1 Architektur Patterns

Max Kleiner

Die Software-Architektur beschreibt eine geeignete Strukturierung des Gesamtsystems sowie die zu verwendenden Standards, Schnittstellen und Protokolle. Diesen Strukturierungsüberlegungen wird eine mehrschichtige Referenzarchitektur, meistens das Model-View-Controller-Modell (MVC) oder die Layers zugrunde gelegt. [i]

Nebst dem sollte man für die Realisierung dieser Referenzarchitektur auch die Anforderungen an Programmiersprachen, Modellierungs - und Deployment-Werkzeuge beschreiben. Konkrete Vorgaben über die Verwendung von Produkten und Geräteeinheiten sollte man vermeiden.

1.1 Architekturziele

Das Ziel einer Softwarearchitektur besteht darin, ein klares Verständnis darüber zu erhalten, wie der Architekt ein Softwaresystem strukturieren soll und welche Protokolle, Formate und Schnittstellen vorgesehen sind. Eine Architektur basiert somit auf den zwei fundamentalen Hauptbereichen:

- Die Plattform
- Das Framework

Ein Framework setzt auf einer Plattform auf. Die Verbindung zwischen den Frameworks und den verschiedenen Plattformen ermöglicht die sogenannte Middleware. Das Team, zuständig für die Referenzarchitektur, sollte die Plattform wie die Frage des Frameworks mit den eingesetzten Komponenten vorher klären. Damit lässt sich ein zukunftsorientiertes, interoperables System bauen. Eine Architektur besteht im weiteren aus einer vertikalen wie horizontalen Schichtung:

- Die Struktur ist vertikal im Prozessraum eines Rechners, horizontal im Netzraum der verschiedenen Rechner zu sehen
- Vertikale Schichtung (logische Struktur) mit Schnittstellen zwischen den Packages und Komponenten – Framework Orientiert
- Horizontale Schichtung (physische Struktur) mit den Protokollen zwischen der eingesetzten Middleware – Plattform Orientiert

Zu empfehlen ist die Notation einer Architektur in UML, d.h. mit dem Paket- oder dem Komponentendiagramm für die Schnittstellen in vertikaler Sicht und mit dem Verteilungsdiagramm (Deployment) für die Protokolle aus horizontaler Sicht, siehe Musterdiagramm in Abb. 3.2.

Das Festlegen auf ein Schichtenmodell resultiert letztlich in einer Vereinheitlichung und Vereinfachung des Betriebs, womit nicht gesagt ist, dass ein Standard die Struktur einfacher macht, siehe die Architektur von J2EE. Eine Struktur kann kompliziert sein, mit dem Erklären zum Standard erhöht sich aber mit der Zeit das Verständnis.

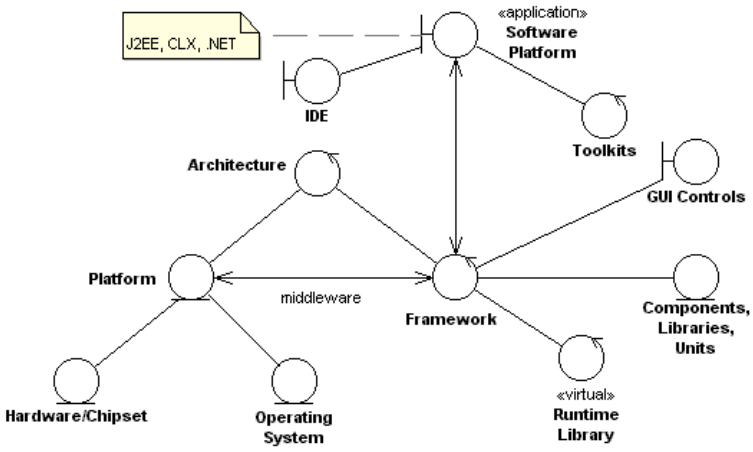

Abb. 3.1: Das Architekturschema im Softwarebau

Abb. 3.1 zeigt das generelle Architekturschema im „Schichtenbau“: Die vertikale Schichtung (logische Struktur) innerhalb eines Framework dient der geplanten und durchdachten Trennung verschiedener Verantwortlichkeiten durch bspw. Komponenten oder Module. Das Framework selbst besteht aus weiteren Zugehörigkeiten, die innerhalb der jeweiligen Software Plattform zu finden sind. Meist ist hier mindestens eine Abgrenzung zwischen View-, Logik- und Datenzugriffsschicht erforderlich.

Die horizontale Schichtung (physische Struktur) ermöglicht die Verteilung einer Applikation auf mehrere Plattformen (Geräte, Rechner) mittels der nötigen Middleware. Diese beiden Strukturen sind in Übereinstimmung mit der UML-Notation in Abb. 3.2 zu sehen. Mit der zugehörigen Softwareplattform als Entwicklungsumgebung lässt sich das Framework und indirekt auch die eingesetzte Architektur bestimmen.

Ziele der vertikalen Schichtung sind:

- Modularisierung
- Ausbaufähigkeit
- Wartbarkeit
- Gute Testfähigkeit
- Softwarequalität

Ziele der horizontalen Schichtung sind:

- Performance
- Sicherheit
- Skalierung
- Redundanz
- Portabilität

In einem ausgebauten Verteilungsdiagramm ist eine Referenzarchitektur mit der jeweiligen vertikalen (logical) und horizontalen (physical) Schichtung zu sehen. In der folgenden Abbildung 3.2 ist bspw. ein „Video on Demand System“ (VDS) mit den beiden Schichten einer vollständigen Architektur aufgespannt:

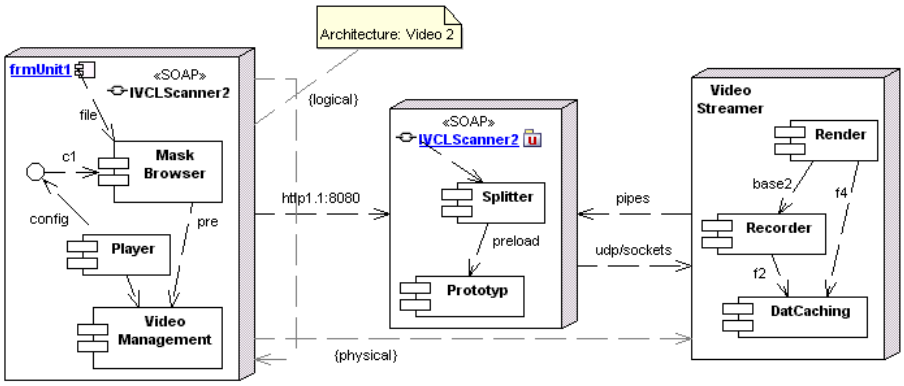

Abb. 3.2: Eine ganze Architektur im Deployment

Die einzelnen Komponenten innerhalb der Knoten kommunizieren **untereinander** über die Schnittstellen. Die Knoten als Geräteeinheiten kommunizieren dann über die zugehörigen Protokolle **nebeneinander**. Es ist auch möglich, dass Komponenten innerhalb eines Knoten schon Protokolle einsetzen, Interprozesskommunikation[1] genannt.

