungswünsche und Verbesserungsvorschläge eine weitere Entwicklungsanstrengung rechtfertigen. Das beste Vorgehen ist hier, die Änderungswünsche auf *Use Cases* abzubilden und von dort aus festzustellen, wie viel Aufwand nötig wird.

Für jede Phase im Projektzyklus können wir drei Komponenten identifizieren:

- **Rollen und Ressourcen.** Die Personen, die an dieser Phase teilnehmen.
- **Aktivitäten.** Was diese Rollen und Ressourcen während dieser Phase tun.
- **Assets und Artefakte.** Was in dieser Phase produziert wird.

Kein Projektzyklus kann als statisch angesehen werden. Jeder Projektzyklus muss sich anpassen und wachsen, um Fortschritte in der Technologie zu berücksichtigen, neuen fachlichen Anforderungen und Umgebungen nachzukommen und die wachsende Erfahrung des Projektteams zu reflektieren.

4.3.2 Der iterative inkrementelle Projektzyklus

Evolutionärer Ansatz

Das Konzept der inkrementellen und evolutionären Entwicklung hat sich bereits Mitte der Achtzigerjahre entwickelt. Die inkrementelle Entwicklung berücksichtigt, dass es fast unmöglich ist, ein perfektes System in einem einzigen Schritt – ohne Rückkopplung (*feedback*) vom Endbenutzer – abzuliefern. Oft ist es besser, zunächst nur Kernbereiche des Geschäftsmodells in einer ersten Version zu implementieren, anstatt eine vollständige und perfekte Lösung in einem Schritt erreichen zu wollen. Die praktischen Erfahrungen, die mit frühen Versionen des Systems gewonnen werden, sind wertvolle Schätze, die helfen, weitere Funktionalität in zukünftigen Inkrementen zu modellieren und zu implementieren. Die Implementierung neuer Funktionalität kann so auf einen stetig wachsenden Schatz von Erfahrungen gegründet werden und so die fachlichen Anforderungen besser erfüllen als ein System, das nur am 'Reißbrett' entsteht.

Der SELC unterstützt einen solchen inkrementellen Entwicklungsprozess und erlaubt mehrere Iterationen innerhalb eines jeden Inkrements.

Was ist ein Inkrement?

Im inkrementellen Ansatz wird das Projekt in eine Anzahl Mini-Projekte aufgebrochen. Jedes dieser Mini-Projekte liefert eine komplette Implementierung ausgewählter fachlicher Funktionen. Jedes Inkrement enthält eine Untermenge der *Use Cases* des Gesamtsystems. Wenn das Inkrement fertiggestellt ist, sind auch die *Use Cases* vollständig implementiert.

Was ist eine Iteration?

Ein Projekt in diskrete Inkremente aufzubrechen, hat den Vorteil, dass die Entwicklungsaktivitäten zunächst auf ein engeres Feld fokussiert werden und wichtige Funktionalität früher geliefert werden kann. Allerdings ist noch offen, wie diese Aktivitäten innerhalb eines bestimmten Inkrements organisiert werden.

Im iterativen Ansatz zerlegen wir jedes Inkrement in eine Anzahl von Iterationen, gewöhnlich drei. Diese Iteration decken alle Phasen des Projektzyklus ab, aber im Unterschied zu Inkrementen entwickelt nicht jede Iteration notwendigerweise die gesamte Funktionalität eines *Use Cases*.

Diese Iterationen sind:

1. **Exploration**. Die anfängliche Phase der Entdeckung.
2. **Evolution**. Die Erweiterung der Konstrukte und Prozesse, die während der Exploration entdeckt wurden.
3. **Verfeinerung**. Die endgültigen Verfeinerungen (*refinements*) für die vollständige Ablieferung.

Das Konvergenzprinzip

Dieser Drei-Iterationen-Ansatz ermöglicht die Anwendung einer Kernstrategie des Projektmanagements, nämlich die des *Konvergenzprinzips*. Dieses Prinzip sagt aus, dass – unter konsistenter Anwendung objektorientierter Technologie – die Lösungsmenge innerhalb von drei Iterationen mit einer maximalen Abweichung von 10% auf die tatsächlich benötigte Lösung konvergiert. Wenn korrekt angewandt, erhält man mit diesem Prinzip klare und frühe Warnzeichen für Fehler in der Spezifikation, mangelndes Verständnis der Anforderung, unangemessene Architektur oder instabile Umgebung.

