3 Architektur und Patterns

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.

3.1 Architekturziele

Das Ziel einer Software-Architektur 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 so genannte 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 den Erklärungen zum Standard erhöht sich aber mit der Zeit das Verständnis.

Abb. 3.1 zeigt das generelle Architekturschema im „Schichtenbau“: Die vertikale Schichtung (logische Struktur) innerhalb eines Frameworks dient der geplanten und durchdach-

ten 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.

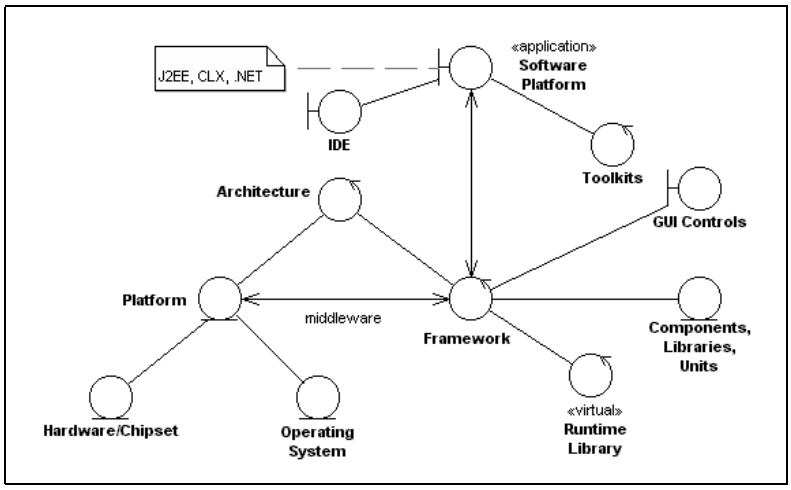

Abb. 3.1: Das Architekturschema im Softwarebau

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 folgen-

den Abbildung 3.2 ist bspw. ein „Video on Demand System“ (VDS) mit den beiden Schichten einer vollständigen Architektur aufgespannt:

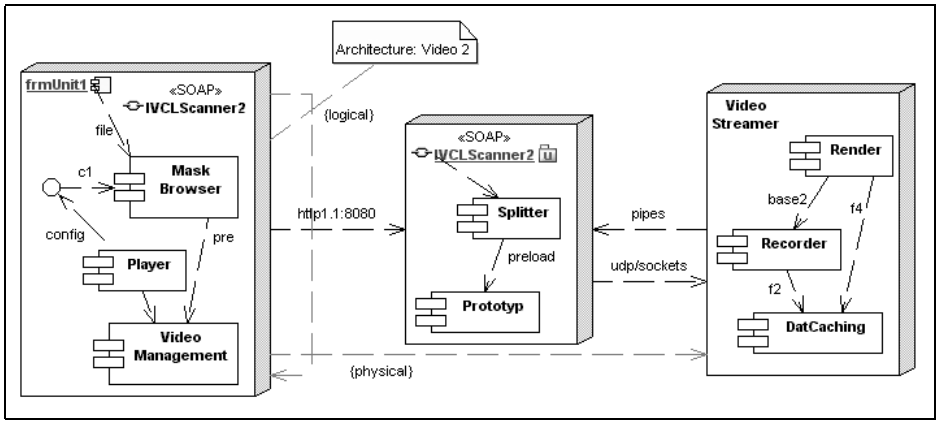

Abb. 3.2: Eine ganze Architektur im Deployment

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 Knotens 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!

3.1.1 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).

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

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).

Wenn es bei Diskussionen bezüglich 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 sequenziellen 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, ist also um etwa Faktor 4 schneller. Der Übergang zwischen VM und nativem OS-Call bleibt also immer noch problematisch, zumal dieser Faktor 4 dann statt zwei Minuten eben acht Minuten Wartezeit stipuliert."

Bei der Wahl einer Architektur sollten die Schnittstellen und die verfügbaren Protokolle mit dem Sourcecode 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 so genannten Robustnessdiagramm vornehmen, welches mit drei Symbolen eine vertikale Schichtung modellieren kann.

- Verarbeitungsdominante Klassen realisieren Kontrolle und Steuerung von 1 oder n-Use Cases 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 Robustnessdiagrammen wieder (Abb. 3.3).

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

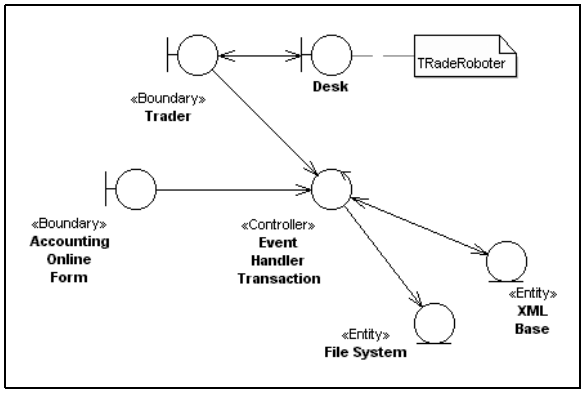

Abb. 3.3: Die Robustness-Notation als Architekturhilfe

3.2 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 Kitt 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 lässt.

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 DIE-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 vier Elemente einer Architektur zu beurteilen.

3.2.1 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, sodass 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.

3.2.2 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? Eine Komponente besteht aus Klassen mit definierten, genormten und veröffentlichten Schnittstellen. Durch die Auswahl geeigneter Vorfahren für die Komponenten und mithilfe 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 wieder verwenden. 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 Scripts zusammenarbeiten. Es ist verlockend, jedesmal ein Applet oder ActiveX als Komponente dynamisch vom Webserver auf den Browser zu laden, dagegen erscheint ein „eingebautes“ Script eher statisch und fehleranfällig.

Ein Script 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.

Das heißt, das Script übergibt dem Applet als Komponente die entsprechenden Parameter, die der Anwender bei der Eingabe auswählt. Durch den Einsatz einer Scriptsprache 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 ein 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);
  };
}
```

3.2.3 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 von Hypermedia-Dokumenten 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 obigen Beispielcode).
- SOAP: Dieses Protokoll ist textbasiert und wird für die Repräsentation von Daten und die Ausführung von Prozessen verwendet und wird 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 über-*

nimmt 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. Stattdessen lassen sich **eigene** Verbindungs-Komponenten durch Vererbung erstellen und in separaten Dateien und Projekten verwalten.

Als konkrete Protokolle dienen die bekannten Sockets als Verwendungsbeispiel. Das am weitesten verbreitete Programmier-API für dieses Protokoll ist das Berkeley-Socket-Interface. Selbst MS ist über seinen 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-Server-Architektur. 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 **Server-Prozess** 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 zugrunde liegenden Socket-Komponente automatisch erstellt. Anwendungen können in einer Ereignisbehandlungsroutine für `OnGetSocket` einer Server-Socket-Komponente eigene Socket-Objekte erzeugen. Nach dem Aufruf des geerbten Konstruktors 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.

Mit 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.

3.2.4 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 einen 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.

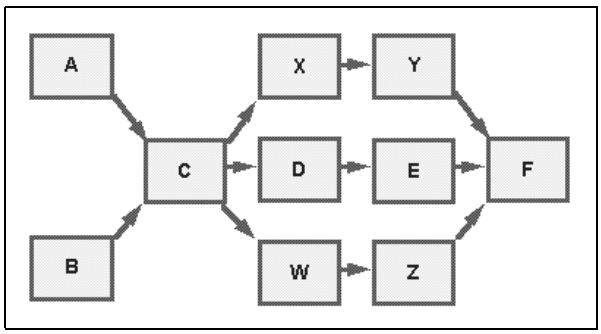

Abb. 3.4: Protokolle sind Aufgaben der Middleware

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 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 Instru-

ment 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 verkörpern. Interessant ist folgender Vergleich, der sich optisch eindrücklich darstellen lässt: Verarbeitungsorientierte Middleware hat vielfach eine Gabelstruktur in den Sequenzdiagrammen[2], wogegen die Nachrichtenorientierung eine Treppenstruktur aufweist, wie Abb. 3.5 verdeutlicht.

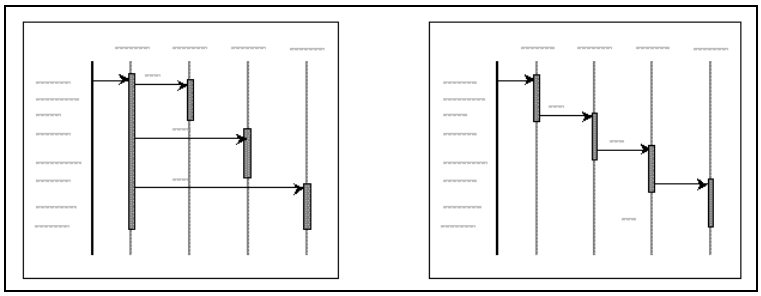

Abb. 3.5: Die Gabel und die Treppe bei Middleware-Architekturen

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.

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

[2] Ist abhängig von der Anordnung der Instanzen.

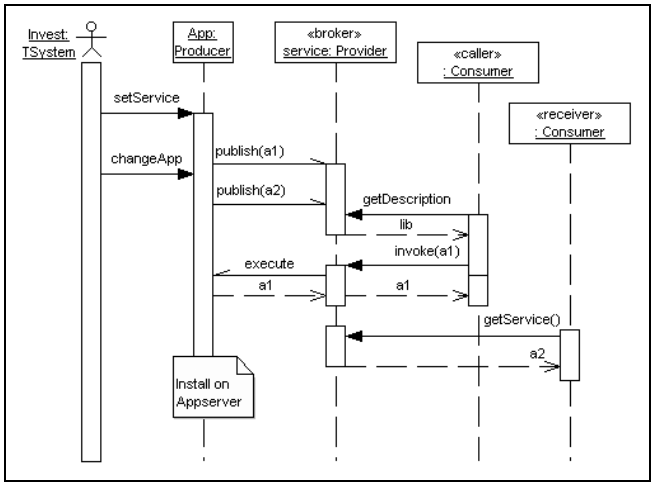

Abb. 3.6: Ein Broker vermittelt den richtigen Dienst

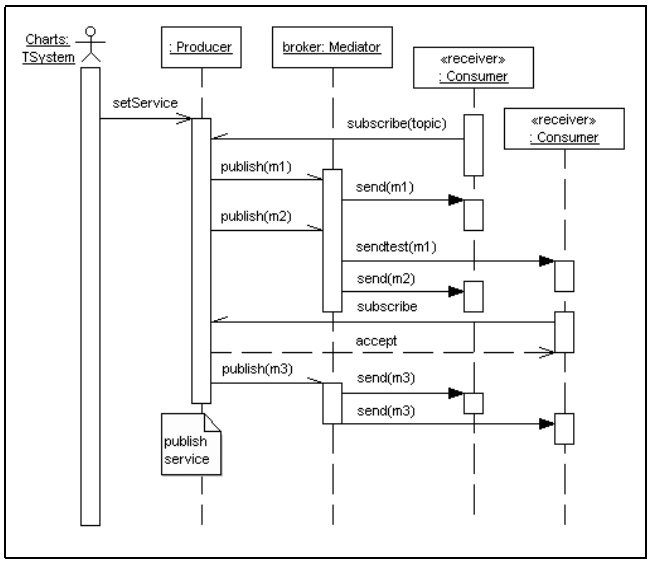

Abb. 3.7: Die Daten lassen sich vom Provider bereitstellen

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.

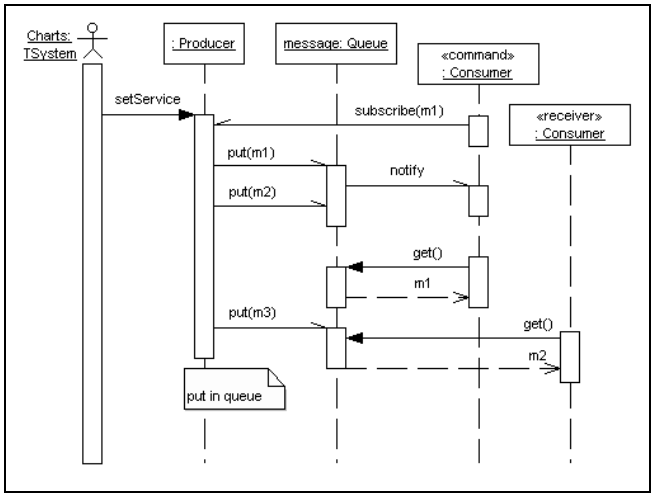

Abb. 3.8: Die Nachrichten lassen sich in die Queue legen

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 momentane 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, E-Mails, 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 weit gehend 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-Automatisierungs-Objekte.

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.

3.3 Architektur-Patterns

3.3.1 Automation (Bus)

Ein kombiniertes Architekturmuster (Softwarebus zu industriellen Anlagen).

Zweck

Ermögliche eine Kommunikation zwischen einem Leitrechner auf der einen Seite und den Mess-, Steuer- und Regelgeräten in der Feldebene auf der anderen Seite, unabhängig von der eingesetzten Hardwareschnittstelle.

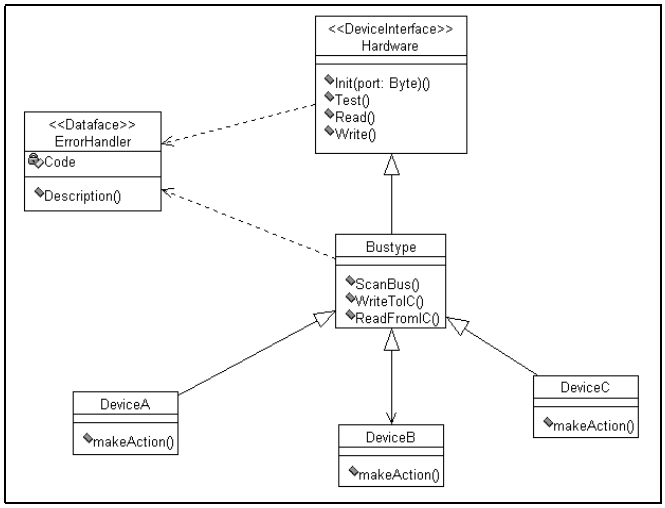

Abb. 3.9: Automation regelt die Aktoren und Sensoren

Motivation

Maschinen bauen Maschinen. Autos entwickelt man auf Supercomputern, ohne den Bildschirm zu verlassen. Herzerkrankungen lassen sich mit einer Computergrafik früh erken-

nen. Die Wetterprognose wird mit dem Computer gemacht. Ohne Computer geht heute nichts mehr. Vor dreißig Jahren hat er an der Wall Street Einzug gehalten; zuerst nur als Hilfsmittel und zur Automatisierung des Back Office. Die Financial Engineers brachten ihn an die Front und setzten ihn unter anderem für das Dynamic Hedging und die Verwaltung der großen Swapbücher ein, dann kam die Industrie mit den Feldbussen dazu. Vom Geld zur Produktion also.

Die datentechnische Schnittstelle zu den Mess- und Stellgrößen einer industriellen Anlage lässt sich häufig mit einem so genannten OPC-Server (OLE for Process Control) bilden. Dieser Server ermöglicht den kontrollierten Zugriff auf eine Vielzahl von Sensoren, die Daten über den aktuellen Zustand der Anlage geben, sozusagen Anlageberatung liefern ;).

Dadurch wird die explizite Beeinflussung der Anlage mithilfe der zugehörigen Stellgrößen, wie Pumpen, Relais, Hydraulik, Ventile oder Fördertechniken, möglich. Die Kommunikation des OPC-Servers mit den Mess- und Stellgrößen als physikalische Sensoren und Aktoren eingesetzt, erfolgt dabei über den in der Industrie gebräuchlichen Profibus, den Interbus oder einen anderen Feldbus. Eine Kopplung des Feldbusses mit einem „Industrial Ethernet“ ist immer häufiger im Einsatz und ermöglicht bspw. die Visualisierung von Messdaten oder ihre statistische Auswertung am PC.

Verwendung

Softwaretechnisch kann man einen OPC-Server als eine im Intranet verfügbare Echtzeitdatenbank nutzen, auf die man mit DCOM-Objekten Zugriff hat. Eigentlich eine proprietäre Lösung, wobei die meisten Architekturen dieser Art proprietär sind.

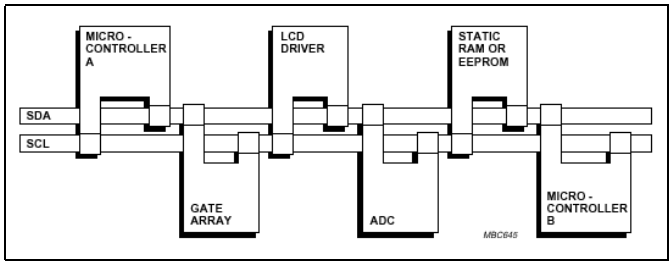

Abb. 3.10: Automation kommuniziert über einen Bus

Ein OPC-Server lässt sich mit einer globalen Konfigurationsdatei einrichten:

```
[Global]
Portnumber=2344
OPC-NodeAdress=121.012.017.001
OPC-Server=OPC.SimaticNet
MaxUsers=30
PathConfig=K:\TRade\ServiceNet\
```

`PathConfig` legt den Pfad fest, wo die einzelnen Anlagen- oder Maschinenkonfigurationen zu finden sind.

Das folgende Framework beschreibt eine ähnliche Struktur, die, ähnlich dem CAN-Bus in der Autoelektronik, im Einsatzgebiet der Unterhaltungselektronik anzutreffen ist. Ich wechsle also von der Makroansicht einer Anlagensteuerung zur Mikrosicht eines Gerätes. Es geht um den I^2C-Bus, der ausführlich im Buch[iii] „Messen, Steuern, Regeln mit Delphi“ beschrieben wird. Dieser Bus ist entwickelt worden, um in Elektronikgeräten, wie ein TV-Gerät, eine einfache Kommunikation zwischen den diversen ICs (Inter IC = I^2C) zu ermöglichen.

Dieser Bus hat konkret zwei Leitungen:

- SDA als Serial Data, in 8-Bit-Blöcken
- SCL als Serial Clock, seriell und synchron

Das Ansteuern und Initialisieren des Controllers erfolgt nun über ein Framework, das dem Architekturmuster entspricht und eine objektorientierte Steuerung erlaubt:

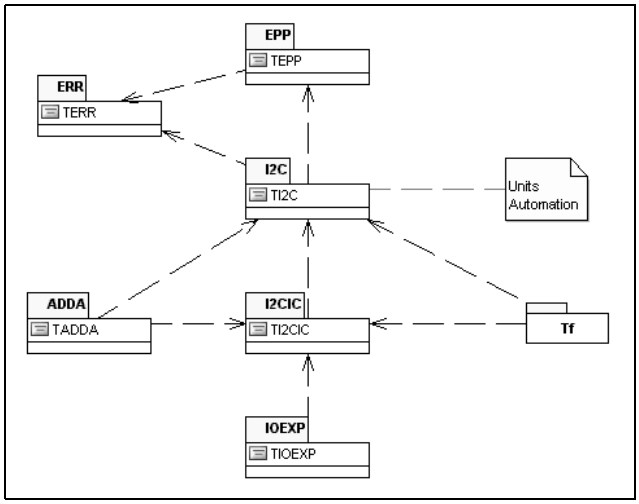

Abb. 3.11: Die ICs kommunizieren über den Bus-Controller

Dabei lässt sich der Bus bspw. nutzen, um einen Schrittmotor am IO-Expander anzusteuern. Diesen Expander kann man als Objekt instanzieren und die Botschaft `WriteValue(27)` senden. Die Initialisierung des Controllers sieht so aus:

```
procedure TI2C.Init(port: Byte);
var i: Integer;
begin
  m_epp.Init(port);
  WriteAdr($00);
  if not (ReadAdr = PIN_MASK) then
    raise TERR.Create(I2C_NO_I2C, 0);
```

```
  for i:= 0 to 29 do
      GenerateStop;
  //RESET/STOP --> dann mehrfach min. 30 clocks
  WriteReg(AR_8584, $55);
  //Takt 8 MHz, SCL 90 kHz in das (Clock-)Register S2
  WriteReg(CL_8584, $18);
  WriteAdr(ESIO_MASK);  //enable serial interface
  Sleep(10); //Warten zur Synchronisierung der IC's
  m_icinit:= true;
end;
```

Nachdem ich den Controller initialisiert habe, beherrscht er natürlich das zugehörige Protokoll auf der Interfacekarte. So implementiert er das korrekte Timing mit dem Synchronwort auf der Leitung SCL, die Serialisierung eines Datenbyte in die 8 Bit für die Leitung SDA sowie das Warten auf das Acknowledge-Bit und die Erzeugung der Start- und Stop-Bedingung in den Registern. Da bleibt mir abschließend zu diesem Muster noch zu bemerken:Von der großen Architektur zum kleinen Register mit Automation!

3.3.2 Broker

Ein verarbeitungsorientiertes Architekturmuster (Koordination der Kommunikation)

Zweck

Ermögliche die Struktur verteilter Softwaresysteme mit entkoppelten Komponenten, die durch das Aufrufen entfernter Dienste interagieren. Ein Vermittler (Broker) ist für die Koordination der Aufrufe und der Übermittlung von Ergebnissen verantwortlich.

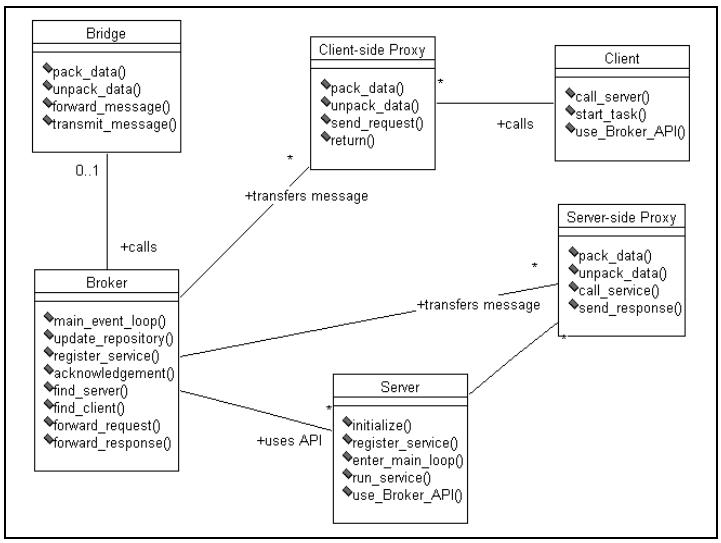

Abb. 3.12: Der Broker vermittelt

Motivation

Delphi stellt Assistenten und Klassen zum Erstellen verteilter Anwendungen zur Verfügung, die auf CORBA (Common Object Request Broker Architecture) basieren. CORBA ist eine Spezifikation, die von der OMG definiert wurde, um die Komplexität, die das Entwickeln **verteilter** objektorientierter Anwendungen mit sich bringt, standardisiert anzugehen.

Wie der Name suggeriert, stellt CORBA einen objektorientierten Ansatz für das Bauen verteilter Anwendungen dar. Sind CORBA-Anwendungen nun Komponenten?

Ja, denn die CORBA-Spezifikation definiert, wie Client-Anwendungen mit Objekten kommunizieren, die auf einem Server implementiert sind. Und diese Objekte können sehr wohl Komponenten sein. Die Kommunikation erfolgt über einen ORB (Object Request Broker). Delphi unterstützt CORBA über den VisiBroker-ORB.

Im Gegensatz zu COM/DCOM ist CORBA ein Standard, der auch für Nicht-Windows-Plattformen in einer Architektur geeignet ist.

Die Verwendung von Schnittstellen in CORBA-Anwendungen basiert auf der Verbindung von Stub-Klassen auf dem Client und Skeleton-Klassen auf dem Server. Diese Stub- und Skeleton-Klassen übernehmen alle Details der Schnittstellenaufrufe, so dass Parameterwerte und Rückgabewerte korrekt verpackt und übermittelt werden können.

> *Anwendungen müssen entweder eine Stub- oder eine Skeleton-Klasse verwenden bzw. das Dynamic Invocation Interface (DII) benutzen, das alle Parameter in spezielle Varianten konvertiert (damit diese ihre eigenen Typinformationstabellen enthalten).*

Mit SOAP hat das Architekturmuster eine aktuelle Alternative erschaffen. SOAP ist das Protokoll, welches der integrierten Unterstützung für Web-Dienste zugrunde liegt. SOAP übermittelt Methodenaufrufe unter Verwendung einer XML-Codierung und setzt HTTP als Transportprotokoll ein.

Verwendung

Angenommen, ein weltweites Netz von Leitrechnern sendet periodisch Diagnosedaten an einen zentralen InterBase-Server, der die ausgewerteten Daten dann per Browser für die Techniker bereithält. Das war mal die Vision, welche mit komplizierten und inkompatiblen RPCs (Remote Procedure Call) vor Jahren bei uns realisiert wurde. Mittlerweile gibt es SOAP, wodurch die künftigen Applikationsarchitekturen wie J2EE oder CLX wieder miteinander sprechen können.

> *SOAP-Verbindungen bieten den Vorteil, dass sie in plattformübergreifenden Anwendungen einsetzbar sind, da sie sowohl unter Windows als auch unter Linux unterstützt werden.*

In der folgenden Fallstudie möchte ich die Grundzüge dieser Technik als konkrete Verwendung des Architekturmusters näher bringen und vor allem das entfernte Speichern von Daten mit dbExpress erklären. Im Gegensatz zu den RPCs oder einer Web-Scriptsprache lassen sich Webservices voll und ganz objektorientiert in den Code integrieren und dann sauber kompilieren. Erweitert wird im Grunde der Funktionsumfang

eines Webservers, der meistens eine Ergebnismenge zurückliefert. Das Beispiel zeigt ein Funktionsmuster, das via Interfaces gesammelte Daten in eine Datenbank oder ein File speichert.

Das Fallbeispiel lief auch schon unter Delphi 6.0, zu empfehlen ist aber Delphi 6.02, das genauer mit WSDL und den entsprechenden Experten umgehen kann. Auch im Bereich dbExpress hat das ServicePack 2 einiges gebracht, zumal die Geschichte auch mit Kylix 2 läuft. Ich werde direkt in die Implementation des Piloten mit den folgenden Themen einsteigen.

- Technische Infrastruktur aufbauen
- Einrichten einer dynamischen dbExpress-Verbindung
- Generieren von HTML als Folge einer Abfrage
- Aufruf von Delphi, C# und Java aus

Das Delphi-Projekt, das für eine CGI-Web-Extension konfiguriert wird, besteht aus den vier Modulen Projektdatei, Webmodul, der Interface-Unit und der Implementierungs-Unit, welche den eigentlichen Zugriff auf den InterBase-Server bewerkstelligt. Unsere Web-Extension wird dann nach dem Speichern der Daten zusätzlich als Antwort auf eine HTTP-Anforderung ein Liste der eingetragenen „Techniker oder Leitrechner“ im Browser auflisten.

Als Laborumgebung benötigen Sie folgendes Inventar:

- Ein Webserver mit eingerichteten Benutzerrechten, in meinem Fall Apache 1.3 unter Linux im Shell-Modus
- dbExpress Native Treiber auf dem Server (wird mit Delphi installiert)
- InterBase mit der Standard-Vendor-Library *GDS32.DLL* und die beigefügte Datei *umlbank.gdb*

Bei den Vorarbeiten in der Delphi IDE unter `Options` empfehle ich, als output-directory das Script-Verzeichnis des Webservers einzutragen, z. B. *d:\apache\webshare\scripts*, da dort die EXE jeweils vom Server bei jedem Aufruf geladen wird und sich auch wieder schließt. Aktivieren Sie auch die Option einer Map-Datei, die stabileres Debuggen erlaubt, und deaktivieren Sie vorläufig den Code-Optimizer.

Und hier kommt das nächste Problem beim Testen, denn eine CGI-Extension lässt sich, außer man simuliert den Webserver mit einer komplizierten Konfigurationsdatei, nicht so einfach debuggen.

> *Hier ist es am besten, das Projekt als DLL zu kompilieren und dann mit den entsprechenden Startparametern als Host zu starten und zu debuggen. Dies erfordert in der Regel ein kleine Änderung in der Projektdatei.*

Als Erstes werfen wir einen Blick auf die Implementierungsklasse, welche die Schnittstelle `IVCLScanner` unterstützt. Die Registerkarte Webservices im Object Repository baut für Sie ja das Grundgerüst zusammen. Uns interessieren die Methoden `PostData` und `PostUser`, welche die Kernfunktionalität anbieten.

Die Funktion `PostData` speichert die gescannten Daten in ein File auf dem Server, und die Prozedur `PostUser` verewigt die zugehörigen Techniker in der Datenbank:

```
uses InvokeRegistry, Types, XSBuiltIns, VCLScannerIntf;
type
TVCLScanner = class (TInvokableClass, IVCLScanner)
public
function PostData(const UserData: WideString;
                    const CheckSum: DWORD): Boolean; stdcall;
procedure PostUser(const E-Mail, FirstName,LastName: WideString);
                                stdcall;
end;
```

Auf dem Webmodul haben sich mittlerweile die drei Standardkomponenten Dispatcher, Invoker und Publisher eines jeden Webservices platziert, wir benötigen noch einen TableProducer, welcher die gespeicherten Daten auch auf dem entfernten Browser sichtbar macht. Welche Aufgaben aber erfüllen diese Komponenten:

Die Komponente `THTTPSoapDispatcher` empfängt und beantwortet hereinkommende SOAP-Nachrichten und entscheidet, welches Interface man verwendet.

Das korrekte Objekt, welches das Interface realisiert, wird an den PascalInvoker weitergeleitet, der die richtig ausgepackte Methode mit den formatierten Argumenten einschließlich Errorcodes in den eigentlichen Aufruf inklusive Parametern transformiert.

> *`TWSDLHTMLPublish` ist verantwortlich für das Veröffentlichen von WSDL auf dem Browser, zusätzlich wird die WSDLADMIN-Datei im Webverzeichnis erzeugt, also aufgepasst, die EXE muss Schreibrechte besitzen.*

Die Struktur des folgenden Klassendiagramms zeigt nun das Beispiel in all seiner Pracht. Deutlich zu sehen, weil fast im Mittelpunkt, das Interface `IVCLScanner`, das wiederum von `IInvokable` abstammt. Ein kurzer Blick in die Quellen zeigt Erstaunliches.

Mit der Compiler-Direktive wird allen von `IInvokable` abgeleiteten Interfaces die Run Time Type Information (RTTI) ermöglicht, somit lässt sich zur Laufzeit der Typ von allen Methoden und Eigenschaften bestimmen, welcher nach dem XML-Transport mit SOAP wiederhergestellt werden muss. Normalerweise haben nur Published-Sektionen diese RTTI.

```
{$M+}
  IInvokable = interface(IInterface)
  end;
{$M-}
```

`TInvokableClass` selbst besitzt nur einen Konstruktor, der mithilfe einer Klassenreferenz auch zur Laufzeit den Typ bestimmen muss. Virtuelle Konstruktoren machen auch nur dann Sinn, wenn sie über eine Klassenreferenz aufgerufen werden. In den anderen Fällen steht ja schon zur Designzeit fest, welcher Konstruktor zum Zuge kommt, weshalb dann eine statische Bindung genügt!

```
TInvokableClass = class(TInterfacedObject)
public
```

```
  constructor Create; virtual;
end;

TInvokableClassClass = class of TInvokableClass;
```

Weiter zeigt das Klassendiagramm: Sowohl der Client als `TfrmMain` wie das Webmodule (vor allem der Dispatcher) kommunizieren strikt nur über das Interface. Da die Verwendung noch eine kleine Konfigurationsdatei einliest, gibt es auch eine Assoziation zwischen dem VCLScanner und dem Webmodule. In dieser Konfigurationsdatei *pathinfo.txt* sind Angaben betreffend Datenbank- und Dateipfad zu finden, da ansonsten der Datenbankort hart codiert wäre.

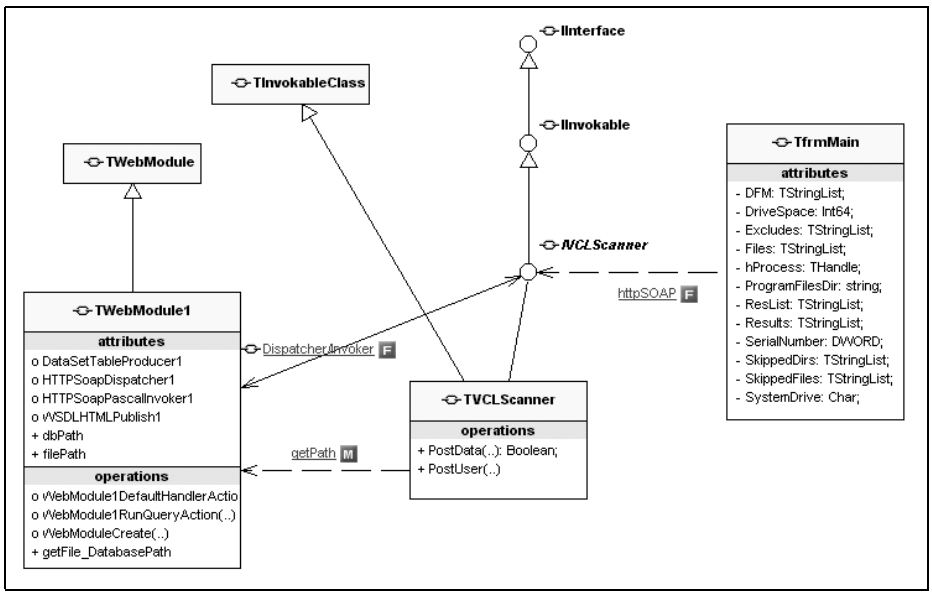

Abb 3.13: London sendet via PostData() an Frankfurt

Im Weiteren benötige ich einen Action-Eventhandler, der aus der Anfrage eine HTML-Seite produziert. Das SQL-Statement für die Abfrage folgt weiter unten. Mittels eines Doppelklicks auf das OnAction-Event des Webmoduls entsteht der gewünschte Methodenrumpf. Mit dem Index-Property `Action` können Sie alle Aktionen der Kollektion `Actions` durchlaufen.

0 ist der Index der ersten Aktion, 1 der Index der zweiten Aktion usw. In unserem Fall gibt es zwei Aktionen, den DefaultHandler für die WSDL-Publikation und die hausgemachte Aktion `runselect` für den TableProducer.

> *dbExpress ist eine Cross-Plattform, welche ein gemeinsames Interface für mehrere SQL-Server definiert. Für jeden unterstützten Server bietet dbExpress einen nativen Treiber bezüglich Queries und Stored Procedures für Linux und Windows an.*

Nebst der außerordentlichen Performance ist es vor allem die Konfiguration und Verteilung, die enorme Effizienz aufweist („dbExpress was designed to be fast, lightweight, and easy to deploy“). Wie Sie gleich sehen, genügen zwei Dateien (im Falle von zusätzlichem ClientDataSet drei Dateien), die auf dem Server residieren müssen. Das Ziel ist es, den Verbindungsaufbau wie die Konfiguration ohne nicht visuelle Komponenten oder ein Alias à la BDE zu ermöglichen. Alle Parameter sollen transparent und direkt im Code erscheinen.

Als wichtigen Parameter lässt sich die Datenbank auch explizit im Code hinzufügen:

```
Params.Add('Database=milo2:D:\ftech\wsc\umlbank.gdb');
```

Wir aber ersetzen diesen Teil durch das Einlesen besagter Datei. Die folgende Routine PostUser erstellt nun eine `TSQLConnection` mit den benötigten Parametern, welche wiederum mit einem DataSet verknüpft wird. Nach wie vor besteht die Möglichkeit, mit der Methode `LoadParamsOnConnect` die Parameter aus einer Datei, wie z. B. unter Linux *dbxConnections,* zu beziehen.

Aber wir wollen so schnell und autonom wie möglich sein. Anschließend speichert das DataSet mit einer Insert-Operation die gewünschten Daten. Sicherheitsaspekte lassen sich hier der Einfachheit halber nicht berücksichtigen. Unter Kylix heißt der Library Name *libsqlib.so* und die VendorLib *libgds.so*.