> *Mittelschichten im Sinne der Middleware sind dadurch ersichtlich, dass entweder ein ganzer Knoten eine Middleware darstellt oder die Zugriffspunkte in einem Knoten zwischen den Verbindungen eine Middleware symbolisieren. Bspw. regelt die Middleware den Zugriff auf sensitive Daten mit einem Socketmonitor.*

In komplizierteren mehrschichtigen Anwendungen können sich zusätzliche Dienste als Middleware „zwischen" den Clients und dem Remote-Datenbankserver befinden. Dabei kann es sich bspw. um einen Broker für Sicherheitsdienste handeln, der sichere Internet-Transaktionen gewährleistet, oder um Brückendienste für den gemeinsamen Zugriff auf Datenbankdaten auf anderen Plattformen, wie ein DB2-Connector.

Eine moderne Applikationsarchitektur muss die beiden Anforderungen logischer wie physischer Strukturierung mit den technischen Aspekten der Verteilbarkeit zusammenbringen. Idealerweise wird die Schichtung durch ein geeignetes Framework unterstützt oder zumindest darin implementiert. Somit gibt ein Framework immer die vertikale Architektur vor, nicht aber die horizontale im Sinne der Verteilung!

1.1.1 Kriterien einer Komponenten Architektur

Die zeitgemässe logische Architektur von Informationssystemen ist nicht mehr zwei- oder dreischichtig, sondern sie umfasst etliche Schichten mehr (n-tier Architektur).

> *Grundsätzliches Gestaltungskriterium ist die wiederholte Anwendung des bewährten Client-Server Prinzips.*

Dadurch sind derartige Anwendungen naturgemäss auf viele Rechner mit eventuell unterschiedlichen Plattformen verteilt (horizontale Schichtung).

[1] Out-of-Process-Server (lokaler Server) mit RPC-Protokoll

Wenn es um Diskussionen betreffend Sprachen und deren Architektur geht, enden die Debatten meist damit, dass die eine Sprache mit der Architektur schneller als die andere ist. Eigentlich geht es nicht um die Frage ob z.B. Java schnell genug ist, sondern die Frage müsste lauten, ist Java schnell genug für meine Architektur?

Im Verlauf der Debatte wird dann mit irgendeinem Zahlenmaterial im Sinne der Performanz argumentiert. Auch wir haben diese Performanzmessung einmal untersucht und sind zwischen Delphi und Java unter W2k mit einem simplen sequentiellen Lesen auf folgendes Ergebnis gekommen:

„Während die Java-Lösung bei einer 10 MByte grossen Datei 695 Millisekunden benötigt, braucht die Delphi-Lösung nur 161 Millisekunden, etwa Faktor 4 schneller. Der Übergang zwischen VM und nativem OS-Call bleibt also immer noch problematisch, zumal dieser Faktor 4 dann statt 2 Minuten eben 8 Minuten Wartezeit stipuliert."

Bei der Wahl einer Architektur sollten die Schnittstellen und die verfügbaren Protokolle mit dem Source-Code bekannt sein, damit Änderungen oder eine Neukompilation möglich sind. Auch die Kopplung und die Schichtung der Komponenten läßt sich mit der Source natürlich besser beurteilen.

Diese Beurteilung lässt sich konkret mit einem sogenannten Robustness-Diagramm vornehmen, welches mit drei Symbolen eine vertikale Schichtung modellieren kann.

- Verarbeitungsdominante Klassen realisieren Kontrolle und Steuerung von 1 oder n Use Case und werden auf <entity> oder <control> -Klassen verteilt.
- Dialogdominante Klassen sollten wenig «Wissen» enthalten und sind in <boundary> -Klassen zu finden. Diese Notation findet man in den Robustness Diagrammen wie folgt wieder:

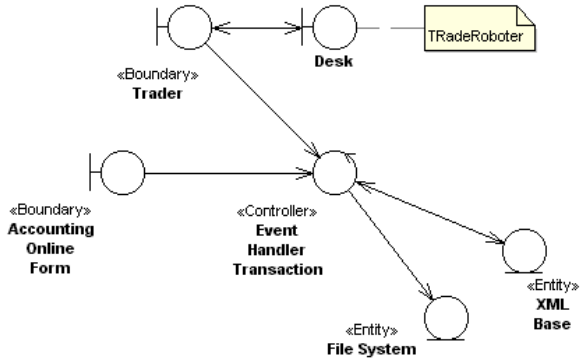

Abb. 3.3: Die Robustness Notation als Architekturhilfe

Jede bedeutende Firma, die Marktforschung im Bereich Informationstechnologie betreibt, hat vorausgesagt, daß die architekturbasierte Entwicklung die nächste große Welle in der Entwicklung von Applikationen sein wird.

Keine Architektur ohne Komponenten, wie Abb. 3.2. zeigt. Natürlich steckt hinter dieser Anspielung die künftige MDA (siehe Teil 1), die nicht nur architekturbasiert sondern achitekturgetrieben ist. Auch die Komponentenentwicklung besitzt das Potential, die

Produktivität zu verbessern, den Entwicklungsprozeß zu optimieren und flexiblere und stabilere Anwendungen zu entwickeln.

