E8 Architektur & Design

Das Ergänzungskapitel soll in Architektur- und Design-Themen einführen. Dazu betrachten wir die Client-Server-Architektur sowie das Factory-Pattern. Allgemeine Betrachtungen über den professionellen Einsatz von Programmiersprachen runden das Kapitel ab.

E8.1 Client-Server-Architektur

Grundlegend für das Verständnis verteilter Systeme ist die Client-Server-Architektur. Das Softwaresystem besteht dabei aus zwei unabhängigen Komponenten: Dem Client (Benutzer), der die Dienste der Serverkomponente aktiv durch Requests in Anspruch nimmt und dem Server (Dienstleister), der diese Dienste allen Clients zur Verfügung stellt und auf deren Anfragen passiv durch eine Response reagiert. Typischerweise greift nur der Client auf den Server zu und "kennt" dessen Dienste, während der Server selbst nur auf Anfragen des Clients reagiert und keine weiteren Annahmen über Funktionen und Arbeitsweisen des Clients macht. Es liegt eine einseitige Zugriffsabhängigkeit: Client → Server vor – erst dem aktiven Request des Clients folgt die passive Response als Reaktion des Servers; ohne vorangegangenen Request erfolgt auch keine Response.

Bei Client und Server kann es sich um Prozesse handeln, die auf dem selben Rechner laufen. Bei der klassischen Client-Server-Architektur sind Client und Server jedoch auf verschiedene Rechner verteilt und kommunizieren über das Netzwerk. Der Client enthält die Benutzeroberfläche der Anwendung. Besonders schlanke Clients besitzen Webanwendungen – hier fungiert der Browser als (portierbarer) Client. Die gesamte Anwendungslogik und die Organisation der Datenhaltung (Persistenz) ist Aufgabe des Servers. Typischerweise greifen viele Clients auf einen Applikationsserver zu.

E8.1.2 Mehrschicht-Architekturen

Eine noch feinere Unterteilung der Client-Server-Struktur hinsichtlich der Aufgaben des Servers führt zur Mehrschicht-Architektur. Verschiedene Aufgaben des Anwendungssystems werden von separaten Komponenten übernommen, die jeweils auf Dienste der "tieferliegenden" Komponenten-Schicht zugreifen.

Wieder liegt eine einseitige Zugriffsabhängigkeit vor: Tiefere Schichten bieten Dienste an, ohne Komponenten höherer Schichten zu "kennen", die dieses Dienste in Anspruch nehmen. Komponenten einer Schicht dürfen aktiv nur auf Komponenten derselben und nächsttieferen Schicht zugreifen.

Für Komponenten einer Schicht ist dabei völlig *transparent* (nicht erkennbar und auch nicht relevant) wie Komponenten der tieferen Schicht ihre Dienste im Detail

erfüllen, bzw. ob diese selbst auf Komponenten tieferer Schichten zugreifen. Somit sind die Komponenten der einzelnen Schichten austauschbar.

E8.1.3 Drei-Schicht-Architektur (3-Tier)

Die klassische Architektur insbesondere im Bereich kommerzieller Anwendungssysteme (SAP R/3) ist die dreischichtige Architektur. Typischerweise verteilen sich die drei Schichten auch auf separate Rechner. Die Schichten sind:

- **Präsentationsschicht:** Sie realisiert die (meist grafische) Benutzerschnittstelle, nimmt Eingaben des Benutzers entgegen, leitet dessen Requests an den Applikationsserver weiter und stellt deren Response dar. Die Präsentationsschicht ist eine auf dem Client-Rechner des Benutzers fest installierte Anwendung oder auch häufig nur ein Browser bei Zugriff via Inter-/ Intranet.
- **Applikationsschicht:** Sie enthält die Programme der fachlichen Anwendungslogik und verwaltet aktuell benötigte Daten. Die Applikationsschicht ist der Systemkern; ihre Prozesse laufen auf einem leistungsstarken Applikationsserver.
- **Persistenzschicht:** Sie dient der dauerhaften Datenhaltung und wird meist durch ein relationales Datenbanksystem dargestellt, das die Speicherung und Verwaltung der Anwendungsdaten leistet. In der Regel ist das Datenbanksystem auf einem separaten leistungsstarken Rechner (DB-Server) installiert.

Grosser Vorteil mehrschichtiger Architekturen ist *Skalierbarkeit*: Nimmt die Zahl der gleichzeitig auf das System zugreiffenden Clients zu, so können deren Requests verschiedenen, parallel arbeitenden Applikationsservern des gleichen Systems zugewiesen werden, die mit derselben Datenbank arbeiten können. Die Zahl der Applikationsserver-Prozesse kann der wachsenden Belastung des Systems angepasst werden.