```
procedure TVCLScanner.PostUser(const E-Mail, FirstName,
                                    LastName : WideString);
var
  Connection: TSQLConnection;
  DataSet: TSQLDataSet;
  logdate: String[15];
begin
  Connection:= TSQLConnection.Create(nil);
  with Connection do begin
    ConnectionName:= 'VCLScanner';
    DriverName:= 'INTERBASE';
    LibraryName:= 'dbexpint.dll';
    VendorLib:= 'GDS32.DLL';
    GetDriverFunc:= 'getSQLDriverINTERBASE';
    Params.Add('User_Name=SYSDBA');
    Params.Add('Password=masterkey');
    with TWebModule1.create(NIL) do begin
      getFile_DataBasePath;
      Params.Add(dbPath);
      free;
    end;
    LoginPrompt:= False;
    Open;
  end;
```

```
  logdate:= dateTimetostr(now);
  DataSet:= TSQLDataSet.Create(nil);
  with DataSet do begin
    SQLConnection:= Connection;
    CommandText:=
      Format('insert into KINGS values("%d","%s", "%s", "%s",
        "%s")',[10, E-Mail,FirstName,LastName, logdate]);
    try
      ExecSQL;
    except
      raise
    end;
  end; //dataSet
  Connection.Close;
  DataSet.Free;
  Connection.Free;
end;
```

Nachdem der Webservice die Daten entfernt auf dem InterBase-Server gespeichert hat, ist nun die Darstellung auf dem Browser an der Reihe. Das Webmodul sucht die Instanz mit dem zugehörigen Aktionselement und löst die passende Ereignisbehandlungsroutine für `OnAction` aus, um eine Antwort auf die Anforderung zu generieren und zu senden. Die Hauptaufgabe von `Request` ist also, aus dem URL-String verschiedene Informationen, vor allem die `Pathinfo` und den Abfrage-Parameter, auszufiltern, in unserem Fall das `runselect`:

```
http://milo2/scripts/vclscannerserver.exe/runselect
```

Unser Eventhandler (auch Actionhandler genannt) lässt sich also mit der `Pathinfo` `runselect` verbinden. Im Action-Editor entspricht deshalb `runselect` dem `Pathinfo` (Pfadenteil).

Das zweite Objekt `Response` des Actionhandlers ist eine TWebResponse-Instanz, die von der Ereignisbehandlungsroutine ausgefüllt wird, d. h., der generierte HTML-Inhalt wird durch `Response` an den Webserver übergeben. Der Parameter `Handled` zeigt schlussendlich an, ob das TWebActionItem-Objekt die Bearbeitung der Anforderung abgeschlossen hat.

Wenn man die Sourcen durchstöbert, kommt man zum schlichten Schluss: Es gibt keine statischen HTML-Vorlagen. Das Ergebnis wird durchwegs dynamisch mittels `response.content` dem Webserver übergeben. Das folgende Sequenzdiagramm zeigt die beiden Szenarios als Use Case <Speichern der gesammelten Daten> und <Abfragen der Technikerliste> auf einen Blick (Abb. 3.14).

Sobald die GET-Botschaft vom Browser kommt, generieren die TDataSetTableProducer-Objekte die Datenquelle intern. Die Eigenschaft `Query` des `QueryTableProducer` ist mit der Eigenschaft `DataSet` identisch, allerdings verwendet `Query` nicht die Basisklasse `TDataSet`, sondern eine eigene.

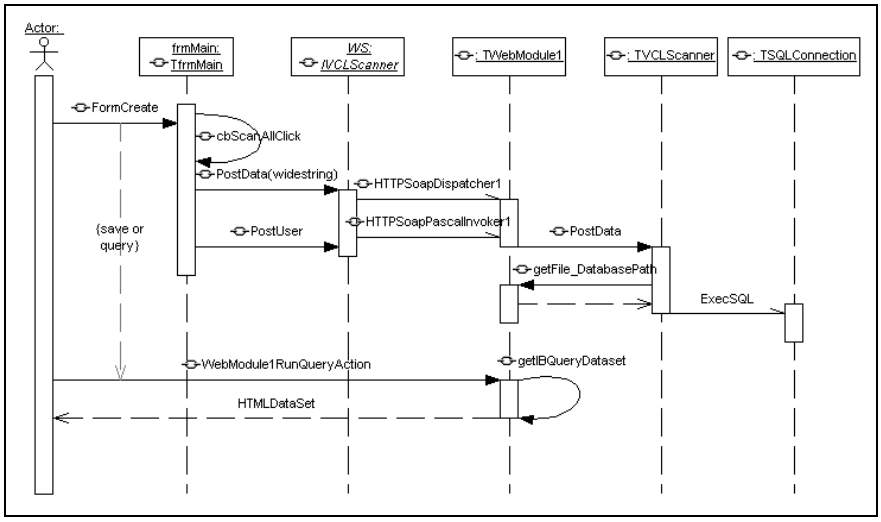

Abb. 3.14: Der Speicherfluss und die Abfrage im SEQ

Auch hier hat man wieder konsequent auf native dbExpress gesetzt, sodass im Webmodul keine weiteren Komponenten mehr ersichtlich sind:

Abb. 3.15: Das Gerüst im Webservice

Kommen wir zur eigentlichen Abfrage von `TSQLDataSet`, die ich zur besseren Strukturierung in eine eigene Funktion auslagere, welche ein DataSet zurückliefert, das sogar kompatibel zum TableProducer DataSet ist.

Ein `TSQLDataSet` ist ressourcenschonend und schnell, hat aber seinen Preis. Da die Komponente keine Kopien der Daten gepuffert im Speicher verwaltet, ist die Menge unidirektional und nicht schreibbar, hat also in einem `DBGrid` nichts zu suchen ;).

```
procedure TWebModule1.WebModule1RunQueryAction(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                                       var Handled: Boolean);
begin
getFile_DatabasePath;
  with DataSetTableProducer1 do begin
    try
      dataSet:= getIBQueryDataset;
      response.Content:= content;
    except
      Response.Content:= Format('<html><body><b>database
                           "%s" or query not found!' +
                           '</b></body></html>',[dbpath]);
    end;
  end;
  Connection.Close;
  myDataSet.Free;
  Connection.Free;
end;
```

Wenn schlussendlich ein Client den Dienst nutzt, hat man durch die sprachübergreifende SOAP-Spezifikation freie Wahl. Bevor ich am Schluss noch exemplarisch den Aufruf in den drei Sprachen Delphi, Java und C# zeige, möchte ich kurz ein Geheimnis des hexadezimalen Hexenwerkes lüften. Wer verpackt eigentlich die ganze Datenfülle im XML-Envelope?

Es ist die Komponente `THTTPRIO`, die bei genauem Debuggen (Abb. 3.16) offenbart, dass die CLX mithilfe von `TOPToSoapDomConvert` als `IOPConvert` das eigentliche Marshaling, d. h. Verpacken und Auspacken von Parametern mit den zugehörigen Funktionsaufrufen umsetzt. Als Parser setzt man DOM in Abhängigkeit der Library ein.

Klassen, die `IOPConvert` implementieren, benötigen zwingend die RTTI des Invokable Interfaces, womit die Compiler Direktive `{$M+}` wieder ins Spiel kommt. Das eigentliche Konvertieren und Interpretieren dieser „transportfähigen Strings“, welche ja codierte Methodenaufrufe mit Schnittstellen darstellen, genau das ist die Hexerei.

Der Aufruf aus Delphi basiert auf der Schnittstellen-Instanz, die nach erfolgreichem Abfragen des unterstützten Interfaces den wohlverdienten Zeiger erhält:

```
Var
WS: IVCLScanner;
WS:= HTTPRIO1 as IVCLScanner;
WS.PostData(reFinalResults.Text,CRC);
```

In Java und C# sind die Verhältnisse nicht unähnlich, bei Java wird der URL im Sinne des Schnittstellenzugriffs gleich mit dem Konstruktor übergeben, was bei Delphi in der Komponente als Eigenschaft erfolgt, womit wieder mal gesagt ist, dass die Mutter aller wohlerzogenen Sprachen Pascal ist ;).

```
IVCLScannerStub stub = new IVCLScannerStub(new URL
      "http://milo2/scripts/vclscanner.exe/soap/IVCLScanner"));
  stub.PostData(reFinalResults.Text,CRC);

IVCLScanner.IVCLScannerservice objIVCLScanner = new
                            IVCLScanner.IVCLScannerservice();
        bool booScanner=
           objIVCLScanner.PostData(reFinalResults.Text,CRC);
```

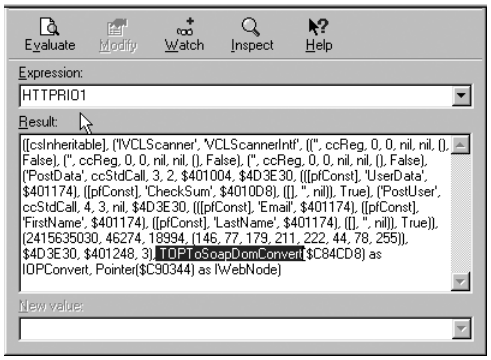

Abb. 3.16: Der Client-Aufruf vor dem Verpacken

3.3.3 Container Whole-Part

Ein verarbeitungsorientiertes Architekturmuster (Schnittstelle für Gesamtfunktionalität).

Zweck

Eine zusammenfassende Komponente, das Gesamtobjekt, kapselt die Teilobjekte, steuert deren Kooperation und stellt eine Schnittstelle als Whole-Part für die Gesamtfunktionalität der semantischen Einheit zurVerfügung.

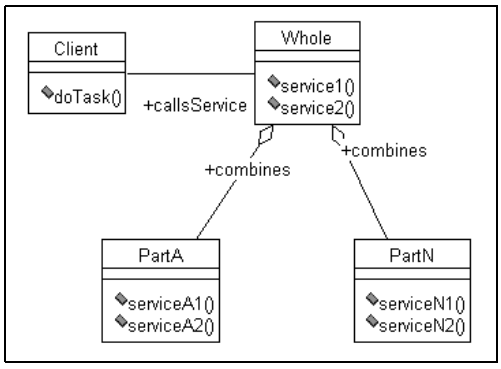

Abb. 3.17: Der Container hält die Teile

Motivation

In fast jedem Softwaresystem gibt es Objekte, die aus anderen zusammengesetzt sind. Mit einem Container können Sie auf die eingebauten Klassen des Containers bzw. den übergeordneten Objekten der Komponente zugreifen. Mit dem Begriff Container ist auch eine Kombination von zwei Pattern (Composite und Iterator) verbunden. Ich betrachte hier vor allem den Container aus der Architektursicht des Whole-Part Pattern.

So ein Container kann aus zusammengesetzten Steuerelementen bestehen. Er beinhaltet mehrere Komponentenklassen, welche die inhärente Fähigkeit haben, andere Komponenten zu enthalten. Beispiele hierfür sind u. a. `TForm`, `TPanel`, `TControlBar` und `TGroupBox`. Ein Container ist den in ihr enthaltenen Komponenten übergeordnet.

Sicher haben Sie im Zuge der Objektorientierung auch schon vom Exportieren von Klassen aus DLLs geträumt. Die Ruhe vor dem Schirm sozusagen. Mithilfe von Schnittstellenroutinen lässt sich dieser Wunsch erfüllen. Zudem erleben wir einen Hauch davon, wie COM im Prinzip funktioniert.[3]

Diese DLL als Container ist Bestandteil des TRadeRoboter, es handelt sich um die Datei *income.dll*, die ich nun Schritt für Schritt in das Grundprinzip, sprich in die Philosophie, des Container-Musters einführen will.

Das Muster repräsentiert ein exportiertes Gesamtobjekt, in meinem Fall eine Referenz auf die DLL, das aus Teilobjekten innerhalb der DLL besteht. Das Gesamtobjekt koordiniert die Teilobjekte und erhält ein Resultat aus der Zusammenarbeit der angebotenen Dienste der Teilobjekte zurück. Das Gesamtobjekt kann auch Dienste anbieten, ohne dass es auf die Teilobjekte zurückgreifen muss.

Verwendung

Als Erstes benötigen wir eine DLL als Service, die exemplarisch den Zinseszins als Resultat der einzelnen Dienste berechnen kann. Unser Ziel ist es, eine Referenz zu exportieren, sodass andere Anwendungen auf die gewünschten Methoden der DLL zugreifen können. Eigentlich kann Object Pascal kein Objekt aus einer DLL exportieren, außer wir setzen virtuelle Methoden ein. Ich bezeichne diese OO-Technik seit Jahren als DLL+.

Jedes Objekt führt einen Zeiger auf die eigene virtuelle Methodentabelle (VMT) mit, sodass sozusagen durch die Hintertüre die aufrufende Anwendung in den Besitz der Adresse der virtuellen Methode kommt.

Die DLL besteht also aus zwei Units:

1. Die Library *income.dpr* beinhaltet als Gesamtobjekt die eigentlichen Methoden und die zu exportierende Referenz als eigene Funktion.
2. Die Unit *income1.pas* beinhaltet die virtuellen/abstrakten Methoden als Schnittstelle.

Die Unit *income1.pas* ist fast wie eine Objektschnittstelle zu betrachten, welche abstrakte Methoden definiert, die von einer anderen Klasse, nämlich der Klasse `TIncomeReal`,

[3] Somit wird eine solch konstruierte DLL wieder salonfähig.

implementiert werden können. Die richtigen, normierten Schnittstellen werden auch wie Klassen deklariert. Nun, das Muster lässt sich auch mit Interfaces realisieren.

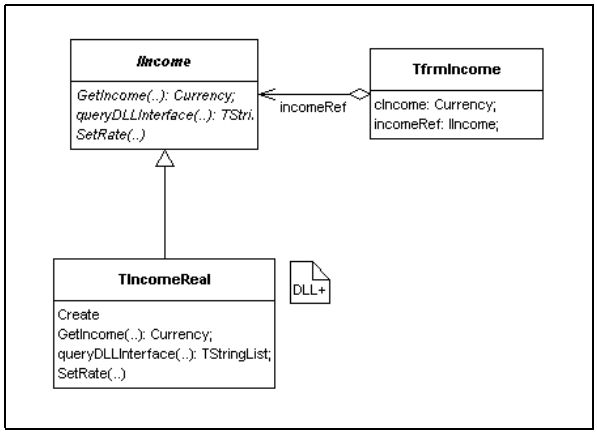

Abb. 3.18: Das CD der exportfähigen DLL

Schnittstellen haben keine Konstruktoren oder Destruktoren. Sie können nicht instanziert werden, ausgenommen durch Klassen, über die sich die Methoden der Schnittstelle implementieren lassen. Vom Konzept her (nicht aber von der Deklaration her) ist das unser Fall, denn *income1.pas* besteht nur aus diesem Rumpf:

```
unit income1;
interface
type
IIncome = class
public
  function GetIncome(const aNetto: Currency): Currency;
                           virtual; abstract;
  procedure SetRate(const aPercent : Integer; aYear: integer);
                           virtual; abstract;
end;
```

Ich komme nun zur eigentlichen Library *income.dpr*, die ja von der Schnittstelle, d. h. von der Klasse `IIncome`, erbt. Hier werden die effektiven Methoden implementiert und die Referenz auf die folgende Klasse `TIncomeReal` wird mithilfe der eigenen Funktion `CreateIncome` exportiert:

```
Type
TIncomeReal = class(IIncome)
  private
    FRate : Real;
  public
    constructor Create;
```

```
    function GetIncome(const aNetto: Currency): Currency;
override;
    function Power(X: Real; Y: Integer): Real;
    procedure SetRate(const aPercent: Integer; aYear: integer);
override;
end;
```

Der eigentliche Export befindet sich in einer eigenen Funktion, die als Rückgabewert die Instanz über den Konstruktor der Klasse zurückgibt. In der „Export Table“ der DLL erscheint die Funktion `CreateIncome` als einziger Entry Point:

```
function CreateIncome: TIncomeReal; stdcall;
begin
  Result:= TIncomeReal.Create;
end;
{ Export der Schnittstellenroutine }
{-------------------------------------------------------------}
  exports CreateIncome resident;
  begin {nur ein fake} end.
```

Wenn mehrere Referenzen nötig sind, zeigt Abb. 3.19 die Möglichkeit, mit einem vorgeschalteten Iterator die einzelnen Objekte verwalten zu können. Dies als Kombination von Iterator und Composite. Wenn Sie eine wichtige Struktur oder einen Datentyp in Delphi definieren, sollten Sie immer auch einen Zeiger auf diesen Datentyp definieren. Wobei Zeiger auf Prozeduren niemals kompatibel zu Zeigern auf Methoden sind, die sich innerhalb einer Struktur befinden können.

> *Viele höhere Programmiertechniken, wie etwa verkettete Listen dynamisch verwalteter Datensätze oder Hash-Tabellen, erfordern die Verwendung solcher Zeiger anstatt der Variablen selbst.*

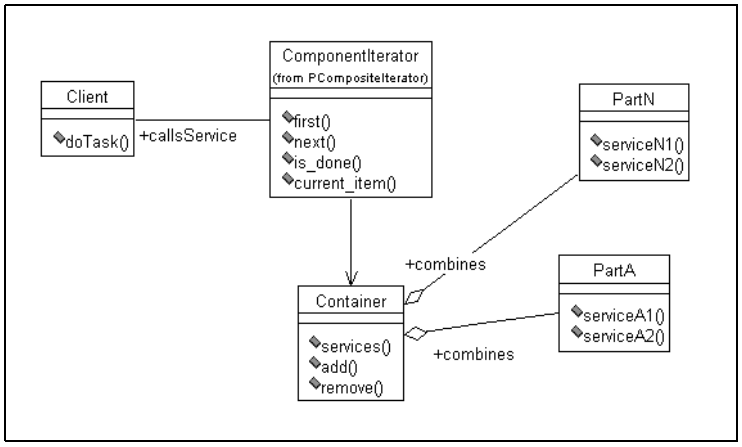

Abb. 3.19: Das Verwalten der Referenzen

So ist es z. B. möglich, einem Aufrufer gleich den Zeiger auf die Adressliste zurückzugeben oder zusätzlich eine Methode zu aktivieren. Hier schnell ein Beispiel aus der Praxis unserer eigenen Komponenten:

```
function TPerson.CreateAndReadAdressList: TAdressList
begin
  result:=TAdressList.create;
  result.DBReadAllToObject(self,nil)
end;
```

Nachdem die DLL als *income.dll* compiliert wurde (achten Sie auf die Groß- und Kleinschreibung bei der External Deklaration), baue ich den Client. Der Client befindet sich in der Unit *zinsView.pas* , d. h., der „Zinsserver“ lässt sich aus dem Form aufrufen.

Der Client benötigt vor allem die Datei *income1.dcu* in der `uses`-Anweisung, die somit statisch gelinkt ist. Also doch ein Haken, da eine normale DLL ohne weitere Dateien beim Aufrufer oder eben Client zum Einsatz kommt.

Nun, dies ist der Preis des Exportes mit DLL+, da sowohl die DLL als auch der Client die gleiche **Klassenstruktur** als abstrakte Klasse definieren müssen, ähnlich einer Typenbibliothek unter COM. Nur so kann ein Eintrag in die virtuelle Methodentabelle erfolgen. Instanziert wird aber nicht `IIncome`, sondern der Nachfolger `TIncomeReal`:

```
unit zinsView;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
       Controls, Forms, Dialogs, StdCtrls, Buttons, Income1;
type
  TfrmIncome = class(TForm)
  private
    incomeRef: IIncome;
......
  end;

var
  frmIncome: TfrmIncome;
implementation
{$R *.DFM}
  function CreateIncome: TIncomeReal; stdcall;
                                external('Income.dll');

procedure TfrmIncome.FormCreate(Sender: TObject);
begin
  incomeRef:=createIncome; //Objektreferenz
end;
```

```
begin
  incomeRef.SetRate(strToInt(edtZins.text),
                    strToInt(edtJahre.text));
  try
    cIncome:= incomeRef.GetIncome(StrToFloat(edtBetrag.Text));
  except on EConvertError do
    raise exception.create('must be string');
  end;
 edtBetrag.text:= Format('%m',[cIncome]);
end;
```

Ein weiterer Blick auf das Sequenzdiagramm bestätigt unsere Absicht. Der Client erhält in der Methode `FormCreate` eine Objektreferenz, die unsere DLL im Prozessraum exportiert. Die Koordination der Teilobjekte erledigt die DLL, sodass ich nur am Resultat als Service interessiert bin. Ich habe auch keinen Zugriff auf die Teilobjekte.

Dieses Exportieren hat klar den Vorteil, dass künftig nicht jede einzelne Methode oder Funktion einer DLL im Client deklariert werden muss, das Arbeiten mit Referenzen ist flexibler und führt zu modularer Architektur und zu strukturiertem Code.

Eine DLL wird sozusagen als Service Provider betrachtet, der Zugriff auf die Teilobjekte oder lose Funktionen ist mir untersagt. Was uns im Vergleich zu COM natürlich fehlt, ist der ganze Prozess der Registrierung und schlussendlich der Aufruf über die **Rechnergrenzen**, d. h. der Netzfähigkeit wie bei DCOM hinweg.

> *Bei COM oder DCOM wird ja ein Aufruf via Netzwerk auf den entfernten Rechner geleitet und dort das Server-Objekt mit einer CLSID_X lokalisiert. Nach dem Start gibt das entfernte Server-Objekt einen Zeiger auf sein Interface mittels der COM-Bibliothek an den Client zurück.*

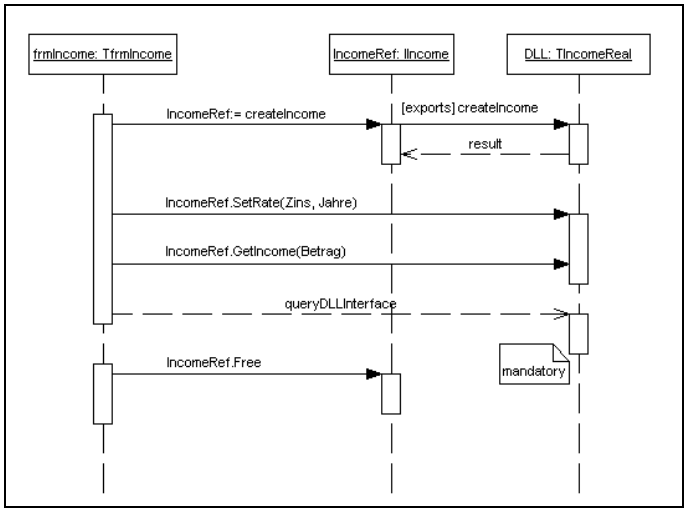

Abb. 3.20: Das Gesamtobjekt als DLL-Referenz koordiniert

Eine nützliche Erweiterung ist der Einbau einer Liste, die mit einem standardisierten Aufruf wie `incomeRef.QueryInterface()` jedem Aufrufer oder Client zeigt, welche Methoden die DLL zur Verfügung stellt. Sie sehen, es gibt gewisse Designprinzipien, die so oder ähnlich in den digitalen Kammern der COM-Ingenieure auch mal begonnen haben.

3.3.4 Layers

Ein kombiniertes Architekturmuster (Strukturierung von Anwendungen in Ebenen).

Zweck

Mit den Layers lassen sich Anwendungen strukturieren, die in Gruppen von Teilaufgaben zerlegbar sind und in denen diese Gruppen auf verschiedenen Abstraktionsebenen liegen. Das System benötigt eine Zerlegung.

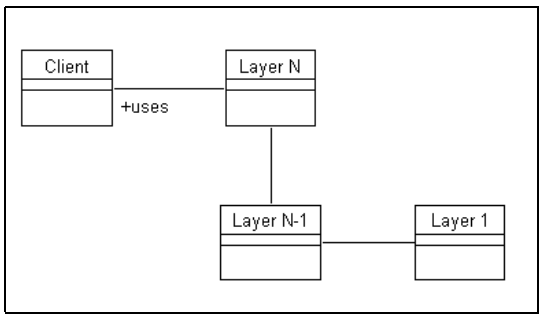

Abb. 3.21: Layers: das Schichtendenken par excellence

Motivation

Es ist das wohl bekannteste Architekturmuster, das vor allem eine vertikale Struktur propagiert, diese Schichten aber auch später horizontal verteilen kann. Bereits in der Analyse[iv] macht man sich Gedanken, wie denn ein grobes Schichtenmodell des künftigen Systems daherkommt. Diese Gedanken resultieren meist in einem ersten Paketdiagramm, das vor allem die Fachdomänen in die Pakete und deren Abhängigkeit miteinbezieht.[v]

Systeme, welche eine Mischung von Aufgaben auf verschiedenen Ebenen beinhalten, haben meist auch ein Gegenüber, das mit diesen Aufgaben horizontal kommuniziert. Bestes Beispiel dafür ist das ISO-OSI-Referenzmodell, das als 7-Schichtenmodell konzipiert wurde.

Diese Systeme benötigen dann auch eine horizontale, physische Struktur, die rechtwinklig zur vertikalen (logischen) Zerlegung ist (siehe Architekturschema Abb. 3.2/3.3). Nehmen wir als Beispiel Schicht 4 des OSI-Modells, die als Transportschicht die Nachrichten in Pakete zerlegt **und** eine Auslieferung der Pakete sicherstellt. Das Zerlegen in Pakete gehört zur vertikalen Struktur, das Ausliefern hingegen bedingt auch eine horizontale Struktur, da ein Gegenüber vorhanden ist.

Konkret bedeutet die Strukturierung auch, dass jede Schicht zumindest ein Modul verkörpert. Als Standardschichten haben sich folgende Bezeichner etabliert:

- **Visualisation**: Hier findet bei Thin-Clients oder Terminal-Servern nur noch die Visualisierung und Input-Steuerung statt (bei einem Terminal).
- **Security**: Der Einstiegspunkt des Clients mit der entsprechenden Autorisierung.
- **View**: Hier erfolgt die spezifische Aufbereitung der Daten als Präsentation, allgemein als Fenster, Formular oder Form bekannt.
- **Controller Logic**: In dieser Schicht findet die Programmkontrolle statt, wenn nötig lassen sich hier die Ereignisse und GUI-Signale kanalisieren, also allgemein ein Steuern des Benutzerverhaltens. In Webtechnologien findet man hier auch den Session State des Benutzers.
- **Business-Logic**: In dieser Schicht ist die eigentliche Programmlogik implementiert, die ja unabhängig von deren Präsentation ist. Auch der Datenzugriff hat hier sein Zuhause.
- **Data**: Diese Schicht ist eindeutig dem physischen Datenmodell zugewiesen und ist der einzige Layer, wo Daten persistent gespeichert sind. Aufgrund von Legacy oder Systemoptimierungen kann diese Schicht auch Hosttransaktionen oder Stored Procedures enthalten.
- **Periphery**: Diese unterste Schicht regelt den Zugriff auf externe Hardwareschnittstellen, die im Zusammenhang mit dem Data-System auftreten können, wie ein Gateway oder ein Massenspeicher im Archivierungssystem.

Mehrschichtige Anwendungen greifen gemeinsam auf Daten zu und kommunizieren über ein lokales Netzwerk oder auch das Internet miteinander. Wenn diese horizontale Struktur gegeben ist, sind Zugriffskomponenten als Middleware meistens mit von der Partie.

Verwendung

Die Abbildung 3.22 zeigt eine vierschichtige Anwendung. Die Logikschicht (nach dem Controller), die Datenbankinformationen bearbeitet, befindet sich mit dem Controller auf einem separaten System, einer horizontalen Schicht. Diese Zwischenschicht zentralisiert die Logik, welche die Datenbankzugriffe steuert, sodass eine zentrale Steuerung über die Datenbeziehungen entsteht.

Auf diese Weise können mehrere Client-Anwendungen dieselben Daten verwenden, während gleichzeitig eine konsistente Datenlogik sichergestellt ist. Außerdem werden damit kleinere Client-Anwendungen möglich, weil ein Großteil der Verarbeitung in die Zwischenschicht verschoben wird.

Diese kleineren Client-Anwendungen sind einfacher zu installieren, zu konfigurieren und zu verwalten. Darüber hinaus können mehrschichtige Anwendungen auch die Leistung verbessern, indem sie die Datenverarbeitung auf mehrere Systeme verteilen.

Es gibt verschiedene Typen von Verbindungskomponenten, die eine Client-Datenmenge mit einem Anwendungsserver verbinden können. Bei allen handelt es sich um Ableitungen von `TCustomRemoteServer`. Sie unterscheiden sich hauptsächlich im verwendeten Kommunikationsprotokoll (TCP/IP, HTTP, DCOM, SOAP oder CORBA). Verknüpfen

Sie die Client-Datenmenge mit ihrer Verbindungskomponente, indem Sie die Eigenschaft `RemoteServer` setzen.

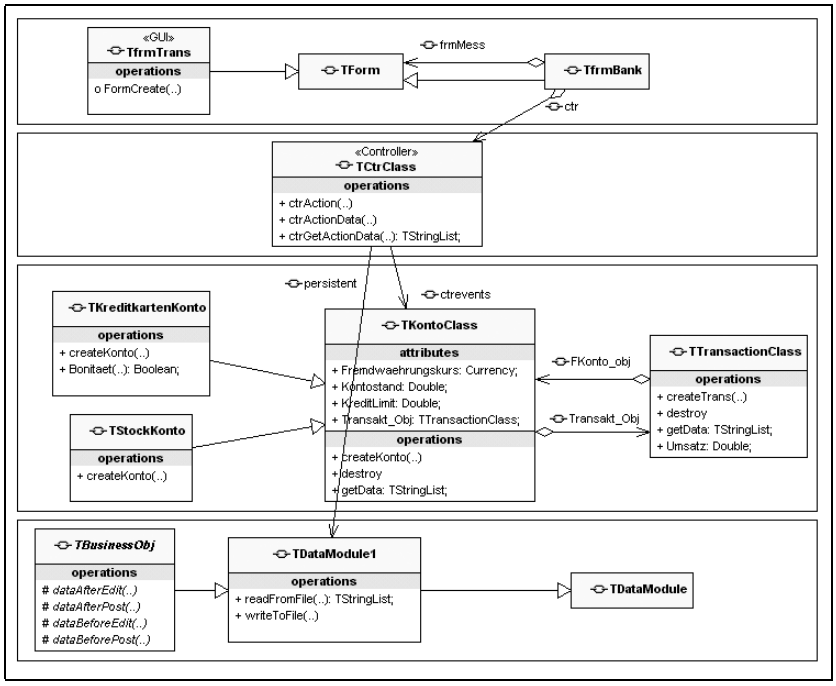

Abb. 3.22: Die logische Schichtung im ersten Wurf

Wenn ich die erste logische Architektur in Abb. 3.22 nun in eine physische horizontale Struktur verschiebe oder umbaue, zeigt sich in Abb. 3.23 konkret, dass die logischen Schichten nicht nur auf einem einzigen System installiert und konfiguriert sind.

Als Codeausschnitt sei der Zuriff auf einen SQL-Server gezeigt, da sich InterBase- und SQL-Server im Einsatz als gleichwertig erwiesen haben.

Wenn es sich bei der Standardschnittstelle um eine Dispatch-Schnittstelle handelt, dann wird keine Creator-Class für die CoClass generiert. Stattdessen lässt sich die globale Funktion CreateOleObject aufrufen, wobei man das GUID für die CoClass übergibt (für dieses GUID ist eine Konstante am Anfang der _TLB-Unit definiert). `CreateOleObject` gibt dann einen IDispatch-Zeiger für die Schnittstelle zurück.

```
procedure TDataModule1.getSQL;
var sqlOle: variant;
  data: variant;
begin
  try
    sqlOle:=createOleObject('sqlole.sqlServer');
  except
```