Welches sind nun die Kriterien für den Komponenteneinsatz, welche grösstenteils die Architektur mit der zugehörigen Middleware bestimmen:

- Geringe Einstiegskosten, da kein zusätzlicher Aufwand
- Lerneffekt bei den vorhandenen Algorithmen
- Es gibt Vorabversionen und Updates
- Installation, Support und Hotline der Komponentenfirma
- Volle Verfügbarkeit über den Sourcecode
- Bekannte Benchmarks als Performancekriterium
- Know How Transfer bei schon bekannten Komponenten
- Sicherheit und erhöhte Stabilität

1.2 Architektur Elemente

Die moderne Idee der Wiederverwendbarkeit von Software verkörpern Komponenten und Frameworks wohl am besten. Es sind die Hauptelemente einer Architektur (Protokolle und Schnittstellen sind der Kit dazwischen). Sie sollen die Wiederverwendung weiter vereinfachen. Im wesentlichen handelt es sich bei Komponenten um Klassen einer OO-Sprache mit spezifischen Anforderungen.

Der große Vorteil gegenüber einer Klasse besteht in dem **genormten** Interface und dem verborgenen Inhalt der Funktionalität. Hierdurch lassen sich sehr einfach auch Komponenten anderer Hersteller verwenden, deren Implementation man nicht kennt.

Wer kennt das nicht, man quält sich mit dem einen oder dem anderen Problem herum und vermutet, vielleicht hat da irgendwo schon jemand eine Unit oder die Komponente erstellt, die so manchen Entwicklungsaufwand auf ein Minimum schrumpfen lassen.

Man sollte aber nicht sofort dem Komponentglauben verfallen. Zu gross ist die Auswahl, die betriebsblind macht. Die Programmdatei kann unverständlich werden und wer weiß, vielleicht wird die Komponente in der nächsten IDE Version gar nicht mehr unterstützt oder man baut sich ungewollt Bugs in seine eigene Anwendung ein. Dann haben wir eine tickende Zeitbombe.

Gemäss dem Architekturschema in Abb. 3.1 gilt es die folgenden 4 Elemente einer Architektur zu beurteilen.

1.2.1 Schnittstelle

Wie kann man sich eine «moderne» Softwareschnittstelle (Interface) vorstellen? Eine Schnittstelle besteht aus einer Reihe von Methodendefinitionen, deren Funktionalität die Schnittstelle definiert (siehe Teil 1 Interfaces). Zu diesen Methodendefinitionen gehört die Signatur, d.h. die Anzahl und die Typen der Parameter, die man übergibt, der Rückgabetyp und das erwartete Verhalten.

Die Art und Weise, in der diese Methoden implementiert werden, ist nicht festgelegt, so daß in Bezug auf die Schnittstelle die tatsächliche Methodenimplementierung immer vollständig verborgen ist. Aus diesem Grund ist die Schnittstelle polymorph, solange die

Implementierung der Klasse in der Schnittstelle mit der Definition der Schnittstelle übereinstimmt.

Das heißt, daß der Zugriff und die Verwendung der Schnittstelle für jede ihrer Implementierungen identisch sind. In diesem Punkt gleicht eine Schnittstelle einer Klasse, die nur abstrakte Methoden und eine klare Definition ihres Verwendungszwecks aufweist. Genau wie abstrakte Klassen können auch Schnittstellen selbst nie instanziert werden.

1.2.2 Komponente

Je autonomer eine Komponente entwickelt wurde, desto leichter verwendbar ist sie für den Entwickler, wenn sie einen definierten Zweck erfüllt. Es liegt in der Natur von Komponenten, dass sie in einer Anwendung in den verschiedensten Kombinationen und in wechselndem Kontext vorkommen. In der Verantwortung der Komponentenbauer liegt es, dass sich die Komponenten ohne Vorbedingung und Abhängigkeiten in „fast“ jeder Situation korrekt verhalten.

Definition Komponente

Wann haben wir es mit einer Komponente zu tun? Ein Komponente besteht aus Klassen mit definierten, genormten und veröffentlichten Schnittstellen. Durch die Auswahl geeigneter Vorfahren für die Komponenten und mit Hilfe von Schnittstellen, die nur die vom Entwickler benötigten Eigenschaften und Methoden zur Verfügung stellen, lassen sich wiederverwendbare Komponenten entwickeln.

Wichtig sind die spezifizierten Schnittstellen nach außen mit dem Vermögen, Ereignisse zu verarbeiten oder weiterzugeben.

> *Ein Ereignis z.B. ist eine spezielle Eigenschaft, deren Wert sich zur Laufzeit als Folge einer Benutzereingabe ändert. Die Komponente gibt dieses Ereignis an den Anwendungsentwickler weiter, der auf verschiedene Arten von Eingaben reagieren kann, ohne neue Komponenten oder Routinen zu definieren.*

Eine DLL ist also keine Komponente vom Prinzip her, klar sie ist ein physisches Element als Bibliothek, aber die dokumentierte Referenz nach aussen fehlt.

Natürlich dauert es länger, Komponenten zu entwickeln, die frei von Abhängigkeiten sind. Die zusätzliche Zeit ist aber gut angelegt. Ausgereifte Komponenten ersparen nicht nur den Anwendungsprogrammierern alle möglichen Unannehmlichkeiten, sondern verringern auch ihren eigenen Aufwand für die Dokumentation und die technische Unterstützung.

Erst durch die Änderung der Eigenschaften wird es möglich eine Komponente anzupassen. Je mehr Möglichkeiten sie hierbei bietet, desto öfter läßt sie sich einsetzen und wiederverwenden. Doch damit die geänderten Eigenschaften auch sinnvoll sind, muß dem Anwender bekannt sein, welche Möglichkeiten er hat und wie er sie benutzt.

Die Dynamik von Komponenten wurde auch durch das Web vorangetrieben, indem die Komponenten mit Skripts zusammenarbeiten. Es ist verlockend, jedesmal ein Applet oder ActiveX als Komponente dynamisch vom Web-Server auf den Browser zu laden, dagegen erscheint ein „eingebautes“ Skript eher statisch und fehleranfällig.

Ein Skript bietet aber die Möglichkeit, Web-Anwendungen basierend auf generisch entwickelten Applets oder ActiveX-Steuerelementen anzupassen. Zum Beispiel liesse sich ein Applet so entwerfen, dass es basierend auf den Eingaben eines Anwenders ein Bild in 3D bearbeitet (rendering), das dann über HTML empfangen wird.

D.h. das Skript übergibt dem Applet als Komponente die entsprechenden Parameter, die der Anwender bei der Eingabe auswählt. Durch den Einsatz einer Skriptsprache kann man eine Kommunikation zwischen den verschiedenen Webkomponenten (Applets, ActiveX) realisieren.