E8.1.4 Fünf-Schicht-Web-Architektur (5-Tier)

Bei umfangreichen Webapplikationen macht es Sinn, von einer maximal fünfstufigen Schichtarchitektur auszugehen. Typisch ist dabei eine feinere Unterteilung der Präsentations- und Applikationsschicht. Die fünf Schichten sind:

- **Clientseitige-Präsentationsschicht:** Diese besteht aus dem Webbrowser als Benutzerschnittstelle und alleinigem Frontend. Übermittelte HTML-Seiten werden darstellt. Eventuell können diese clientseitigen Skriptcode enthalten, um komfortable Benutzerinteraktionen und Eingabeprüfungen zu ermöglichen.
- **Serverseitige Präsentationsschicht (Webserverschicht):** Sie nimmt http-Requests des Clients entgegen, ruft die Anwendungslogik entsprechend auf und generiert (meist auf Basis von Server Pages) dynamisch eine Response im HTML-Format.
- **Anwendungslogik:** Sie enthält die fachliche Logik und setzt die betrieblichen Anwendungsfälle des Systems um. Zur Datenhaltung greift sie auf gesonderte Anwendungsobjekte zu.
- **Anwendungsobjekte:** Diese dienen der aktuellen Datenhaltung für laufende Anfragen auf dem Applikationsserver und garantieren den transaktionalen Da-

tenabgleich mit der Datenbank. Die Anwendungsobjekte stehen applikationsseitig entsprechenden Datenbank-Entitäten gegenüber und repräsentieren die betriebswirtschftlich relevanten Gegenstände der Webapplikation.

- **Persistenzschicht:** Ein (meist) relationales Datenbanksystem, das die Speicherung und Beschaffung der Daten der Anwendungsobjekte leistet.

E8.1.5 Ausblick: J2EE und Fünf-Schicht-Architektur

Die Java Enterprise Edition (J2EE) stellt eine Spezifikation dar, die von zahlreichen (auch frei verfügbaren) Applikationsservern implementiert wird. Das fünfschichtige Architekturmodell kann durch J2EE-Technologien direkt umgesetzt werden, wie Tabelle E1.1 aufzeigt:

Schicht	Verteilungsort	Technologie
Client	Browser	HTML, clientseitiges Skript
Webschicht	Webserver	Java Server Pages (JSP)
Anwendungslogik	Applikationsserver	Java Servlets
Anwendungsobjekte	Applikationsserver	Enterprise Java Beans (EJB), JDBC
Persistenz	DB-Server	Relationales DB-System, SQL

Tab. E8.1: Fünfschichtige Web-Architektur und ihre Umsetzung mittels J2EE

Dies diene als kleiner Ausblick auf J2EE und das weite Feld Java-basierter Technologien. Dem Leser sollte es möglich sein, sich mit den erworbenen Java-Kenntnissen auch diese Gebiete zu erschliessen.

E8.2 Das Factory-Pattern

Das sogenannte *Factory-Pattern* diene als ein Beispiel für Design-Patterns (Entwurfsmuster).

Patterns stellen bewährte Lösungsstrukturen für häufig auftretende Probleme bereit; dank Patterns muss das "Rad nicht jedes Mal neu erfunden werden". Die *Pattern-Idee* wurde in den Bereich des Software Engineering durch Gamma, Helm, Johnson und Vlissides eingeführt und ist in dem dem Standardwerk:

E.Gamma, R.Helm, R.E, Johnson: "Entwurfsmuster", Addison-Wesley, 2004.

ausführlich beschrieben (Orginal in Englisch: "Desing Patterns").

Das Factory-Pattern löst das Problems, wie *Objektinstanzen* zweckmäßigerweise *erstellt* werden sollten: Bei der Erzeugung eines Objekts sind eventuell zahlreiche Daten zur Initialisierung zu beschaffen, vielleicht müssen sogar Netzwerk- oder Datenbankverbindungen oder andere Systemressourcen erstellt und verwaltet werden. All diese Aufgaben im Konstruktor (oder weiteren Methoden) der Klasse abzuhandeln, würde dessen Coding stark anwachsen lassen – das Klassencoding zur Erzeugung von Objekten würde eventuell umfangreicher als das Coding der eigentlichen Services und Operationen der Klasse, an denen deren Verwender interessiert sind.