```
   ShowMessage('Could not start SQL-DMO');
    Exit;
  end;
  sqlOle.connect('dynatxt_SRV1','sa','dynatxt');
  with listSql do begin
    items.add(sqlOle.application.name);
    data:=sqlOle.databases;
    items.add(data.parent.name);
  end;
 sqlOle.close;
end;
```

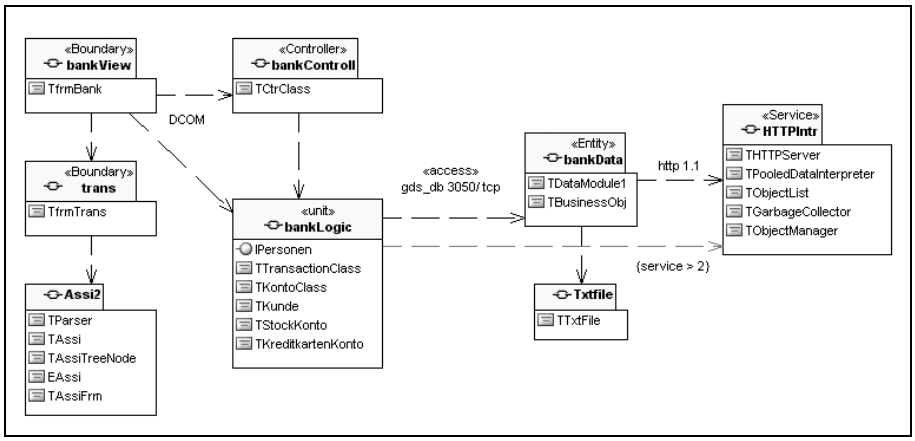

Abb. 3.23: Die physische Struktur im zweiten Wurf

Eine weitere reale Verwendung, die beim Studium der Sourcen eine vertikale wie horizontale Schichtung aufweist, ist Samba!

Seit diversen Jahren schon ist Samba, ein SMB-Server für viele Unix-Plattformen, ein Tool zur schnellen und einfachen Integration von Linux und Windows. SMB unter NT steht für Server Message Block, ein Dateifreigabeprotokoll, das Systemen den transparenten Zugriff auf Dateien ermöglicht, die auf Remote-Systemen gespeichert sind.

Integration meint bei Samba Folgendes: Linux-Server stellen NT- oder Win-Maschinen Datei- und Druckerdienste zu Verfügung, und zwar gemäß den Protokollen von NT (SMB). Es wird also ein Teil der NT-Dienste, inklusive WINS, auf Linux portiert. Somit passt sich Linux NT an und es wird keine weitere Client-Software mehr auf den NT-Rechnern benötigt. Der Linux-Server erscheint einfach wie ein „normaler“ Win-Rechner im Netz.

> *WINS ist ein Dienst zur Namensauflösung, der die Computernamen in Windows-Netzwerken auf IP-Adressen in einer so genannten routed-Umgebung abbildet. Ein WINS-Server bearbeitet Namensregistrierung, Abfragen und Freigaben.*

Das Geniale von Samba ist, dass der Linux-Rechner unter NT wie eine weitere Maschine oder ein weiterer Laufwerksbuchstabe angesprochen wird, ohne dass ein Tool unter NT erforderlich ist. Die Verwaltung erfolgt also transparent und zentral unter Linux. Das Erstaunliche ist auch, dass die Performance besser und stabiler als bei einem NT-eigenen Server ist.

3.3.5 Master-Slave

Ein nachrichtenorientiertes Architekturmuster (fehlertolerante parallele Systeme).

Zweck

Ermögliche einem Master, der die zu lösende Aufgabe zwischen gleichberechtigten Slaves aufteilt, die fehlertoleranten und genauen Ergebnisse zu einem Gesamtergebnis zusammenzuführen.

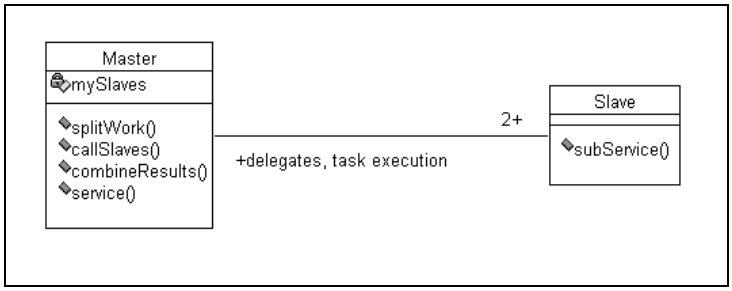

Abb. 3.24: Der Master verteilt die Aufgaben

Motivation

Meistens sucht man nach einer Lösung, die ein Optimum darstellt. An diesem Optimum sollen mehrere Lösungssucher beteiligt sein, die voneinander abhängig oder unabhängig sind. Für die Modellierung bspw. von Geschäftsprozessen sind parallele Vorgänge von Bedeutung, da man hier oft unnötige Sequenzen und Bedingungen oder sogar Leerläufe aufdecken kann. Und man hofft auf eine schnelle und fehlertolerante Ausführung, da auch beim Master-Slave gilt: „Viele Hände schaffen schnell ein Ende."

Für die Parallelität muss es nicht gerade OCCAM oder eine andere parallele Programmiersprache sein, Threads genügen auch. Dadurch lassen sich Flaschenhälse vermeiden. In einem Programm mit nur einem Thread muss man die gesamte Ausführung unterbrechen, wenn auf den Abschluss eines langsamen Prozesses gewartet wird. Solche Prozesse sind das Schreiben von Dateien auf Datenträger, die Kommunikation mit anderen Rechnern oder die Anzeige von Multimediadaten. Die CPU befindet sich bis zum Ende solcher Prozesse im Leerlauf.

Wenn eine Anwendung dagegen mehrere Threads umfasst, kann die Ausführung der anderen Threads fortgesetzt werden, während ein Thread auf das Ergebnis eines langsa-

men Prozesses wartet. In meinem Beispiel soll der schnellste Sortierer mit Threads gefunden werden.

Ein weiterer Vorteil des Master-Slave ist die Strukturierung des Programmverhaltens. Oft lässt sich das Verhalten eines Programms in mehreren parallelen Prozessen organisieren, die unabhängig voneinander funktionieren. Mithilfe von Threads kann die Ausführung eines bestimmten Quelltextabschnitts für jeden dieser parallelen Zweige gleichzeitig gestartet werden. Ebenso lassen sich den verschiedenen Programm-Tasks unterschiedliche Prioritäten zuordnen, damit kritische Tasks mehr CPU-Zeit erhalten.

Verwendung

Gesucht wird aus einer anfangs zufälligen Verteilung von Daten der schnellste Sortieralgorithmus, den die Anwendung erst noch grafisch aufbereitet. Das Beispiel stammt von Borland und ist mit Delphi wie mit dem Builder im Verzeichnis Demos\Threads nachvollziehbar.

Wenn das Master-Slave-Muster auch als Architekturmuster gilt, dann hat das gemäß der Plattform und einem zugehörigen Framework folgende zwei Gründe:

- Multiprocessing. Wenn eine Anwendung auf einem System mit mehreren Prozessoren läuft, kann die Leistung gesteigert werden, indem man die Verarbeitung auf einzelne Threads verteilt und diese gleichzeitig durch einen Master auf verschiedenen Prozessoren ausführen lässt.
- Eingaben mehrerer Datenübertragungsgeräte verarbeiten. Verschiedene Tasks werden nach ihrer Priorität unterschieden. Bspw. übernimmt ein Thread hoher Priorität zeitkritische Aufgaben.

Die Koordination zu vieler Threads verbraucht aber beträchtliche CPU-Zeit. Auf einem Ein-Prozessor-System stellen 16 aktive Threads die praktikable Obergrenze dar.

> *Nicht alle Betriebssysteme bieten jedoch echtes Multiprocessing, auch wenn die zugrunde liegende Hardware dies unterstützt. So wird beispielsweise unter Windows 9x Multiprocessing lediglich simuliert.*

Das Beispiel erzeugt einen Nachkommen von `TThread`, um einen Ausführungs-Thread in einer mehrschichtigen Anwendung darzustellen. Jede weitere Instanz eines TThread-Nachkommen bildet einen neuen Ablaufstrang. Wenn eine Anwendung über mehrere solcher Threads verfügt, wird sie als Multithread-Anwendung bezeichnet.

Beim Start einer Anwendung werden die einzelnen Sortierer in Abb. 3.25 zur Ausführung in den Speicher geladen. In diesem Moment wird sie zu einem Prozess, der nach dem `Execute` (s. Abb. 3.26) einen oder mehrere Threads umfasst, die jeweils Daten, den Code und weitere Systemressourcen für das Programm enthalten.

Ein Thread führt einen bestimmten Teil einer Anwendung aus und erhält hierfür vom Betriebssystem CPU-Zeit zugewiesen. Alle Threads eines Prozesses greifen auf denselben Adressbereich und auf die „nötigen“ globalen Prozessvariablen zu.

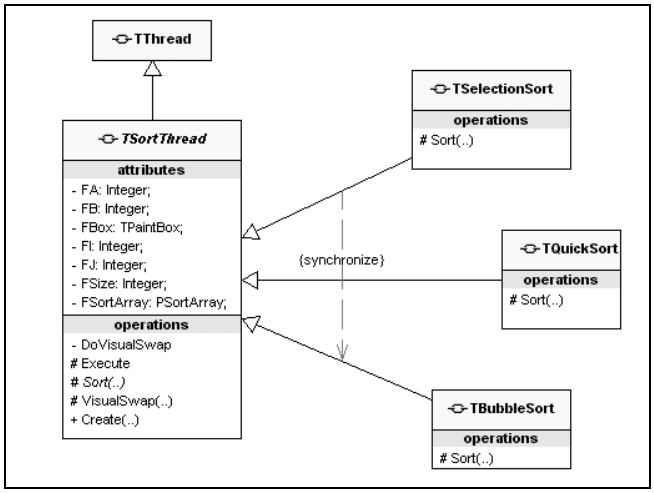

Abb. 3.25: Jeder Sort hat seinen eigenen Ablaufstrang

Das Muster erfüllt in diesem Kontext folgende Kriterien:

- Parallelverarbeitung: Das Endergebnis entspricht dem besten Sortierer mit hoher Geschwindigkeit oder setzt sich aus den Teilergebnissen zusammen.
- Fehlertoleranz: Dies ist ein oft unterschätzter Punkt, der meint, auch wenn ein Slave oder ein Prozessor ausfällt, kann der Master dem Client garantieren, die Daten sortiert zu erhalten.
- Genauigkeit: Der Dienst lässt sich an unterschiedliche Slaves delegieren, die durch Vergleich bestätigen können, dass keine Ungenauigkeiten entstehen. Jeder Sortierer muss zur gleichen Reihenfolge gelangen. Dieses Kriterium erfolgt meistens am Schluss zur Bestätigung.

Diese Kriterien lassen sich meist im Konstruktor festlegen, denn die Fehlertoleranz basiert auf dem Definieren der Abbruchbedingung und die Rechengenauigkeit auf dem abschließenden Vergleich nach der Abbruchbedingung:

```
constructor TSortThread.Create(Box: TPaintBox;
                               var SortArray: array of Integer);
begin
  FBox:= Box;
  FSortArray := @SortArray;
  FSize:= High(SortArray) - Low(SortArray) + 1;
  FreeOnTerminate:= True;
  inherited Create(False);
end;
```

Im Normalfall setzt man `FreeOnTerminate` auf True. Es gibt jedoch Situationen, in denen die Beendigung eines Threads bezüglich Parallelverarbeitung mit anderen Threads koordiniert werden muss. Das ist beispielsweise der Fall, wenn man mit der Ausführung

eines Threads warten muss, bis ein anderer Thread einen bestimmten Ergebniswert geliefert hat.

> *Wenn eine rechenintensive Operation hohe Priorität erhält, kann dies die Ausführung der anderen Threads in der Anwendung blockieren. Daher sollten nur solche Threads höchste Prioritätsstufen erhalten, die den Großteil der Zeit auf externe Ereignisse warten.*

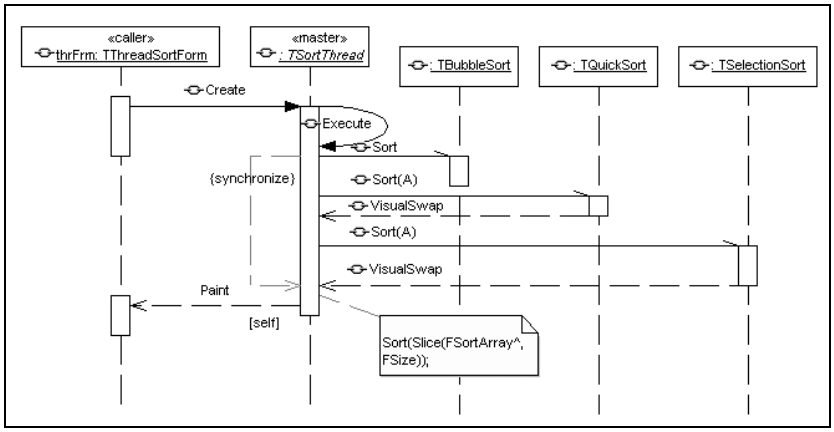

Abb. 3.26: Ein Ergebnis kann einzeln oder gesamt zurückkommen

Meiner Meinung nach ist das Hauptproblem die Synchronisation von Threads. Das Problem ist Folgendes: Wenn zwei Prozesse (Threads) abwechselnd eine aktive Systemroutine, z. B. eine `TList`, in einer Komponente aufrufen, muss diese Routine wissen, welcher **Thread** an welcher Stelle unterbrochen wurde.

> *Wenn mehrere Threads dieselben Ressourcen aktualisieren, ist unbedingt ein Synchronisieren erforderlich, das Konflikte unterbindet.*

Die meisten Methoden, die auf ein Objekt zugreifen und ein Formular aktualisieren, dürfen nur vom Hauptthread aus aufgerufen werden. Man kann auch ein Synchronisationsobjekt, wie z. B. `TMultiReadExclusiveWriteSynchronizer`, verwenden.

Den Vorteil, den wir z. B. in Linux durch die Verwendung von Threads (Subprozesse oder Programmfäden) haben, bedeutet auch, dass ein laufender Thread seine Aktivität mitten in der Routine abbrechen kann, während der nächste aktive Thread versucht, genau diese Routine aufzurufen.

Die meisten der API-Funktionen sind auf solche Fälle vorbereitet. Die Routinen sind derart aufwändig konzipiert, das ein erneuter Aufruf richtige Ergebnisse liefert, selbst wenn ein vorgängiger Aufruf eines anderen Threads noch nicht abgearbeitet wurde.

Auch sollte der vorhergehende und momentan inaktive Aufruf ebenfalls richtig fortgesetzt werden. Solch kompliziert realisierter Code wird als **reentrant** bezeichnet, was in etwa wiedereintrittsfähig bedeutet.

Dies würde dem Multithreading-Apartment-Modell entsprechen:[4]

> *Clients können die Methoden jedes Objekts von jedem Thread aus jederzeit aufrufen. Objekte können eine beliebige Anzahl von Threads gleichzeitig bearbeiten. Daher müssen die Objekte alle Instanzendaten und globalen Daten schützen (synchronisieren). Hierzu muss man kritische Abschnitte oder eine andere Form der Serialisierung verwenden.*

```
procedure TSortThread.VisualSwap(A, B, I, J: Integer);
begin
  FA:= A;
  FB:= B;
  FI:= I;
  FJ:= J;
  Synchronize(DoVisualSwap);
end;
```

Folgender Ausschnitt stammt aus der `Synchronize`-Methode, die in der Datei *classes.pas* mit den kritischen Abschnitten die Möglichkeit bietet, eine übergebene Methode zu synchronisieren und sie somit threadsafe zu machen:

```
procedure TThread.Synchronize(Method: TThreadMethod);
...
{$IFDEF MSWINDOWS}
    SyncProc.Signal:= CreateEvent(NIL, True, False, NIL);
    try
  {$ENDIF}
  {$IFDEF LINUX}
      FillChar(SyncProc, SizeOf(SyncProc), 0);
  {$ENDIF}
      EnterCriticalSection(ThreadLock);
      try
        FSynchronizeException:= NIL;
        FMethod:= Method;
        SyncProc.Thread:= Self;
        SyncList.Add(@SyncProc);
        ProcPosted:= True;
        if Assigned(WakeMainThread) then
          WakeMainThread(Self);

  {$IFDEF MSWINDOWS}
        LeaveCriticalSection(ThreadLock);
        try
```

[4] Immerhin gibt es danach kein kompliziertes Modell mehr.

```
          WaitForSingleObject(SyncProc.Signal, INFINITE);
        finally
          EnterCriticalSection(ThreadLock);
        end;
  {$ENDIF}
  {$IFDEF LINUX}
        pthread_cond_wait(SyncProc.Signal, ThreadLock);
  {$ENDIF}...
```

`Synchronize` löst den Aufruf einer bestimmten Methode aus, die vom Hauptthread ausgeführt werden soll. Der Thread wird unterbrochen, während die angegebene Methode im Hauptthread ausgeführt wird.

VCL ist sicher auch die Geburt eines Fensters eigen, die ja auch die Allozierung des Speichers beinhalten muss. Die VCL als Master und die Fenster als Slaves. Demzufolge soll eine Instanz auch mal ins Leben gerufen werden und eine Referenz als Zeiger zurückgeben. Tief in der API sollte doch die Speicherverwaltung ihre Arbeit verrichten und der *kernel32.dll* eine Nachricht übergeben. Diese Aussage will ich nun genauer untersuchen. Hier ist das Kernstück in Form einer Funktion aus der Unit `Forms.pas`:

```
function MakeObjectInstance(Method: TWndMethod):Pointer;
const
  BlockCode: array[1..2] of Byte = (
    $59, { POP ECX }
    $E9); { JMP StdWndProc }
  PageSize = 4096;
var
  Block: PInstanceBlock;
  Instance: PObjectInstance;
begin
  if InstFreeList = NIL then begin
    Block:= VirtualAlloc(NIL, PageSize, MEM_COMMIT,
    PAGE_EXECUTE_READWRITE);
    Block^.Next := InstBlockList;
    Move(BlockCode, Block^.Code, SizeOf(BlockCode));
    Block^.WndProcPtr:=
    Pointer(CalcJmpOffset(@Block^.Code[2],@StdWndProc));
    Instance := @Block^.Instances;
.....
```

So, nun wissen wir es genauer, eine Fenster-Instanz ist ein Zeiger auf einen Block, der mit mehreren Blöcken à 4 KBytes in einer Instanzenliste verwaltet wird.

Abschließend zeigt der Master auch noch seine Slaves in Aktion:

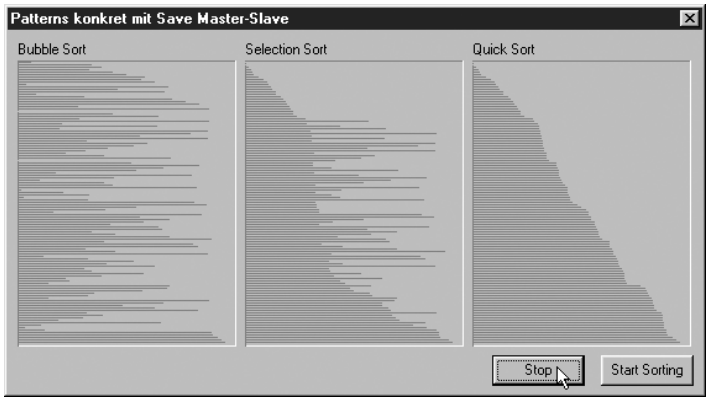

Abb. 3.27: Zwischenstop der drei Slaves

3.3.6 Microkernel

Ein verarbeitungsorientiertes Architekturmuster (trennt Kern von spezifischen Teilen).

Zweck

Die zusammenfassenden minimalen Funktionen sind von den erweiterten Funktionen und kundenspezifischen Teilen getrennt. Der Microkernel dient als Basis für solche Erweiterungen und ist für die Koordination zuständig.

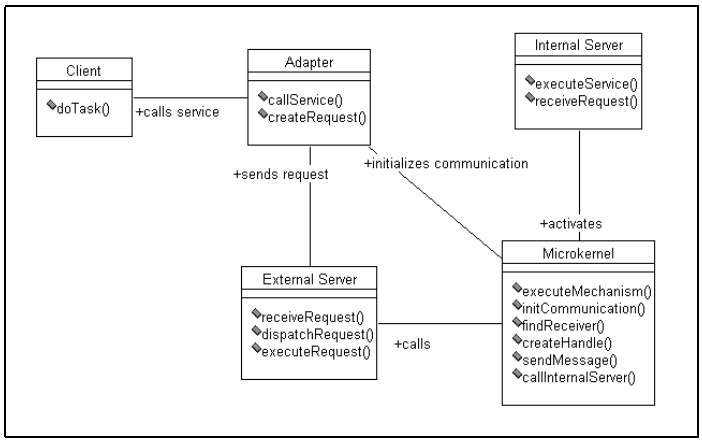

Abb. 3.28: Tief im Innern des Microkernel

Motivation

Bei diesem Muster kommt einem unweigerlich ein Betriebssystem in den Sinn.

Diese Komponente der Exekutive stellt das Innerste eines Betriebssystems dar, auf den Kernel bauen die übrigen Komponenten auf. Er besitzt Schnittstellen, über die höhere Funktionen auf die Hardware zugreifen können.

Eine seiner Hauptaufgaben ist es, die Ablaufsteuerung der Prozesse zu überwachen, den geeignetsten Prozess zu wählen und bei Bedarf einen so genannten Kontextwechsel durchzuführen. Außerdem kümmert er sich um die auftretenden Interrupts und Ausnahmen.

> *Da die Unterbrechungen unterschiedlich sind, lassen sie sich in verschiedene Interrupt-Ebenen (IRQL) einteilen, deren Palette von der höchsten Ebene über Spannungsausfall und Clock-Interrupt bis zu den Softwareunterbrechungen reicht.*

Diese Ereignisse lassen sich an eine Routine weiterleiten, die entweder vom Kernel selber oder von einer anderen Komponente geleitet sind.

Der Anwender kommt mit dem Kernel selbst nie in Berührung, spürt jedoch ständig dessen Existenz. Die dem Betriebssystem zugrunde liegenden Programme sind in ein Schichtenmodell angeordnet. Höhere Programmschichten basieren auf Funktionen der darunter liegenden Programmschichten. Von allen Schichten ist der Kernel die innerste Ebene. Somit dienen vor allem die Basisdienste des Kernel als Grundlage für die API-Funktionen der verschiedenen Subsysteme.

Verwendung

Als Verwendung dienen direkt die echten Betriebssysteme, die ja mit dem frei verfügbaren Code von Linux neue Interessenten angezogen haben.

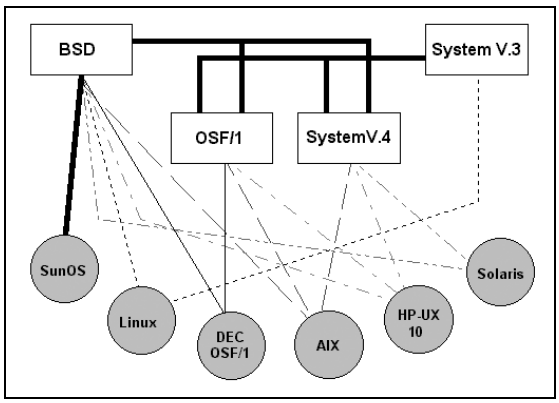

Abb. 3.29: Externe Server kommunizieren mit dem Microkernel

Bei der Struktur des Musters gehen die Ansichten auseinander: Auf der einen Seite wird das Muster mit dem Layer verglichen, sodass jeder Aufruf nur die darunter liegende Schicht erreichen kann. Jede Stufe kapselt also die darunter liegende Schicht und verhindert das „Überspringen“ einer Schicht. Dies erleichtert auch eine künftige Migration und

trennt somit die Strategie vom Mechanismus. Die Autoren der zweiten Struktur schlagen folgende fünf Komponenten vor:

- Clients Layer 1, extern
- Adapter Layer 2, extern
- Externer Server Layer 3, extern
- Microkernel Layer 4, intern
- Interner Server Layer 5, intern

Diplomatisch lässt sich sagen, dass der Microkernel eine Variante des Layers darstellt. Kernfunktionen, die nicht im Microkernel enthalten sind, weil die Größe und Stabilität darunter leiden könnte, implementiert man in internen Servern. Ein externer Server empfängt die Anforderungen der Clients, die nach der Analyse durch den Microkernel wieder als Antwort zu den Clients gelangen. Die Aufgabe des Adapters besteht in der Vermittlung zwischen den geforderten Diensten des Clients und der Weiterverarbeitung durch den externen Server.

> *Aus der Architektursicht ist NT ein Microkernel-System. Es stellt drei externe Server zurVerfügung, nämlich POSIX, OS/2 und natürlich Win32.*

Vorstellbar ist auch, eine Klassenbibliothek als Microkernel zu implementieren, wie die Borländer dies teilweise in der CLX realisierten, obwohl sie vor Jahren noch keine Linux-Pläne hatten.

Als externer Server der visuellen Komponenten stellt die Unit *classes.pas* ihre Dienste zur Verfügung. Die internen Server sind die vom jeweiligen Betriebssystem abhängigen Packages *Libc.pas* und *Windows.pas*. Es sind auch direkte Zugriffe auf den Microkernel erlaubt, sofern sie das Initialisieren einer Kommunikation einleiten.

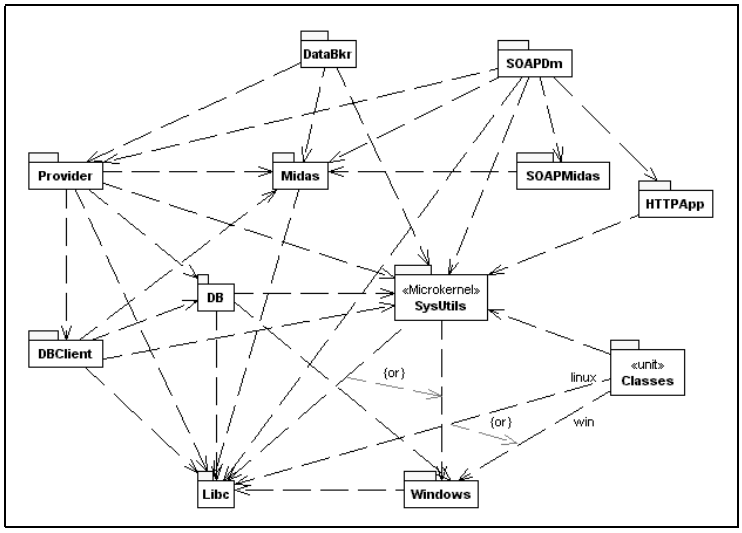

Abb. 3.30: Clients kommunizieren vor allem mit der SysUtils

Die direkten Zugriffe auf die beiden internen Server Libc und Windows wären dem Muster nach nicht erlaubt, sind aber aus Performancegründen erforderlich. Dies ist zugleich der größte Nachteil der Microkernel-Architektur gegenüber einem monolithischen System: die Geschwindigkeit. Dies ist der Preis für die Erweiterbarkeit und Flexibilität.

Einige der Units, die bei reinen Win-Anwendungen Verwendung finden, sind auch Bestandteil von plattformübergreifenden Anwendungen, darunter beispielsweise *Classes, DateUtils, DB, System, SysUtils* und andere wie z. B. die Units der Laufzeitbibliothek (RTL). Viele der plattformübergreifenden Units befinden sich jedoch in der Teilbibliothek VisualCLX und unterscheiden sich von denjenigen in der Teilbibliothek WinCLX (für reine Win-Anwendungen).

3.3.7 Monitor

Ein datenorientiertes Architekturmuster (fehlertolerante Echtzeit-Systeme).

Zweck

Ermögliche das Vergleichen von Soll- und Istwerten durch einen Monitor, der bei Datenänderung oder einem Ausfall die durch Actuatoren angepassten Aktionen laufend überprüft. Ein Controller gibt die Sollwerte dem Monitor und Actuator vor und leitet neue Korrekturen ein. Die Kommunikation zwischen Monitor und Actuator erfolgt durch synchrone Daten und Nachrichten.

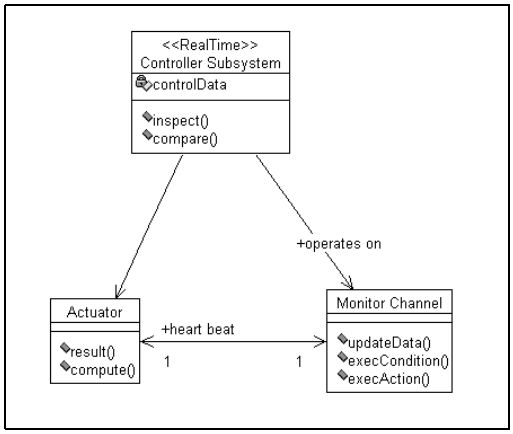

Abb. 3.31: Der Controller füttert Monitor wie Actuator

Motivation

In der Prozesssteuerung ist Echtzeit über terrestrische Einrichtungen, Satellit, Kabel oder andere Medien) oder auch Rundfunk oder andere Netzwerke möglich. Die Prozessdatenverarbeitung ist ein Spezialgebiet der Datenverarbeitung, bei der als Besonderheit die

Lenkung eines technischen Prozesses mit Actuatoren und Sensoren im Vordergrund steht. Die Modellierung ist mit Realtime-UML möglich.[vi] Dieses Konzept lässt sich auch als Analogie in die Finanzwelt übertragen.

Die Erforschung der sich rasch entwickelnden Finanzwelt wurde bis vor Jahren dadurch behindert, dass zu wenig dichte und regelmäßige Daten vorhanden waren, um die zunehmende Komplexität und scheinbar unregelmäßige Natur von Finanztransaktionen zu erfassen.

Gerade im Devisenmarkt, welcher als größter Finanzmarkt rund 42-mal mehr Umsatz generiert als alle weltweiten Aktienmärkte zusammengenommen. Durch die vorhandenen Hochfrequenzdaten (z. B. im Dollar Tick für Tick) und deren anspruchsvolle Analyse ließen sich bereits folgende Entdeckungen machen:

Es existieren innertägliche Saisonalitäten von Preisbewegungen, die man bei asiatischen Devisenhändlern fand, welche kurz vor Mittag eher verkaufen als kaufen. Solche Verhaltensmuster kennt man auch aus der Börse, wo am Montag eher gekauft und Ende der Woche eher verkauft wird. Eine andere Entdeckung belegt, dass sich die Volatilität in den Finanzmärkten wiederholt.

Verwendung

Suchroboter (robots, agents) analysieren laufend die Inhalte der verfügbaren Webseiten und fügen die Suchergebnisse (Verweis) in den Index ein. Mit einem Monitor lässt sich der Zustand wie das geforderte Resultat durch den Controller ständig überprüfen. Die Roboter können jedoch die gefundene Information weder gewichten noch einteilen oder zuordnen. Diese Situation gipfelt dann in der Tatsache, dass die Suche nach einem Stichwort Tausende von Verweisen liefert.

In einer Handelsplattform wird durch den Controller ein bestimmter Preis vorgegeben, der dem Monitor und dem Actuator bekannt ist. Aufgabe des Actuators ist es, den Preis in diversen Auktionssystemen, wie z. B. eBay, zu platzieren. Der Monitor[5] (`TfrmMain`) als Scanner (`TVCLScanner`) überwacht nun die diversen Preisangebote innerhalb einer definierten Bandbreite, ändert zwischenzeitlich bei veränderten Bedingungen durch den Controller den Preis, bis der gewünschte Verkauf (oder Kauf) stattfinden kann.

Im folgenden Szenario seien die Schritte gezeigt. Der Controller sendet das gewünschte Resultat „gleichzeitig“ an den Monitor und den Actuator. Der Actuator entscheidet gemäß dem Sollresultat, welche Aktion er tätigen soll. Der Monitor lässt sich eine erste Bedingung der Aktion zurückgeben und vergleicht mit dem Zielwert des Controller. Dieser wiederum korrigiert oder verändert das gewünschte Resultat.

Mehr zu diesem Kontext der allgemeinen Prozesssteuerung ist im Bericht „Objekt-Steuerung – Ereignisgesteuerte Programmierung von Industriecomputern“ zu finden, der im Heft Elektronik 8/2001 erschienen ist und sich auf der CD-ROM befindet.[6]

[5] Monitor hieß auch mal: Datensichtendstation.

[6] Basiert auf dem Delphi-Framework WDosX.

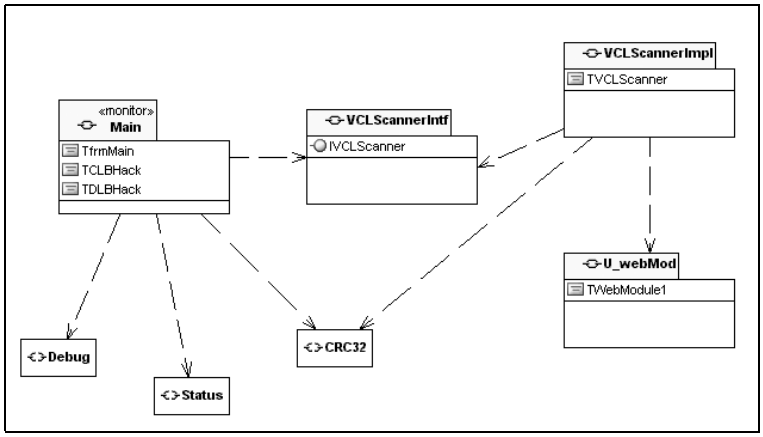

Abb. 3.32: Der Monitor vergleicht Resultat mit Zielwert

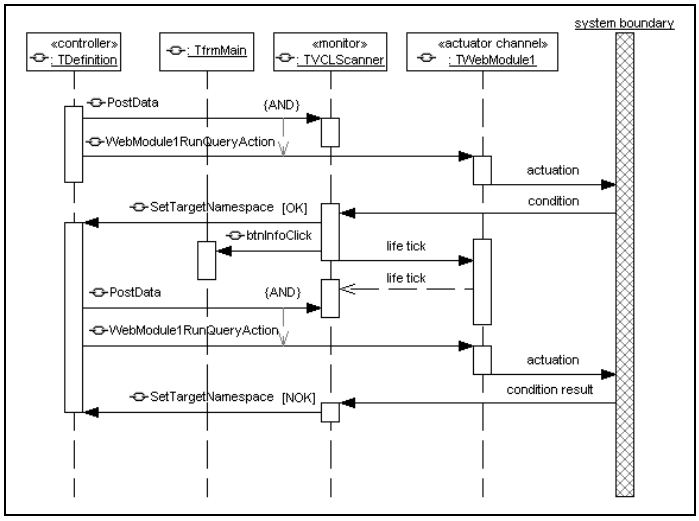

Abb. 3.33: Der Monitor vermittelt zwischen Actuator und Controller

3.3.8 MVC

Ein nachrichtenorientiertes Architekturmuster (steuert interaktive Anwendungen durch Nachrichten mittels eines Controllers).

Zweck

Ermögliche einen Mechanismus zur Benachrichtigung von Änderungen, die durchgängig konsistent zwischen GUI und Modell sind. Lasse den Controller die

Ereignisse vermitteln und erlaube dem Modell, die Daten und Kernfunktionen zu enthalten.

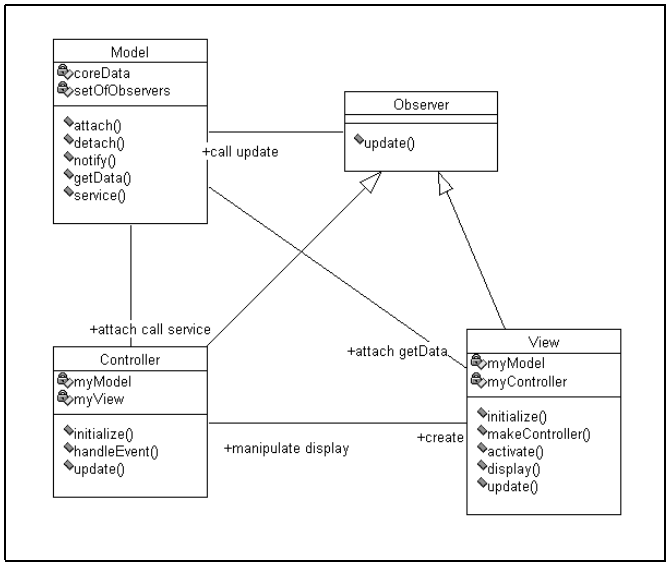

Abb. 3.34: Das MVC-Prinzip in voller Montur

Motivation

Das MVC-Muster ist eine Implementierung des Observer (siehe Teil 2) auf der Stufe Architektur, neu hinzugekommen ist der Controller. MVC ist dem Pattern Publisher-Subscriber ähnlich. Das MVC besteht aus drei Teilen:

Der Controller beinhaltet die Logik und bearbeitet die eingehenden Ereignisse. Das Model (pro Anwendung ein Model) enthält die Daten oder weiß zumindest, wo es die Daten findet. Als View kommt ein beliebiges Steuerelement in einem Fenster zum Tragen. Hauptvorteil ist die Unabhängigkeit des Models vom View.

> *Bestens geeignet auch als Middleware-Grundlage für eine indirekte, asynchrone und lose gekoppelte Kommunikation, z. B. für die Datenfluten eines Wetter-Informationssystems oder für ein Navigationssystem.*

Wenn man bedenkt, dass die Händler pro Sekunde an den Finanzmärkten Tausende von Updates generieren, dann muss ein System diese Datenfluten effizient an die Abonnenten weitergeben. Zudem sind auch Timingaspekte zu berücksichtigen, da der Erste eine Information nicht 40 Sekunden vor dem Letzten erhalten soll.

Das MVC-Muster nutzt die Tatsache, dass nicht jeder Abonnent an allen Updates pro Sekunde interessiert ist. Einzelne Daten oder Instrumente sind in derselben Kombination von allen nachgefragt, andere liest man selten. Somit lassen sich die Daten im Model nach Themen oder Lektionen (Topics) gliedern.

Verwendung

Das System erhält über den Controller das Ereignis von neuen Daten. Dann sind sie dem Model (evtl. Middleware) zu übergeben, ausgestattet mit dem jeweiligen Topic. Somit ist die Geschichte für den Server erledigt. Die Views als Händler abonnieren diejenigen Topics, die von Interesse sind und wurden vorhergehend beim Model registriert. Sobald eine Nachricht durch den Controller eintritt, wird der View informiert und er kann sich zu gegebener Zeit, meistens in Sekunden, die Nachricht beim Model abholen. Wichtig ist „zu gegebener Zeit“, da man Nachrichten erst nach bestimmten Volumen verarbeiten will. Kein Polling ist nötig und kein Server wird über Gebühren belastet.

> *Mit IP-Multicast als Protokoll lassen sich Daten nur einmal an die Views versenden, auch wenn Tausende von registrierten Finanzhändlern dasselbe Thema abonniert haben.*

Boldsoft mit seinem UML-Modellierungs- und Architekturtool für Delphi ist ein weiteres Highlight des MVC. Zudem gibt es ein anerkanntes MVC-Konzept für die Realisierung von Web-Applikationen. [vii]

Bold ermöglicht die Implementierung und Steuerung der MVC-Architektur, d. h., eine Mittelschicht von Rendering Klassen (Presentation Mapping) vermittelt zwischen den Geschäftsobjekten und der grafischen Repräsentation der Views (siehe Abb. 3.35). Zusätzlich bietet Bold ein durchdachtes Datenbankhandling an, welches nebst dem „Klassen-zu Tabellen-Mapping“ in eine relationale Datenbank auch Cached Updates sowie Transaction Handling erlaubt.

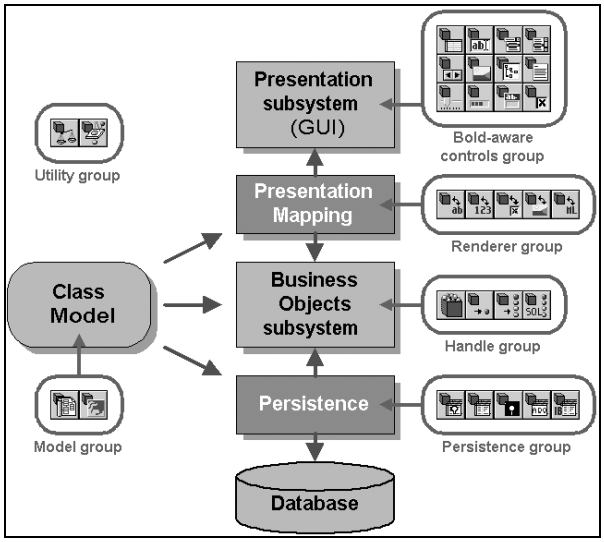

Abb. 3.35: Das MVC mit Bold als Konkretisierung

Das von mir erstellte Modell enthält die nötigen Informationen zur automatischen Generierung der Datenbank und den zugehörigen Tabellen. Dieses „Klassen-zu-Tabellen-

Mapping“ lässt sich nach der Generierung weiter steuern, da in der Regel die gewünschte Datenbank mit Indexen, Triggern oder Constraints erweitert wird.

Bold kann in diesem Sinne nur die Client-seitigen Regeln beeinflussen. Der Schemagenerator ist auf Änderungen des Modells vorbereitet, die Wahrscheinlichkeit ist groß, dass es Änderungen gibt. Wie Bold den Datenzugriff organisiert, lässt sich anhand eines Subclassing von `TDataModule` feststellen:

```
unit datamod;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
    Forms, BoldModelLink, BoldRoseLink, BoldHandles,
    BoldSystemHandle, BoldModel,BoldMM6Link;
type
  TDataModule2 = class(TDataModule)
    BoldModel1: TBoldModel;
    BoldMMLink1: TBoldMM6Link;
    BoldSystemHandle1: TBoldSystemHandle;
```

Der Datenbankalias von InterBase lässt sich dann der entsprechenden Instanz zuweisen:

```
BoldSystemHandle1.DataBase.name:=IB_TVCast
```

Das direkte Einbinden der Business-Objekte ist wohl der größte Vorteil von Bold. Der Klassencode für die Geschäfts-Objekte lässt sich größtenteils automatisieren und auch bei Änderungen aktualisieren, ohne den Einsatz von bekannten Round-Trip-Flags im Code, welche auf Veränderungen hin prüfen. Als Schluss des MVC sei das Konzept in generischer Sicht als Paketdiagramm nochmals dargestellt:

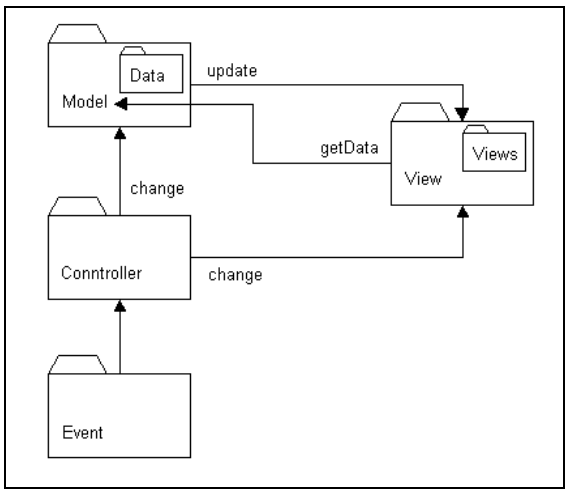

Abb. 3.36: Das MVC dient auch der Komponentengruppierung

3.3.9 PAC

Ein kombiniertes Architekturmuster (zunehmende Hierarchie autonom kooperierender GUI-Agenten).

Zweck

Bestimme einen interaktiven Agenten, der für einen bestimmten Aspekt der Funktionalität verantwortlich ist und aus drei Komponenten besteht: Präsentation, Kontrolle und Abstraktion.

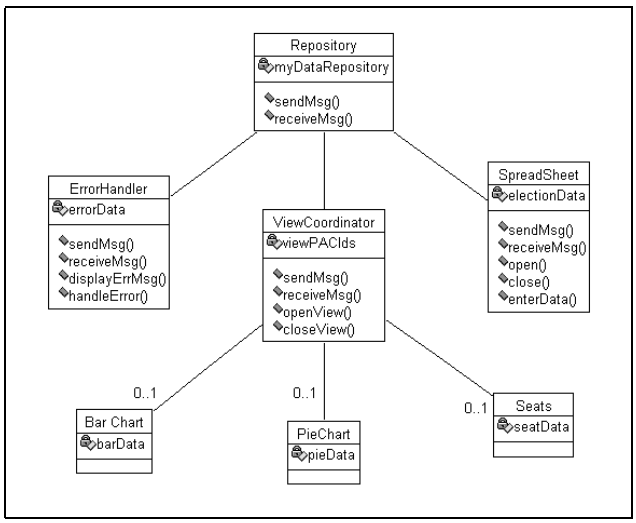

Abb. 3.37: Der Container hält die Teile

Motivation

Das PAC trennt ebenso wie das MVC den funktionellen Kern eines Systems von der Darstellung und der Eingabebehandlung. Das MVC definiert eine Steuerkomponente, die für das Entgegennehmen der Eingaben verantwortlich ist. Es existiert aber keine vermittelnde Steuerkomponente wie beim PAC. Diese Komponente ist für die Koordination zuständig und erlaubt dem PAC, ein ganzes Framework zu steuern. Der Schwierigkeitsgrad ist aber gegenüber dem MVC deutlich höher.

In der PAC-Architektur ist jeder Agent auf seinem Level eine ziemlich autonome informationsverarbeitende Komponente, die auf eine bestimmte Aufgabe spezialisiert ist. Sie verfügt über die Möglichkeit zum Empfangen und Versenden von Ereignissen. Jeder Agent besteht (wenn nötig) aus drei Teilen:

1. Die Präsentation stellt das sichtbare Verhalten dar.
2. Der Controller agiert als Vermittlerschicht, zudem kommuniziert er mit andern Agenten, die sich auf höherem oder tieferem Level befinden.
3. Die Abstraction hat Zugriff auf das Datenmodell.

Als Hauptaufgabe des PAC sei eine interaktive Anwendung mithilfe der Agenten erwähnt. Jeder Agent kann mit jedem anderen Agenten auf den drei Ebenen Top-Level, Intermediate und Bottom kommunizieren, die Abhängigkeiten sind also transitiv.

> *Zusammenfassend: Ein Agent besteht aus drei Teilen und kann über drei Levels mit anderen Agenten interagieren.*

Generell sollte die Hierarchie vom einzigen Top-Level-Agenten zu den Bottom-Level-Agenten zunehmen, da die Views und GUI-Elemente von den so genannten Ansichtenkoordinatoren gesteuert werden. Das Muster ist wie eine umgekehrte Pyramide zu betrachten. Der Top-Level-Agent hat meistens keinen Präsentationsteil.

Beispielsweise sollte man MDI-Anwendungen sorgfältig planen. Ihr Entwurf kann etwas komplizierter sein als der von SDI-Anwendungen. Sie spannen innerhalb des Client-Fensters untergeordnete Fenster auf; das Hauptformular enthält untergeordnete Formulare. Setzen Sie die Eigenschaft `FormStyle` des `TForm`, um anzugeben, ob ein Formular ein untergeordnetes Formular (`fsMDIChild`) oder ein Hauptformular (`fsMDIForm`) ist. Sie sollten eine Basisklasse für Ihre untergeordneten Formulare definieren und jedes untergeordnete Formular von dieser Klasse ableiten, um zu vermeiden, dass Sie die Eigenschaften des untergeordneten Formulars erneut setzen müssen.

Verwendung

Als Verwendung dient auch hier wieder Bold, nur betrachten wir das System detaillierter. Bei der Implementierung der GUI sind wir auf die Bold-**eigenen** Komponenten wie Grid oder Navigator angewiesen, da nur sie Verknüpfungen zur Business-Schicht ermöglichen und zudem bei Änderungen auch modellgetrieben sind. Das Positionieren und Verknüpfen der Komponenten erfolgt größtenteils ohne manuelles Codieren.

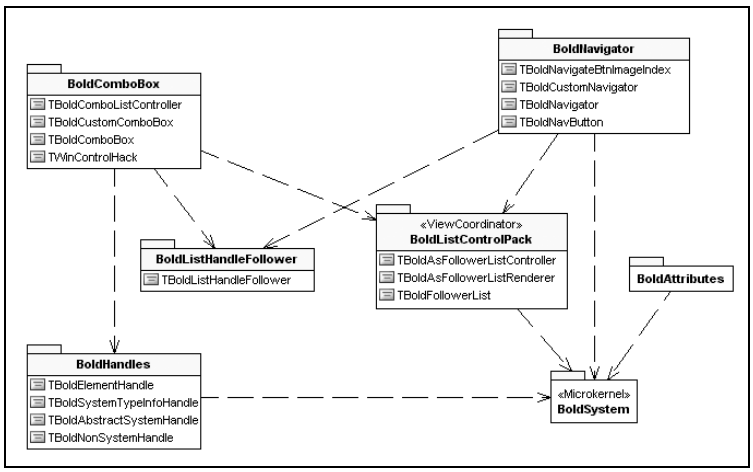

Abb. 3.38: Der Agent koordiniert

In Abb. 3.38 ist die Steuerung `BoldListControlPack` als Intermediate-Agent deutlich zu sehen. Er besteht aus den drei Teilen Renderer für die Präsentation, Controller als Vermittler und FollowerList für den Datenzugriff. Die Bottom-Level-Agenten `BoldNavigator` und `BoldComboBox` kommunizieren mit den Intermediate-Agenten.

Konkret ist bspw. eine Datumsanzeige in einem „DateTimePicker“ mit einem Follower verknüpft, der die Eigenschaften eines Forms erbt.

```
procedure TBoldDateTimePicker.SetBoldProperties(const Value:
                    TBoldDateTimeFollowerController);
begin
  fBoldProperties.Assign(Value);
end;
```

Innerhalb des Konstruktors lassen sich die Agenten verknüpfen, d. h., die `BoldProperties` sind für die Vererbung der oberen Levels zuständig und werden dem Agenten für den Datenzugriff (`fHandleFollower`) übergeben.

```
constructor TBoldDateTimePicker.create(owner: TComponent);
begin
  inherited;
  fBoldProperties:= TBoldDateTimeFollowerController.Create(self);
  fBoldProperties.AfterMakeUpToDate := _Display;
  fHandleFollower:= TBoldElementHandleFollower.
                          create(Owner,fBoldProperties);
end;
```

Der eigentliche Zugriff auf das aktuelle Datum erfolgt über den Follower, der vom Typ `TBoldElementHandleFollower` ist. Sobald also bei der Methode _Display ein Aktualisierungsereignis eintrifft, lässt sich über den Controller das aktuelle Datum mit dem Follower holen. Man hat aber auf den Follower nur über den Controller Zugriff!