Es gibt bereits konkrete sprachübergreifende Komponententechnologien, dazu zählen OpenDoc, EJB, ActiveX etc.. CORBA ist wohl eher ein mächtiges Konzept und eine Spezifikation innerhalb der Middleware, das erst mit den eingesetzten Tools seine Macht entfaltet.

Anbei noch einen Ausschnitt aus einer IDL-Datei (Interface Definition Language), welche die Schnittstelle zu einer Komponente sprachübergreifend definiert:

```
// Bank.idl
module UMLBank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
}
```

1.2.3 Protokolle

Was die Komponenten und Schnittstellen für die vertikale Schicht bedeuten, sind Protokolle und die Middleware für die horizontale Schicht. Viele Protokolle zur Steuerung der Aktivitäten bspw. im Internet sind in so genannten RFC-Dokumenten definiert (Request for Comment), welche die Internet Engineering Task Force (IETF) [ii]erstellt, aktualisiert und verwaltet. Es handelt sich dabei um eine Arbeitsgruppe für Protokollentwicklung im Internet.

Nachstehend seien die wohl wichtigsten RFCs aufgelistet:

- RFC822 (Standard für das Format von ARPA-Internet-Textbotschaften): Beschreibt die Struktur und den Inhalt eines Botschafts-Headers.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) Teil 1: „Mechanisms for Specifying and Describing the Format of Internet Message Bodies“
- RFC1945, Hypertext Transfer Protocol - HTTP/1.0: Beschreibt einen Transfermechanismus, der zur Verteilung Hypermedia-Dokumente benutzt wird.

Für die Entwicklung eines Netzwerkservers oder Clients benötigen Sie Kenntnisse über den Dienst, den Ihre Anwendung bereitstellt bzw. nutzt. Viele Dienste verwenden Standardprotokolle, die Ihre Netzwerkanwendung unterstützen muss. Wenn eine Anwendung für einen Standarddienst wie HTTP oder FTP zum Einsatz kommt, ist das Kennen des Protokolls und der zugehörigen Komponente teilweise unerlässlich.

Jedes Protokoll zur Einrichtung von Verbindungen zwischen Clients und dem Anwendungsserver besitzt spezielle Vorteile.

> *Bevor Sie ein Protokoll auswählen, sollten Sie die voraussichtliche Anzahl der Clients bestimmen, die Weitergabe der Anwendung berücksichtigen und geplante Weiterentwicklungen beachten.*

Für den Datentransport oder die Prozesssteuerung in einer Architektur lassen sich folgende Protokolle und Datenformate, basierend auf TCP/IP, einsetzen:

- HTTP(S): Maschinenunabhängiges, textbasiertes Datenformat, definiert durch die W3C. Das Protokoll ist Streaming- oder nachrichtenorientiert und fast allgegenwärtig. Im Gegensatz zu anderen Verbindungskomponenten sind über auf HTTP basierende Verbindungen keine Callbacks möglich.
- IIOP(S): Maschinenunabhängiges, binäres Datenformat, definiert durch die OMG in CORBA oder Java eingesetzt. Die Eigenschaften sind performant, interoperabel, nachrichtenorientiert, verschiedene Produkte und Open-Source-Implementationen. Das Interface wird sauber in Form einer IDL-Beschreibung abstrahiert, siehe oben Beispielcode.
- SOAP: Dieses Protokoll ist textbasiert und wird für die Repräsentation von Daten und die Ausführung von Prozessen verwendet, wie auch mit XML-RPC zunehmend für den Transport eingesetzt. Diese Kombination von Daten und Prozessen zeigt sich indem die Adressierungs- und Fehlerinformationen von der HTTP-Protokollschicht (URL, HTTP Status, Encodings) vermehrt in ein XML-basiertes Nachrichtenprotokoll wie JAX/RPC oder SOAP wandern.
- Sockets: Mit TCP/IP-Sockets können Sie extrem kleine Clients erstellen. Die von den Sockets bereitgestellten Verbindungen basieren zwar auf dem TCP/IP-Protokoll, sind aber so universell gehalten, dass auch verwandte Protokolle wie User Datagram Protocol (UDP), Xerox Network System (XNS) oder das DECnet von Digital verwendet werden können.
- Proprietäre Protokolle: Diese gelangen bei bestehenden Komponenten zur Anwendung, welche dafür standardisierte APIs anbieten und deren Verwendung die Entwicklung neuer Komponenten nicht tangiert. Ein Beispiel ist das DCE-RPC von MS, das mit DCOM zum Einsatz gelangt.

  > *DCOM benutzt für einen RPC (Remote Procedure Call) das RPC-Protokoll der Open Group's DCE (Distributed Computing Environment). Die Sicherung übernimmt das NTLM-Sicherheitsprotokoll (NT LAN Manager). Für Verzeichnisdienste verwendet DCOM das Domain Name System (DNS).*

Protokoll im Einsatz

Die Protokolle (Verbindungs-Komponenten) in Delphi basieren auch auf einer Klassenbibliothek, deren Sourcecode separat verfügbar ist. Grundsätzlich sollte man an einer Klassenbibliothek nichts editieren, da die Gefahr besteht, sich von der Weiterentwicklung der hierarchischen CLX abzukoppeln. Statt dessen lassen sich **eigene** Verbindungs-Komponenten durch Vererbung erstellen und in separaten Dateien und Projekten verwalten.

Als konkretes Protokolle dienen die bekannten Sockets als Verwendungsbeispiel. Das am weitesten verbreitetste Programmier-API für dieses Protokoll ist das Berkeley-Socket-Interface. Selbst MS ist über den Schatten gesprungen und hat die Winsock Definition in Windows voll integriert.

Bei einem Socket handelt es sich um einen Kommunikationsendpunkt. Das Socket-Interface orientiert sich an der Client-Serverarchitektur. Will ein Client-Programm mit einem Server Verbindung (z.B. auf einem anderen Rechner) aufnehmen, meldet es sich zuerst bei einem Socket-Dämon an.

> *Dessen Socket ist durch eine Internet-Adresse (IP) und eine Port-Nummer nach dem Akzeptieren im System eindeutig identifiziert.*

Der Socket Dämon startet einen **Serverprozess** und übergibt diesem nach dem `bind`() und `accept`() einen Handler auf einen Socket. Danach können Client und Server per `send`() und `recv`()-Funktionen Datenblöcke austauschen.