E8.2.1 Object Factory

Eine *Object Factory* (Objektfabrik) [HOR02] ist eine Klasse, die *Instanzen* (Objekte) einer anderen Klasse *erzeugen und liefern* kann. Als Teil ihrer öffentlichen Schnittstelle verfügt sie über die nötigen *Erzeugungsmethoden*, die ein Client aufruft, der eine Instanz der Klasse benötigt. Intern erzeugt die entsprechende Methode der Object Factory eine Instanz der Klasse (mittels `new`-Operator), beschafft erforderliche Daten und Ressourcen und liefert dem Aufrufer eine *Referenz* auf das erzeugte Objekt zurück. Mit dieser Referenz kann der Client auf die öffentliche Schnittstelle des Objekts zugreifen und mit ihm arbeiten.

Eine Object Factory führt einen *klassenspezifischen Objekterzeugungsprozess* durch – je nach Klasse und deren Struktur und Semantik wird die Objekterzeugung anders aussehen. Somit wird eine Object Factory in der Regel *speziell* für eine bestimmte Klasse implementiert. Auch die öffentlichen Erzeugungsmethoden (und deren eventuelle Parameter) der Object Factory selbst werden klassenspezifisch sein: Jede Klasse erfordert andere Daten zur sinnvollen Initialisierung ihrer Objekte. Andererseits könnten durchaus Erzeugungsmethoden für mehrere semantisch verwandte Klassen in einer Object Factory zusammengefasst werden.

Grob könnte man eine Object Factory also als *externe Konstruktor-Klasse* einer anderen Klasse charakterisieren.

Die Architekturvorteile einer Object Factory lassen sich wie folgt zusammenfassen [HOR02]:

- Bereitstellung einheitlicher Erzeugungsoperatoren für Objektinstanzen
- Zentrale Erzeugung von Instanzen inklusive Ressourcenvergabe
- Logische Trennung von Erzeugungs- und Zugriffsoperationen.

Am Beispiel einer Kontoklasse sollen Aufbau und Verwendung einer Object Factory demonstriert werden.

E8.2.2 Beispiel einer Object Factory

Für folgende Klasse `Konto` soll eine Object Factory zur Verfügung gestellt werden:

```
class Konto {
   private String name;   private int kontoNr;
   private double saldo;
   public Konto( String nn, int kn, double sd ) {
      name = nn;   kontoNr = kn;   saldo = sd;
   }
   public double getSaldo() { return saldo; }
   // ...
}
```

Die Klasse `KontoFactory` kümmert sich um die Beschaffung der erforderlichen Daten. In diesem einfachen Beispiel werden die Daten einfach vom Benutzer abge-

fragt, bzw. die nächste Kontonummer ermittelt - und direkt an den Konstruktor der Klasse `Konto` weitergereicht. Denkbar wären aber auch vorgeschaltete Datenbank-abfragen: Hat der Kunde bereits ein Konto bei der Bank? Ist die ermittelte Konto-nummer wirklich verfügbar? All diese Operationen möchte man aus dem Coding der eigentlichen Konto-Klasse heraushalten und ihrer Factory aufbürden:

```
class KontoFactory {
  private static int counter = 0;
  public static Konto createKonto(){
      String n = IO.promptAndReadString( "Kontoinhaber: " );
      double s = IO.promptAndReadDouble( "Anfangssaldo = " );
      counter++;
      Konto k = new Konto( n, s, counter );
      return k;
   }
}
```

In der ausführbaren Klasse `Bank` wird die Factory verwendet:

```
class Bank {
   public static void main( String[] args ) {
      Konto k1 = KontoFactory.createKonto();
      double sld = k1.getSaldo();
      // …
   }
}
```

Im UML-Sequenzdiagramm der Abbildung E8.1 wird der zeitliche Ablauf und das Zusammenwirken der einzelnen Klassen verdeutlicht.

Solange der Prozess der Objekterzeugung einfach ist, nur wenige Daten erfordert und keine zusätzlichen Ressourcen benötigt, genügt sicherlich ein gewöhnlicher Konstruktor als Teil der Klasse. Die Object Factory bietet Vorteile bei komplexeren Objekterzeugungsprozessen.

Durch das *Zwischenschalten einer spezialisierten Object Factory* erreicht man eine logische *Trennung von Objekterzeugungscoding und eigentlichem Klassencoding.*

Ein weiterer technischer Vorteil: Öffentliche Konstruktoren sind Teil der Klassen-schnittstelle; auch bei Vererbung muss eventuell aus Unterklassen heraus mit ihnen gearbeitet werden. Umfangreiche Konstruktoren "schleppt" die Klasse also sichtbar als Teil ihrer Schnittstelle quasi "mit sich herum". Dagegen ist eine zugehörige Object Factory kein integraler Teil der Klasse, sondern mit dieser nur assoziiert – und zwingt einen mittels Vererbung arbeitenden Verwender nicht, sich damit auseinan-der zu setzen.