Aufwandsverteilung auf Iterationen

Wie viel Aufwand muss auf jede Iteration verwendet werden?

Im Wesentlichen gibt es drei Strategien (*loading strategies*):

- **Back-end Loading.** In diesem Ansatz wird der Großteil der Arbeit in der letzten Iteration ausgeführt.

Exploration	< 20%
Evolution	33%
Verfeinerung	> 50%

  Diese Strategie wird paradoxerweise verwendet, wenn ein Problem nur mangelhaft verstanden wird oder die Entwickler unerfahren sind. Wenn man mit einem unbekannten Terrain oder Problemraum konfrontiert ist, ist es besser, zunächst rasch durch den Entwicklungsprozess zu gehen, um schnell einen breiten Überblick über alle Themen zu gewinnen.

- **Front-end Loading.** In diesem Ansatz wird der Hauptteil der Arbeit auf die erste Iteration verwendet.

Exploration	> 50%
Evolution	33%
Verfeinerung	< 20%

  Dieser Ansatz eignet sich am besten für Probleme, die trivial sind und gut verstanden werden. Hier kann die meiste Arbeit im ersten Durchgang erfolgen, ohne dass man Gefahr läuft, dass spätere Entdeckungen die gesamte Arbeit gefährden.

- **Linear Loading.** In diesem Ansatz werden die Anstrengungen gleichmäßig über alle drei Iterationen verteilt.

Exploration	33%
Evolution	33%
Verfeinerung	33%

  Kaum überraschend wird dieser Ansatz dann verwendet, wenn es nicht klar ist, wie gut ein Problem verstanden wird.

Integration

Während Inkremente lose gekoppelt sein sollten, gibt es doch klare Abhängigkeiten zwischen ihnen. So wird es immer die Notwendigkeit geben, Koordinierungsprobleme zwischen den Inkrementen zu lösen. Dieser Prozess wird *Integration* genannt.

Integration muss als eigenständige Projektaktivität eingeplant und adressiert werden. Dabei können zwei Integrationsstrategien verwendet werden:

- **Inkrementabhängige Integration.** In diesem Ansatz liegt es in der Verantwortung jeder Entwicklungsuntergruppe, die von ihnen abzuliefernden Arbeitsergebnisse mit denen vorheriger Inkremente abzustimmen.
- **Getrennte Integration.** Hier ist eine getrennte Untergruppe für alle Integrationsaktivitäten verantwortlich. Das Team nimmt die Arbeitsergebnisse der andern Entwicklungsteams entgegen, koordiniert sie und löst Diskrepanzen.

Unabhängig von der Strategie muss ein Projektleiter genügend Zeit für diese Aktivitäten bereit stellen.

Werkzeuge

Auf dem Markt sind verschiedene Werkzeuge, die den Softwareentwicklungsprozess oder Teile davon unterstützen wie z.B. die Systeme von *Rose* und *TogetherSoft*.

Zusätzlich können diese Werkzeuge bestimmte Qualitätssicherungsaufgaben automatisieren:

- Eine *automatische Bewertung* erlaubt es, die Komplexität der Modelle und der Implementierung zu überwachen.
- *Audits* sichern die Verwendung von Unternehmensstandards und die Befolgung von Konventionen.

- Die *Rückverfolgung von Anforderungen* (*Requirements Traceability)* erlaubt es, zu verfolgen, wie die definierten Anforderung in den verschiedenen Phasen des Modellierungs- und Implementierungsprozesses abgebildet werden.

Rundreise mit Bolero und Together

Software AGs Entwicklungssystem Bolero speichert Entwicklungsobjekte wie Klassen, Schnittstellen, Methoden und Felder in einem Repositorium und benutzt dabei zur internen Darstellung XML. Das erlaubt es, gegenüber Modellierungswerkzeugen offene Schnittstellen anzubieten.

Insbesondere kann hier Bolero mit *Together* von TogetherSoft zusammenarbeiten. Die Integration der beiden Produkte erlaubt ein *Round-Trip-Engineering*: So können UML-Diagramme in Code übersetzt werden, während umgekehrt Änderungen im Code auch wieder zurück in die Diagramme reflektiert werden können.