Durch einen Aufruf von `Create` können Sie ein Socket-Objekt erzeugen. Diese Objekte werden normalerweise von der zugrundeliegenden Socket-Komponente automatisch erstellt. Anwendungen können in einer Ereignisbehandlungsroutine für `OnGetSocket` einer Server-Socket-Komponente eigene Socket-Objekte erzeugen. Nach dem Aufruf des geerbten Konstruktor nimmt `Create` die folgenden Initialisierungen vor:

- Die Initialisierung von *WINSOCK.DLL* wird überprüft.
- Hilfsobjekte für das Multithreading werden zugewiesen.
- `ASyncStyles` wird mit [asRead, asWrite, asConnect, asClose] initialisiert.
- Die Eigenschaft `SocketHandle` wird mit dem Parameter `ASocket` initialisiert.
- Die Eigenschaft `Connected` initialisiert und prüft ob `ASocket` das Handle eines gültigen, geöffneten Socket ist.
- Die Eigenschaft `Addr` wird initialisiert.

Nach diesem Beispiel wollte ich verdeutlichen, dass der Einsatz von Protokollen auf Komponenten basiert. Ist eine ausreichende Anzahl an Komponenten mit zugehörigen Protokollen vorhanden, so ergibt sich die Möglichkeit, durchgehend neue Anwendungen so zusammenzubauen, wofür im Extremfall nicht eine Zeile Code entstanden ist.

Trotzdem haben Komponenten, die ein Protokoll implementieren und anbieten, einen gewissen Overhead, da sie in der Kompatibilität einen gemeinsamen Nenner erfüllen müssen. Bei zeitkritischen oder industrienahen Anwendungen ist dieser Ballast nicht immer akzeptabel. Allerdings sind dies Anwendungen, in denen ohnehin wenig vererbt oder abgeleitet wird.

1.2.4 Middleware

Der Begriff Middleware stiftet immer noch einige Verwirrung. Im Grunde ist es ein Teil Software, welcher für die Kommunikation zwischen heterogenen Systemen oder Komponenten eine Brücke oder ein Kanal als Vermittler bildet. Sonst bleibt es ein schwammiger Begriff. Middleware ist auch nicht identisch mit der OO-Komponenten-Technik, da einige Techniken, wie z.B. MQSeries oder MQBridge, völlig prozedural funktionieren. Middleware ist also nicht nur auf Komponenten angewiesen.

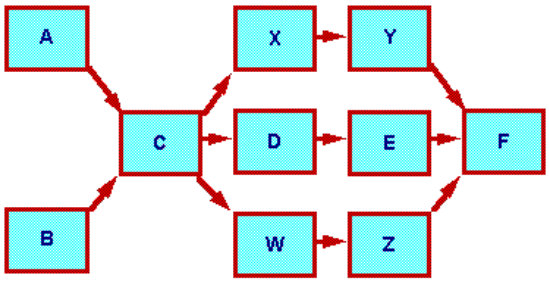

Abb. 3.4: Protokolle sind Aufgaben der Middleware

Innerhalb der Middleware, sozusagen als Verknüpfungspunkte, sind Protokolle im Einsatz, die sich auch kombinieren lassen, wie Abb. 3.4 zeigt. So kann C eine Middleware sein, die im Extremfall drei unterschiedliche Protokolle unterstützt.

Ich kann auch sagen, Middleware ist eine Art Klebstoff (Glue) wie ein Kitt, mit dem ich bestehende Anwendungslogik mit vorhandenen oder neuen Services zwischen verschiedenen Plattformen verbinden kann. Komponenten kann man sich ja als Software-Container vorstellen, die mit Middleware verdrahtet oder verschraubt werden. Ich bringe ein wenig Ordnung ins System. Es gibt drei Kategorien von Middleware:

- **Verarbeitungsorientiert**, synchron wie CORBA, SOAP oder RMI (Abb. 3.6)
- **Datenorientiert**, synchron/asynchron wie DB2Connector oder MIDAS (Abb. 3.7)
- **Nachrichtenorientiert**, asynchron wie JINI, MQ Series oder MOM (Abb. 3.8)

Dort wo eine Kombination vorhanden ist, spreche ich von kombinierten Mustern, denn auch Architekturen lassen sich in diese drei Kategorien einteilen.

> *Da eine Architektur das Framework und die Plattform definiert und die Middleware dazwischen ist, lassen sich diese drei Kategorien auch für die folgenden Architekturmuster nutzen.*

Middleware funktioniert meistens mit einer direkten, synchronen und eng gekoppelten Synchronisation zwischen den verteilten Softwarekomponenten. Das wichtigste Instrument für CORBA, RMI, DCOM oder SOAP ist nach wie vor die Idee des Remote Procedure Call (RPC). Messaging bei nachrichtenorientierter Middleware ist im Vergleich zu den RPC-Techniken noch wenig genutzt.

Bekannte Middleware Techniken sind etwa:

- TP-Monitor von TUXEDO (TP-Heavy oder Lite)
- CORBA Objectbroker VisiBroker von Borland
- Objekt-Transaktionsmonitore (OTM) wie COM+
- Java Transaction Monitor wie EJB
- SOAP mit WebServices und WSDL
- EntireX DCOM Erweiterung für UNIX
- MQSeries Nachrichtenorientiert von IBM

TP steht übrigens für Transaction Processing und symbolisiert die bisherige Technik im Gegensatz zu CORBA oder SOAP, welche die modernere Technik verkörpert. Interessant ist folgender Vergleich, der sich optisch eindrücklich darstellen lässt: Verarbeitungsorien-

tierte Middleware hat vielfach eine Gabelstruktur in den Sequenzdiagrammen[2] wogegen die Nachrichtenorientierung eine Treppenstruktur aufweist, wie Abb. 3.5 verdeutlicht.