Neben dem Object Factory Pattern existieren noch *zahlreiche* andere Design-Patterns. Einen *Überblick* über gängige Patterns und knappen Einstieg in die komplexe Thematik vermittelt:

K.Eilebrecht, G.Starke: "Patterns kompakt", Spektrum Akademischer Verlag, 2003.

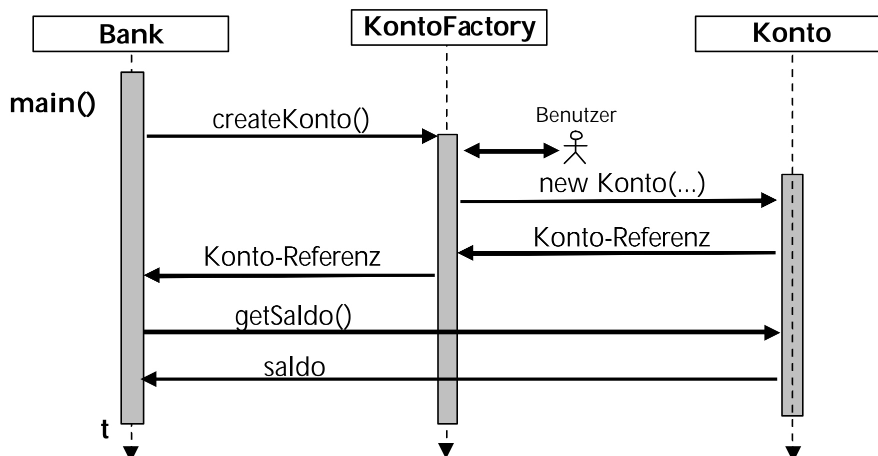

Abb. E8.1: Sequenzdiagramm (UML) zur Verwendung der Object Factory

E8.3 Features, Beurteilung und Auswahl von Programmiersprachen

Das Zusatzkapitel versammelt einige Gedanken zur Beurteilung von Programmiersprachen gerade auch aus betrieblicher Perspektive:

Programmiersprachen sollten so entworfen werden, dass damit insbesondere auch eine möglichst effektive Wartung der produzierten Software betrieben werden kann. Dies ist das grosse Problem von Firmen, die (als Softwarehaus bzw Application Service Provider) gegenüber ihren Kunden für eine große installierte Codebasis verantwortlich sind. Generelle *Anforderungen an Softwaresysteme* sind:

- Performanz und Skalierbarkeit auch bei zunehmenden Benutzerzahlen
- Sicherheit, Verfügbarkeit, Stabilität
- Wartbarkeit (Änderungen und Erweiterungen müssen sich einfach und übersichtlich integrieren, sowie isoliert testen lassen) und teilweise Wiederverwendbarkeit (Reuse in neuen Systemen) sowie Portierbarkeit auf neue Plattformen

Dies macht die *Qualität von Softwaresystemen* aus – und Programmiersprachen müssen grundsätzlich erlauben, Systeme zu produzieren, die diesen Eigenschaften entsprechen.

Begeisterte Entwickler können mit fast jeder Sprache mit Freude und Enthusiasmus zügig Neuentwicklung betreiben. Meist wird am Ende sogar eine lauffähige Anwendung stehen. Aber es ist die anschliessende Wartung des produzierten Codes (Erwei-

terungen, Umbau, Anpassungen, Fehlerbehebung, rechtliche Zusicherungen), die die Firma und ihre Entwicklungskapazitäten anschliessend bindet und teilweise auffrisst - nicht unbedingt die Neuentwicklung.

Häufig besteht die Gesamtlebensdauer erfolgreich entwickelten Codes nur zu 10% aus ursprünglicher Entwicklungszeit und zu 90% aus Wartungszeit. Die ursprünglichen Entwickler sind vielleicht bereits nicht mehr in der Firma, obgleich der Code noch immer in der Wartung ist - getätigt von wechselnden Entwicklern, die sich stets aufs Neue in das Coding einlesen müssen.

Dafür muss das Coding gut lesbar, übersichtlich und wartbar sein. Eine syntaktisch-symbolische Überfrachtung von Programmiersprachen oder schwer durchschaubare Sprachelemente (wie z.T. in C++ und auch in Java 5.0) sollte vermieden werden. Allzu leicht unterliegt man der technisch-logischen Faszination neuer Sprachfeatures.