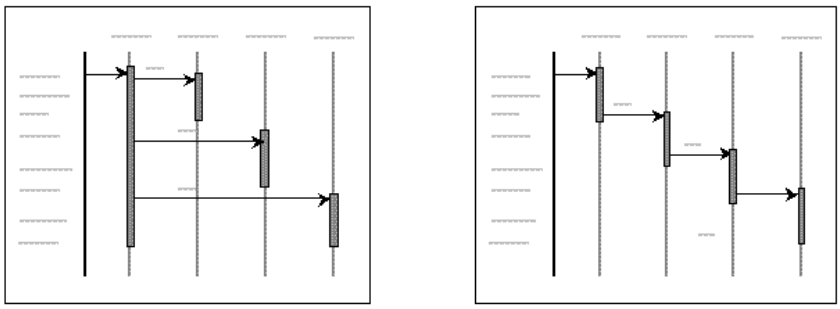

Abb. 3.5: Die Gabel und die Treppe bei Middleware-Architekturen

Bei der Verarbeitungsorientierung (Abb. 3.6), steht das Fernsteuern von Prozessen und Anwendungen im Mittelpunkt. Bei einem Anwendungsserver handelt es sich bspw. um eine Anwendung oder eine Bibliothek, die einem Client Dienste zur Verfügung stellt.

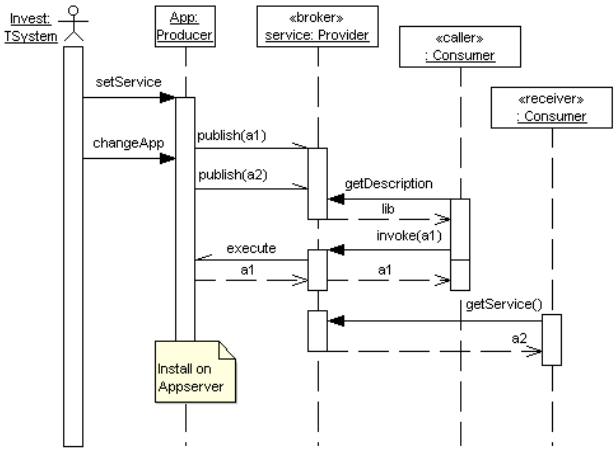

Abb. 3.6: Ein Broker vermittelt den richtigen Dienst

Bei der Datenorientierung sind vor allem Provider-Komponenten (`TDataSetProvider` und `TXMLTransformProvider`) im Einsatz. Sie stellen den Mechanismus bereit, durch den Client-Datenmengen ihre Daten empfangen.

Provider empfangen Datenanforderungen von einer Client-Datenmenge (oder einem XML-Broker), sammeln die gewünschten Daten in einem transportablen Datenpaket und

[2] Ist abhängig von der Anordnung der Instanzen

liefern sie an die Client-Datenmenge (bzw. den XML-Broker) zurück. Dieser Vorgang wird als Bereitstellen von Daten bezeichnet.

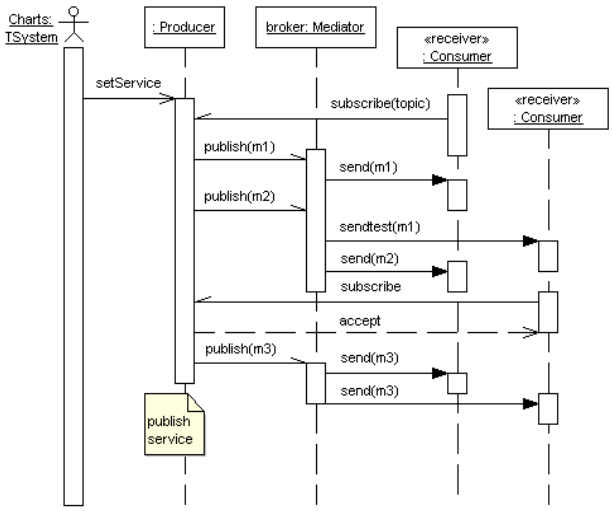

Abb. 3.7: Die Daten lassen sich vom Provider bereitstellen

Provider-Komponenten erledigen den Großteil dieser Arbeiten automatisch. Um Datenpakete aus den Daten einer Datenmenge zu erzeugen, XML-Dokumente zu erstellen oder Aktualisierungen anzuwenden, ist der eigene erstellte Quellcode gering. Dennoch verfügen Provider-Komponenten über eine Reihe von Ereignissen und Eigenschaften, mit deren Hilfe Ihre Anwendung direkt kontrollieren kann, welche Daten man für die Clients in Paketen zusammenfasst und wie man auf Client-Anfragen reagiert.

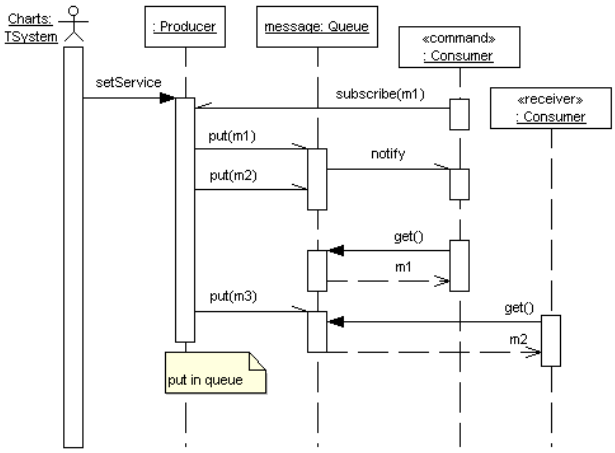

Abb. 3.8: Die Nachrichten lassen sich in die Queue legen

Nachrichtenorientierung ist vor allem in der Telekommunikation und im Monitoring im Einsatz. In der Zeit, wo Nachricht auf Nachricht folgt, muss niemand auf Empfang sein, da die Warteschlange ja Zeit und Kapazität hat, die Anfrage aufzunehmen. Vor allem bei Mobilnetzen, bei denen eine länger dauernde Verbindung schwer zu halten und teuer ist, lohnt sich diese asynchrone Architektur.

Middleware im Einsatz

Der Begriff OLE DB kennzeichnet eine OLE-Schnittstelle, die verschiedenen Anwendungen einen standardisierten Zugriff auf Daten aus unterschiedlichen Quellen bietet. Über eine einheitliche Schnittstelle hat der Entwickler Zugriff auf Datenquellen aller Art. OLE DB orientiert sich an den elementaren Funktionen einer Datenquelle. Weil OLE DB und Komponenten nicht direkt an die Datenquelle gekoppelt sind, sind sie über verschiedene Plattformen und Anwendungen einheitlich verteilbar.

Der heutige Zustand ist, die diversen Schnittstellen (ODBC, JDBC, OLE, MAPI etc.) einzeln kennen und unterstützen zu müssen. Künftig wird auf eine genormte Stelle zugegriffen, von der aus OLE DB die weiteren Knoten ansteuert. OLE DB unterstützt damit auch den Zugriff auf externe Daten (wie Tabellen, Dateisysteme, Emails, Projektmanagement-Tools oder Adresskarteien). Stichwort Adressen: wie oft hat man schon irgendwelche Adressen wieder neu angelegt oder unterschiedlich mutiert, weil die Interoperabilität und Austauschbarkeit im Programm fehlte.

Als zweite Verwendung lässt sich ein Anwendungsserver weitgehend auf die gleiche Weise erstellen, wie Sie andere Datenbankanwendungen erstellen. Der Hauptunterschied besteht in dem Remote-Datenmodul, das vom Anwendungsserver eingesetzt wird.

> *Bei Remote-Datenmodulen handelt es sich nicht nur um einfache Datenmodule. Das SOAP-Datenmodul z.B. implementiert in einer Web-Dienst-Anwendung eine aufrufbare Schnittstelle. Andere Remote-Datenmodule dienen als COM-Automatisierungsobjekte.*

Wenn Ihre Serveranwendung nicht mit DCOM oder SOAP arbeitet, müssen Sie die geeignete Laufzeitumgebung installieren, die Client-Botschaften empfängt, das Remote-Datenmodul instanziert und das Marshaling der Schnittstellenaufrufe übernimmt.

- Bei den TCP/IP-Sockets handelt es sich um eine Socket-Dispatcher-Anwendung namens *SCKTSRVR.EXE*.
- Für HTTP-Verbindungen wird *HTTPSRVR.DLL* verwendet, eine ISAPI/NSAPI-DLL, die zusammen mit dem Webserver installiert werden muss.

1.3 Softwareklassen

1.3.1 Libraries

Entwickeln Sie auch nach dem Minimumprinzip. Bei einem Datenbankzugriff liegt z.B. sehr viel Gewicht auf dem Aspekt der internen Wiederverwendung. Man möchte eben nur so wenig wie möglich implementieren. Patterns in Bibliotheken bieten hierzu ein

Palette an, da die Nutzenbetrachtung voll zur Geltung kommt und die Verwaltung der Muster auch in Bibliotheken[3] erfolgt.

Bibliotheken, Module, Packages oder Libraries sind die erste Softwareklasse, die durch Inventarisierung eine Möglichkeit der Wiederverwendung bieten. Unter einer in Designtime einsetzbaren oder zur Laufzeit ladbaren Bibliothek versteht man in Windows eine dynamische Linkbibliothek (DLL) und in Linux ein SharedObject (so) mit gemeinsamen Objekten. Als Beispiel einer Bibliothek sei SAPx erwähnt, die einen objektorientierten Zugriff auf das SAP RFC API erlaubt. [iii]

Ein Package ist eine auf spezielle Weise kompilierte Bibliothek, die von Anwendungen oder der IDE (oder beiden) verwendet wird. Packages ermöglichen eine Neuanordnung des Code ohne Auswirkungen auf den zugrundeliegenden Quelltext und deren Funktionalität. Dieser Vorgang bezeichnet man auch als Partitionieren von Applikationen, wie die Technik DLL+ im folgenden Diagramm zeigt:

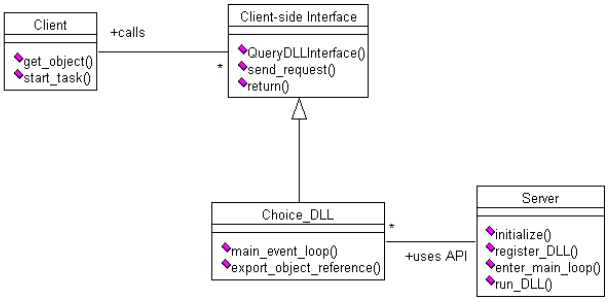

Abb. 3.9: Partitionieren mit DLL+ als OO-Library

Es handelt sich dabei bei Bibliotheken wie bei Paketen um eine Sammlung von Routinen, die sich von Anwendungen und von anderen Libraries bzw. gemeinsamen Objekten aufrufen lassen. Ladbare Bibliotheken enthalten wie Units gemeinsam genutzten Code und Ressourcen. Sie stellen jedoch eine separat kompilierte, ausführbare Datei dar, die der Compiler zur Laufzeit zu den Programmen bindet, die sie verwenden.

> *Klassische Design Techniken legen den Fokus hauptsächlich auf den funktionalen Bereich der Software. Für nicht funktionale Eigenschaften kann nur der Methodiker selbst durch seine Erfahrung wesentliches beisteuern.*

Genau hier setzen Bibliotheken an. Einerseits bieten sie funktionale Lösungen für wiederkehrende Entwurfsprobleme und andererseits bringen sie nicht funktionale Eigenschaften in das Design ein. Diese Eigenschaften ergeben sich daraus, daß Patterns sich bewähren mußten und aus einem Prozeß der Flexibilisierung und Forschung entstanden sind. Solche nicht funktionalen Anforderungen wie Austauschbarkeit, Skalierbarkeit,

[3] Bibliothek, Begriff aus einem Software-Kloster

Robustheit, Wartung oder Erweiterbarkeit der Software sind durch das Anwenden von Patterns und deren Bibliotheken implizit vorhanden. [4]

1.3.2 Toolkits

Vielfach lassen sich funktionsfertige Softwarebibliotheken in Applikationen einbinden. Auf der anderen Seite unterstützen Toolkits die Wartung, Installation oder Verteilung der Anwendung. Diese Helfer nennt man auch Toolkits. Ein Toolkit für einen Editor, ein Reportgenerator, eine Versionsverwaltung oder ein Toolkit für die gesamte Abwicklung einer Mehrsprachigkeit sind Beispiele dazu.

Ein bekanntes Toolkit sind z.B. auch die Delphi Stream Klassen, die Teil der CLX sind. Sie legen nicht ein bestimmtes Design oder ein navigierbares GUI für die Applikation fest, die man entwickeln will, sie dienen einfach dazu, allgemein verwendbare Funktionen nicht immer neu implementieren zu müssen. Man benutzt sie vor allem während der Entwicklung.