Im professionellen Umfeld ist auch nicht die Struktur einer Sprache allein maßgeblich. Zur produktiven, nachvollziehbaren, stabilen Entwicklung und Wartung von SW-Projekten gehört auch eine grössere Infrastruktur, in der Entwicklung und Wartung stattfinden. Dies kann unter dem Begriff "Technologie" zusammengefasst werden [HOR02], der Programmiermodelle, Programmierschnittstellen- und Bibliotheken und ihre Implementation, Werkzeuge und Vorgehensweisen umfasst.

Typische Elemente sind (u.a.):

- Integrierte Entwicklungsumgebungen, die grössere Projektstrukturen übersichtlich darstellen und ein produktives Arbeiten im Team ermöglichen, eventuell auch auf der Basis von Codegeneratoren und grafischen Wizzards. Gerade die oft mühsame Oberflächenentwicklung sollte auf diese Weise erleichtert werden. Auch der Zugriff auf die folgenden Systeme sollte hierin integriert sein:
- Versionierungssysteme, die verschiedene Versionsstände verwalten und rekonstruierbar machen. Dies ist unabdingbare Voraussetzung für professionelle SW-Entwicklung und eine geordnete Wartung verschiedener SW-Stände, die sich bei diversen Kunden in Gebrauch befinden.
- Dictionaries und Repositories: Um das Rad nicht immer wieder neu zu erfinden, müssen entwickelte Klassen und andere strukturierte Datenytpen auch für andere Entwickler zugänglich und für neue Projekte zugreifbar sein. Dazu müssen Klassen und Schnittstellen- / Typbeschreibungen zentral und versionsspezifisch abgelegt und zugreifbar sein.
- Transportwesen: Der entwickelte Code muss in nachvollziehbarer und geordneter Weise vom lokalen Entwicklungsrechner oder Entwicklungssystem in andere Systeme überführt werden können. Dazu gehören auch Korrekturen am Coding.
- Korrekturwesen: Änderungen am Code (Fehlerbehebung, Patches) müssen in dokumentierter, nachvollziehbarer Form erstellt und in andere Systeme und natürlich auch zum Kunden überführt werden können.

Bestehende Applikationen müssen sich flexibel aktualisieren lassen, Wiederverwendung von Softwarebauelementen [HOR02] muss einfach möglich sein, durch Bibliotheken müssen sich (auch verteilte) Anwendungen rasch und zuverlässig entwickeln lassen.

Erst wenn für eine Sprache diese Infrastrukturanforderungen erfüllt sind, werden sich Softwarehäuser und IT-Abteilungen von Firmen zur intensiven Nutzung der Sprache in eigenen Projekten entschliessen. Gutes Beispiel ist die SAP AG, in der Java (neben der "Haussprache" ABAP OO) erst eine wichtige Rolle spielt, seitdem eine komplexe (zum grössten Teil eigenentwickelte) Infrastruktur der genannten Art zur Verfügung steht. (U.a. wurde eine eigene Entwicklungsumgebung auf der Basisi von Eclipse und eine neuartige Oberflächentechnologie auf der Basis von XML-Metadaten namens WebDynpro entwickelt.)

Ein noch übergeordneterer Gesichtspunkt ist die Verwendung nichtpropriäterer offener Standards des Datenaustausches zwischen (Neu-) Anwendungen und (zu integrierenden internen und externen Alt-) Systemen, wie z.B. auf der Basis von SOAP / WebServices zur flexiblen (Um-) Gestaltung und Anpassung von funktions-, abteilungs- und sogar unternehmensübergreifenden Geschäftsprozessen. Schnittstellen müssen einfach und ohne viel Programmieraufwand ansprechbar sein. Schlagworte wie Enterprise Application Integration (EAI), Hub & Spoke Architekturen (Datendrehscheiben), Bus-Technologie auf XML-Basisi und serviceorientierte Architekturen (SOAs) verlangen nach vernetzten, verteilten Anwendungen hoher Performanz und möglichst geringer Zahl von Schnittstellenimplementationen. An die Software werden hohe Anforderungen hinsichtlich Verlässlichkeit, Verfügbarkeit, Skalierbarkeit und Administrierbarkeit gestellt.

Letztlich stellt sich somit heraus, dass die einseitige Fixierung auf die reinen Sprachelemente einer Programmiersprache nur höchst unzureichend deren Nutzlichkeit im professionellen betrieblichem Umfeld erfasst.

Und zugleich wird dem Einsteiger in die Programmierung deutlich, wieviele zusätzliche Aspekte der professionelle Entwickler (besser: SW-Ingenieur) zu berücksichtigen hat - und wie weit das Feld des Wirtschaftsinformatikers ist.