Setup-Toolkits automatisieren z.B. die Erstellung von Installationsprogrammen. Zusätzlicher Quelltext ist in den meisten Fällen nicht erforderlich. Mit Setup-Toolkits erstellte Installationsprogramme übernehmen verschiedene Aufgaben, die man bei der Installation von Anwendungen durchführt. Dazu zählen die ausführbaren und unterstützenden Dateien auf den Server kopieren, die Einträge in der Win-Registrierung vornehmen und für BDE-Datenbankanwendungen die BDE installieren.

Toolkits zu implementieren birgt eine Gefahr, nämlich die Situation zu verkennen, wie und wo im Code ein allgemeines Toolkit seinen Platz finden kann.

> *Eigentlich sind von den über 950 Objekten der CLX die meisten nicht visuell. In der IDE lassen sich aber einige dieser nicht visuellen Komponenten visuell in Designtime zu einer Anwendung hinzufügen.*

Grundsätzlich sollte man wissen, in welchen speziellen Situationen ein Toolkit seine Aufgabe erfüllt. Ein Toolkit legt kein spezielles Design für eine Anwendung fest, sie erfüllen Funktionen in einem klar umrissenen Kontext wie ein Komprimierer, ein Mediaplayer, ein Verschlüssler, ein Mailer, ein PDF-Formatierer oder einen HTML-Editor. Design Patterns haben dann die Eigenschaft, mit den Toolkits und Libraries Teile einer Architektur zu sein.

1.3.3 Frameworks

„Programming for change“

Ein Framework ist ein wiederverwendbares, teilweise fertiggestelltes Softwaresystem für eine Applikation als Business Framework innerhalb des Entwicklungssystems. Es besteht aus fertigen und halbfertigen Teilsystemen, wobei die **vertikale Architektur** des Systems durch diese Teilsysteme vordefiniert ist. Frameworks geben also die Architektur der Applikation vor, als Beispiel sei das geniale Bold (siehe MVC, PAC Muster und Abb. 3.59), Crystal Reports, IntraWeb oder MetaBASE erwähnt . Das hat den Vorteil, daß der

[4] Patterns sind in Packages gespeichert

Entwickler sich auf die spezifischen Funktionen seiner Applikation konzentrieren kann. Frameworks kennt man eigentlich in jeder IDE, sofern eine Integration erfolgt hat.

Das .NET Framework enthält Klassen und Komponenten, wie auch die VCL und CLX. Die VCL und CLX enthalten viele derselben Teilbibliotheken. Beide verfügen über BaseCLX, DataCLX, NetCLX. Die VCL enthält darüber hinaus WinCLX, während die CLX statt dessen VisualCLX beinhaltet. Ein Framework besteht wiederum aus Libraries, Components und Controls, siehe Architekturschema Abb. 3.1.

Einerseits muß man den Anwendungsbereich des Frameworks genau kennen, um festzulegen welche Bereiche sich flexibel gestalten lassen , und andererseits muß man zwischen der Flexibilität und dem späteren Anpassungsaufwand einen Kompromiß finden, zumal die horizontale Architektur noch nicht bestimmt ist.

> *Frameworks geben die existierende vertikale Architektur einer Applikation vor und stellen die Grundbausteine für ihre Erzeugung dar.*

Frameworks sind auf generelle (im Sinne einer Entwicklungsumgebung) wie spezielle Einsatzgebiete „programmtechnisch“ vorbereitet, da sie aus einer großen Menge von zusammenarbeitenden Klassen und Komponenten bestehen. Dieses Subsystem muß noch instanziert werden, damit es als Framework gilt. Fast jede IDE hat aufgabenspezifische Frameworks im Hintergrund. Die IDE bietet zudem alle Tools, um Anwendungen zu entwerfen, zu entwickeln, zu prüfen, von Fehlern zu bereinigen und weiterzugeben.

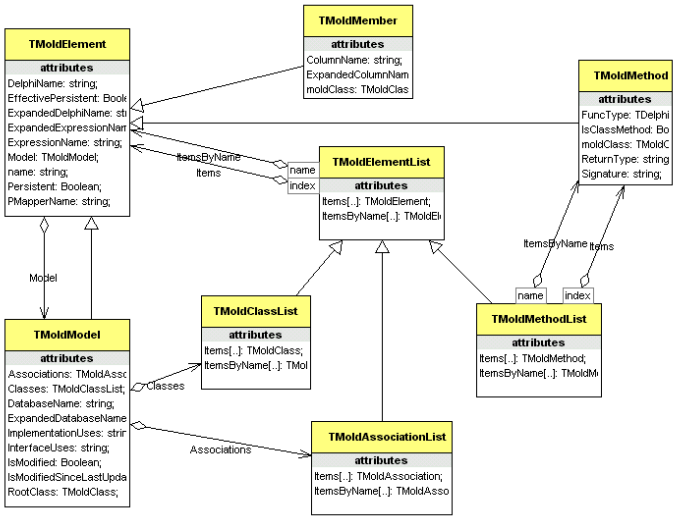

Abb. 3.10: Das Bold Framework der Metaklassen

Aber auch externe Frameworks lassen sich in einer IDE verwenden. Beispiel eines externen speziellen Framework ist ein samplingfähiger MIDI Recorder/Player, der die Architektur und die Schnittstellen bereits vorgibt. Frameworks haben viele Gemeinsamkeiten mit Design Patterns, aber dennoch gibt es drei wesentliche Unterschiede [DP95]:

- Patterns sind viel abstrakter als Frameworks und strikte objektorientiert. Patterns muß man jedes Mal neu implementieren, da sie nicht in konkreter Sprache vorliegen, während ein Framework schon als Instanz funktioniert.
- Patterns stellen im Gegensatz zu den fertigen Frameworks die kleineren Architekturelemente dar. Ein Framework basiert oftmals auf diversen Patterns, aber ein Pattern niemals auf einem Framework.
- Patterns sind weniger spezialisiert als Frameworks: Frameworks sind meistens auf bestimmte Anwendungsbereiche zugeschnitten, während Design Patterns durch ihre höhere Abstraktionsstufe bedingt, sich universeller einsetzen lassen.

[i] Buschmann et al., Pattern-Oriented Software Architecture, 1996, Wiley & Sons

[ii] RFC's der IETF: www.ietf.cnri.reston.va.us

[iii] SAPx Components: www.gs-soft.com