```
procedure TBoldDateTimePicker._Display(Follower: TBoldFollower);
var
  newDateTime: TDateTime;
begin
  newDateTime:=
       TBoldDateTimeFollowerController(Follower.Controller).
                         GetCurrentAsDateTime(Follower);
  if Date <> newDateTime then
    Date:= newDateTime;
end;
```

Ein weiterer Agent ist verantwortlich, dass auch der BoldNavigator in Abb. 3.38 sich aus den einzelnen Klassen Navigator, Image, Button etc. zusammensetzt, somit gilt er auch als View-Coordinator (Ansichtenkoordinator).

Top-Level-Agenten sind in jedem System nur einmal vorhanden, hier ist es das Bold-System. Bspw. kann ein Top-Level-Agent überprüfen, ob eine bestimmte Datenoperation auf dem Datenmodell erlaubt ist. Es ist auch möglich, Aufrufe von Funktionen zu verfolgen, um erneute Aufrufe oder Dienste schneller anzubieten. Dies sei mit der Klasse `TBoldListHandleFollower` angedeutet. Ein vertiefter Blick in das Package des Bold-System als Top-Level-Agent zeigt folgendes (nicht einfaches) Bild der Klassen:

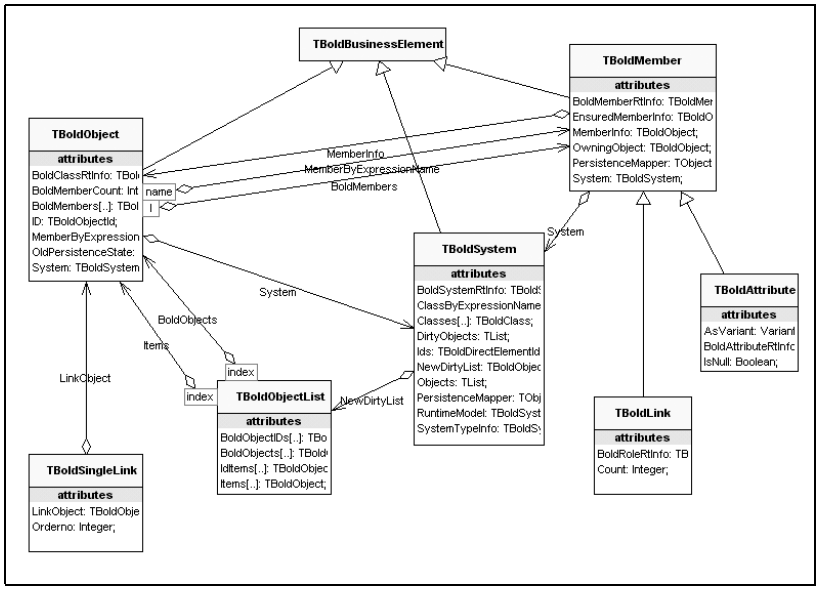

Abb. 3.39: Der Top-Level-Agent ermöglicht den Datenzugriff

Im Zuge der Tabellen-Generierung kann Bold auch die Codegenerierung der Business-Klassen bewerkstelligen. Bold erzeugt zusätzlich den Code, den man im aktiven Modell nicht explizit darstellen will oder kann. Konkret wird ja nebst den Klassen meist noch eine Link-Klasse benötigt (weil wir eine n:m-Relation abbilden), d. h. nebst z. B. den Klassen `TPerson` und `TBuilding` kommt noch die Klasse `TOwnership` dazu. Diese Link-Klasse wird aus `TBoldLink` abgeleitet. Zusätzlich entstehen die Coderümpfe der beiden Methoden direkt in Delphi.

Eine Datenbank wird aus den Klassen dann automatisch generiert und lässt sich später modifizieren, wenn das Modell eine Änderung erfährt. Ein Mapping der Felder zu Attributen ist möglich. Für weitere Änderungen, wie das Kreieren von Bold-Attributen, lassen sich auch Wizards einsetzen. Ohne zu übertreiben ist die Bold-Architektur vom Feinsten, in Bezug auf das Design.

Es ist noch schwierig, eine vernünftige Einteilung für Bold zu finden. Es ist weder ein vollständiges, autonomes CASE Tool noch eine „große“ Komponentensammlung. Ich würde Bold als MDA Business **Framework** bezeichnen.

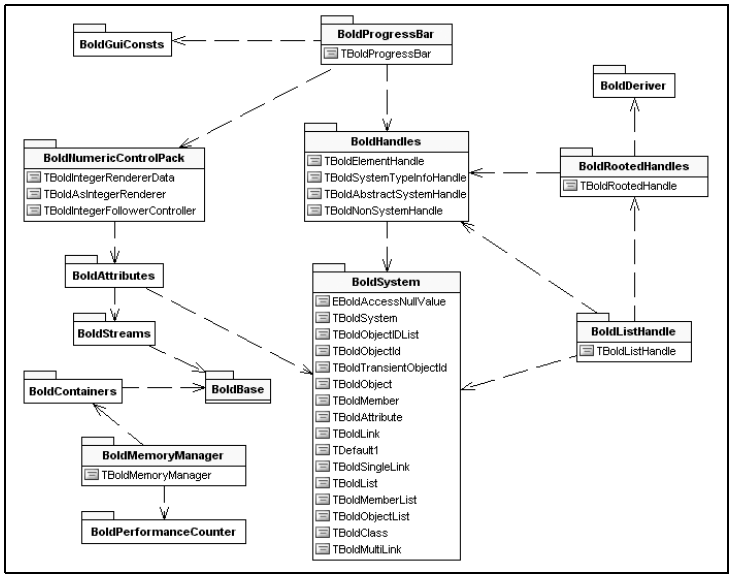

Abb. 3.40: Die Bold-Architektur im Prinzip des PAC

3.3.10 Prototype

Ein kombiniertes Architekturmuster (implementiert die Instanz eines Frameworks).

Zweck

> *Bestimme die Arten von zu erzeugenden Instanzen durch die Verwendung eines prototypischen Frameworks und erzeuge die Objekte durch schlichtes Kopieren des Prototypen. Erweitere das Grundgerüst durch Delegation.*

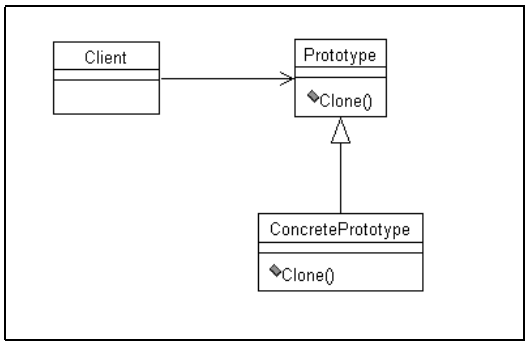

Abb. 3.41: Ausgehend vom Prototypen ein neues Projekt

Motivation

Von einem bestimmten Framework sind mehrere direkte Nachkommen oder Instanzen vorhanden, von denen jeder einem anderen Mechanismus für den Datenzugriff und die Funktionsschicht entspricht. Man arbeitet direkt mit diesen erzeugten Instanzen.

Die einzelnen Nachkommen oder Instanzen führen die Eigenschaften und Methoden ein, die zur Verwendung eines bestimmten Mechanismus für den Datenzugriff und die Logikschicht benötigt werden Diese Eigenschaften und Methoden werden dann von den Nachkommen dargestellt, die an die verschiedenen Arten von Serverdaten angepasst sind.

Verwendung

Nachdem ein Produkt bereitgestellt worden ist, möchte man möglicherweise das Erscheinungsbild der endgültigen HTML-Seiten ändern. Unter Umständen ist das Entwicklungsteam jedoch gar nicht zuständig für das endgültige Seitenlayout. Diese Aufgabe ließe sich bspw. von einem eigenen Webdesigner in der Organisation erledigen.

Die Entwickler verfügen eventuell über keine Erfahrung. Glücklicherweise ist solche Erfahrung auch nicht erforderlich. Denn die Webdesigner können Seitenvorlagen oder Instanzen von Prototypen zu einem beliebigen Zeitpunkt im Entwicklungs- und Pflegezyklus bearbeiten, ohne dazu den generischen Code ändern zu müssen. Auf diese Weise wird die Entwicklung und Pflege von Servern durch HTML-Vorlagen effizienter.

Die Komponente und deren Arten lassen sich generell und sehr praktisch auch von einem MVC-Prinzip als Prototyp ableiten (siehe MVC) und die Kopien davon einteilen:

- Model-Komponenten für den Datenzugriff
- View-Komponenten als visuelle Bausteine
- Controller-Komponenten mit viel Logik und Kalkül

Auch in der COM-Welt scheint der Prototyp einen Ableger gefunden zu haben:
Benötigt ein Programm ein neues Objekt, so schaut Windows in der zentralen Registrierungsdatenbank nach, welche EXE-Datei oder DLL dafür zuständig ist, und lädt den Code. Dort muss sich eine so genannte `ClassFactory` befinden, welche die einzelnen Objekte der Klasse durch den Prototypen erzeugen kann. So hat man dann Zugriff auf die gewünschten Methoden und Eigenschaften des geklonten Objektes.

Vielfach ist nach dem Instanzieren eines Prototypen die Delegation im Zusammenhang mit neuen Objekten gefragt. In Abb 3.42 will ich nach dem Instanzieren des Frameworks die Unit `TBusinessObj` mit neuer Funktionalität ergänzen. Hierzu zieht man die Delegation der Vererbung vor.

Vererbung im üblichen Sinne der OO gibt es auch bei COM nicht. Man spricht von Delegation. Das mag daran liegen, dass COM nicht so sehr für die Entwicklung kompletter Klassenbibliotheken und deren Hierarchie, sondern für das objektorientierte **Verpacken** von Anwendungen beziehungsweise deren Bausteinen gedacht ist.[viii]

Zur Unterstützung dieser Struktur für Schnittstellen stellt Delphi das Schlüsselwort `implements` bereit, mit dem sich die Implementierung einer Schnittstelle ganz oder teilweise einem untergeordneten Objekt übertragen lässt (Delegation).

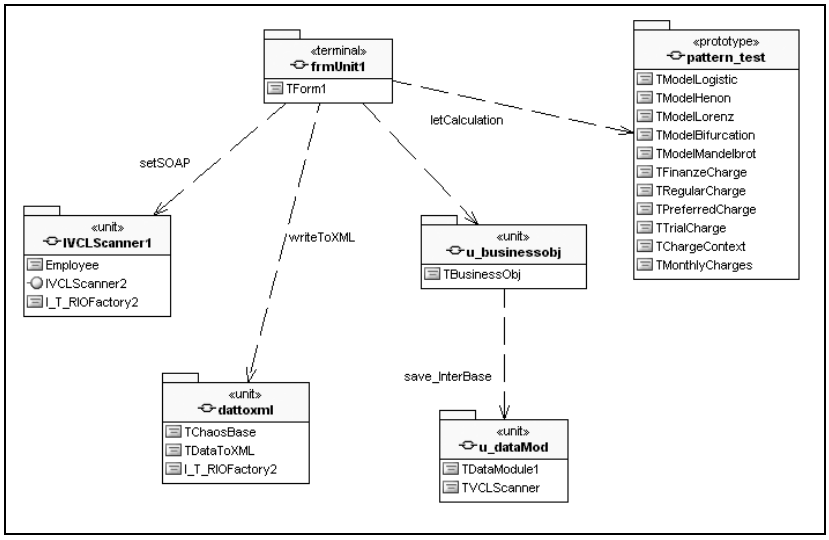

Abb. 3.42: Abgeleitet vom Prototypen als geklontes MVC

3.3.11 Provider (DataSet)

Ein datenorientiertes Architekturmuster (mehrschichtige Anwendung durch Datenpakete).

Zweck

Biete eine Unterstützung als mehrschichtige Anwendung und stelle eine Erweiterung der Art und Weise dar, in der Client-Datenmengen unter Verwendung übertragbarer Datenpakete mit Provider-Anwendungen kommunizieren können.

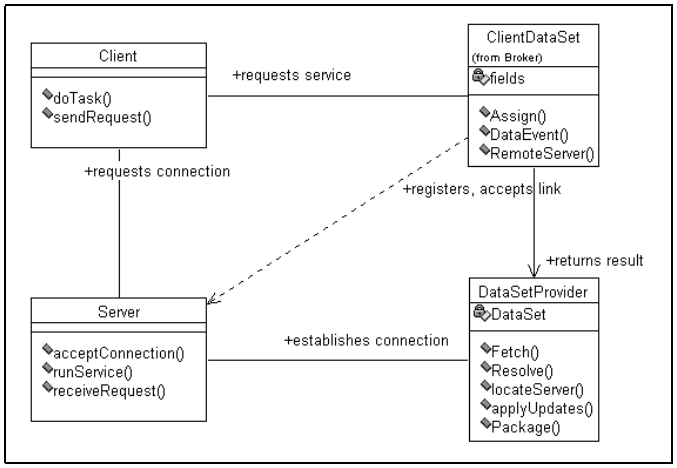

Abb. 3.43: Der Provider lässt Daten verpacken

Motivation

Bereits in der Einleitung hatte der Provider in der Abb. 3.7 seine Kurzvorstellung. Im Grunde ist der Provider eine Konkretisierung des eher allgemeinen Layers. Die Hauptmotivation in einem dreischichtigen Modell liegt sicher darin, dass ein Anwendungsserver den Datenfluss zwischen den Clients und dem Remote-Datenbankserver verwaltet. Mit einem Provider werden Daten für eine Client-Datenmenge oder einen XML-Broker bereitgestellt und die vorgenommenen Aktualisierungen wieder in die ursprüngliche Datenmenge oder in den zugrunde liegenden Datenbankserver eingetragen.

Ein Provider kann sich in derselben Anwendung wie die Client-Datenmenge oder der XML-Broker befinden oder kann in einer mehrschichtigen Architektur Teil eines separaten Anwendungsservers sein und gilt damit als Middleware zwischen Client und Datenbank. Der Provider dient als Daten-Broker zwischen einem Remote-Datenbankserver und einer Client-Datenmenge.

Verwendung

`TDataSetProvider` stellt die Daten der Quelldatenmenge zusammen und übergibt sie in einem oder mehreren Datenpaketen an die Client-Datenmenge oder den XML-Broker.

`TBaseProvider` ist der unmittelbare Vorfahr der Klasse `TDataSetProvider`, die Daten aus beliebigen Datenmengen bereitstellt und Aktualisierungen in diese Datenmengen zurückschreibt oder direkt an einen SQL-Server übergibt.

> *Von TBaseProvider können Sie weitere, individuelle Provider-Objekte, wie z. B. XProvider, die Aktualisierungen direkt (ohne SQL) in eine Datenbank eintragen, ableiten.*

Das folgende Diagramm ist mal nicht UML-Notation, sondern zeigt den Editor von Delphi, der die einzelnen Komponenten mit ihren Abhängigkeiten darstellt:

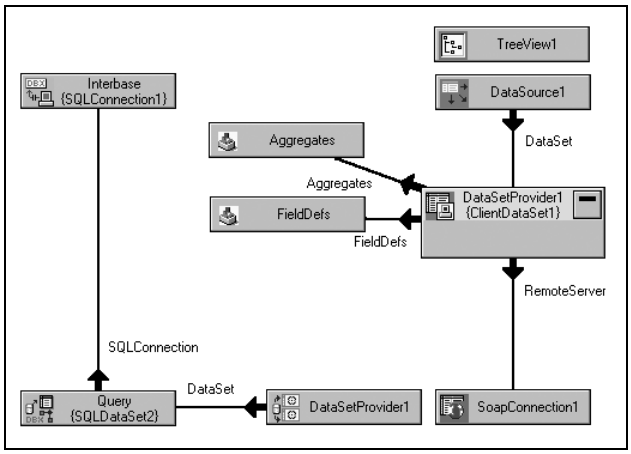

Abb. 3.44: Der physische Ort des Provider ist wählbar

Es ist offensichtlich, dass `TDataSetProvider1` von `ClientDataSet1` gerufen wird, wobei im `ClientDataSet` noch die Verbindung (Sockets, SOAP, ORB etc.) einstellbar ist. Wenn der Provider Daten für eine Client-Datenmenge bereitstellt, wandelt der Client die erhaltenen Daten in dem Datenpaket in eine lokale Kopie um, die für die Dauer des Benutzerzugriffs im Speicher verbleibt.

Wenn der Zugriff abgeschlossen ist, stellt die Client-Datenmenge die geänderten Daten zusammen und sendet sie an den Provider zurück. Der Provider trägt die Änderungen in die Datenbank oder die Quelldatenmenge ein. Wenn das `ClientDataSet` mehr Daten benötigt, wird im Grunde die Methode `InternalGetRecords` des Providers aufgerufen und die Anzahl Records gemäß `PacketRecords` in `RecsOut` auf die Reise geschickt:

```
function TDataSetProvider.InternalGetRecords(Count: Integer;
           out RecsOut: Integer; Options: TGetRecordOptions;
                  const CommandText: WideString;
                    var Params: OleVariant): OleVariant;
begin
  try
    if not DataSet.Active then begin
      DataSet.Open;
      FDataSetOpened:= True;
    end;
    if (Count = 0) or (grMetaData in Options) then begin
      FDataDS.Free;
      FDataDS:= NIL;
      FRecordsSent:= 0;
    end;
    DataSet.CheckBrowseMode;
    DataSet.BlockReadSize:= Count;
    try
      Result:= inherited InternalGetRecords(Count,
                   RecsOut, Options,CommandText, Params);
      Inc(FRecordsSent, RecsOut);
      if (RecsOut <> Count) then Reset;
    finally
      if DataSet.Active then begin
        DataSet.BlockReadSize:= 0;
        if (Count <> 0) and (RecsOut = Count) then
          DataSet.Next;
      end;
    end;
  except
    Reset;
    raise;
  end;
end;
```

Wenn der Provider Daten für einen XML-Broker bereitstellt, fügt der XML-Broker das Datenpaket im XML-Format einem HTML-Dokument hinzu, das von einem Web-Client heruntergeladen wird. Wenn der XML-Broker Aktualisierungen vom Web-Client erhält, sendet er diese an den Provider, der die Daten in die Datenbank oder Quelldatenmenge einträgt.

Client-Datenmengen und XML-Broker kommunizieren mit Providern über die IAppServer-Schnittstelle. In mehrschichtigen Anwendungen ist dies die Schnittstelle des Remote-Datenmoduls, das den Provider enthält. Damit das Datenmodul Aufrufe an den Provider übergeben kann, muss dessen Eigenschaft `Exported` auf true gesetzt sein und in der Eigenschaft `Owner` muss das Remote-Datenmodul angegeben sein.

> *Ist der Provider Teil einer mehrschichtigen Anwendung, so handelt es sich um die Schnittstelle für das Remote-Datenmodul des Anwendungsservers oder (im Falle eines SOAP-Servers) um eine von der Verbindungskomponente erstellte Schnittstelle.*

Nicht ausführbare Aktualisierungen an den Server werden protokolliert und an die Client-Datenmenge zurückgesendet. (Diese Aktivitäten nennt man Resolving.)

Bei Verwendung von `TBDEClientDataSet`, `TSimpleDataSet` oder `TIBClientDataSet` arbeitet der Provider innerhalb der Datenmenge und die Anwendung hat keinen direkten Zugriff auf den Provider. Datenmengen mit einem internen Provider stellen einige der internen Provider-Eigenschaften als eigene Eigenschaften und Ereignisse zur Verfügung. Zur bestmöglichen Steuerung sollte man jedoch `TClientDataSet` mit der separaten Provider-Komponente benutzen.

3.3.12 Simulator (Blackboard)

Ein kombiniertes Architekturmuster (Lösungsstrategien durch spezielle Subsysteme).

Zweck

> *Spezialisierte Subsysteme stellen ihr Wissen für Probleme zur Verfügung, für die keine deterministischen Lösungsstrategien bekannt sind. Das zusammenfassende Resultat stellt eine approximative Lösung dar.*

Motivation

Die Chaostheorie hält nach zugrunde liegenden Verhaltensmustern Ausschau, welche in scheinbar regellosem Verhalten festgestellt wurden. Hier treffen sich unsere Patterns mit den Simulationsmustern! Sie wird auch als deterministische Chaostheorie bezeichnet und basiert auf dem Prinzip der Selbstähnlichkeit, denn verschiedene Objekte gehen sowohl bei Vergrößerungen als auch bei Verkleinerungen in sich selbst über.

Mit dieser Erkenntnis der Selbstähnlichkeit wird versucht, Gestalt in scheinbares Chaos zu bringen. Die geometrische Ausprägung dieser Theorie wird als Fraktal bezeichnet.

Charts sind gemäß dieser Theorie auch Fraktale, weil auch sie ein Selbstähnlichkeitsmuster besitzen.

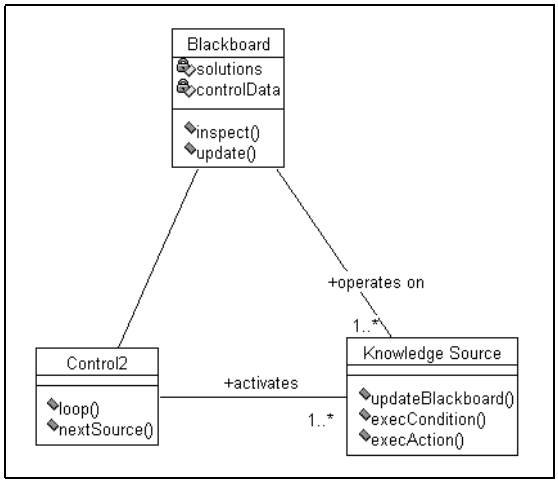

Abb. 3.45: Mit dem Blackboard auf die Lösung warten

Die Finanzmärkte in ihrem chaotischen Verhalten spiegeln verschiedene menschliche Eigenschaften wider. Hoffnungen und Ängste, Stärken und Schwächen, Gier und Idealismus, rationale Urteilskraft und Intuition der an den verschiedenen Börsen teilnehmenden Individuen ergeben als Ganzes den Marktpreis des jeweiligen Finanzgutes. Durch die stetig wiederkehrende, gefühlsmäßige Wechselwirkung zwischen Individuum und Masse, welche auf den ersten Blick chaotisch anmutet, sind Charts Fraktale, welche der Stimmung der Masse entsprechen und subtile Selbstähnlichkeit besitzen.[7]

> *Die Meteorologen arbeiten heute mithilfe von Computersystemen und ihr Vorgehen beruht auf dem Kausalitätsprinzip. Dieses besagt, dass gleiche Ursachen gleiche Wirkungen haben.*

Demnach müsste bei Kenntnis aller Wetterdaten eine exakte Voraussage möglich sein. Dies ist natürlich in der Praxis nicht zu realisieren, weil wir die Messstationen zur Erfassung der Wetterdaten nicht in beliebiger Zahl verteilen können und auch die Grundlagen der Dateninterpretation fehlen.

Deshalb legen die Meteorologen das „starke Kausalitätsprinzip“ zugrunde, wonach ähnliche Ursachen ähnliche Wirkungen haben. Solche Modelle lassen sich in Form von komplizierten mathematischen Gleichungen zur Voraussage berechnen. Beispielsweise lässt sich der Vorgang zur Erstellung einer Prognose von sechs Stunden in ihrem Prinzip wie folgt beschreiben: Die 24-Uhr-Prognose wird einfach dadurch erreicht, dass man die Daten der 18-Uhr-Rechnung wieder in das Modell einfüttert. Das heißt:

[7] Complexity is repeated Application of simple Rules.

> *Das System erzeugt mithilfe der Software Ausgangsdaten. Die so berechneten Ausgangsdaten füttert man nun als Eingangsdaten wieder ein. Es entstehen neue Ausgangsdaten, die wiederum zu Eingangsdaten werden. Das Datenmaterial wird also immer wieder mit dem Programm „rückgekoppelt“.*

Man sollte nun meinen, dass so die Ergebnisse immer genauer werden. Das Gegenteil kann jedoch der Fall sein. War das Modellsystem Wetter eben noch in „harmonischer“ Übereinstimmung mit den Voraussagen, so zeigt es auf einmal scheinbar „chaotisches“ Verhalten. Das Modellsystem kippt um in scheinbare Unordnung, eben in „Chaos“.

Die Ursache für ein „chaotisches“ Verhalten liegt in der Tatsache begründet, dass geringfügige Änderungen der Kenngrößen (Parameter-Ungenauigkeiten), die man rückkoppelt, zu unerwarteten, ja katastrophalen Folgen der Ausgangsgrößen führen können wie ein Schmetterlingsflügelschlag im Amazonas, der dann einen Tornado in Texas verursacht (Butterfly Effect).

Verwendung

Aus diesem Grund beschränkt sich die Chaostheorie meist nur auf den Versuch, die Richtung oder Volatilität der Kursentwicklung oder die Intensität von Kursbewegungen zu prognostizieren.[ix] In der Finanzprognose wurden bislang hauptsächlich statistisch-ökonometrische Verfahren angewandt. Diese Methode unterstellt, dass sich die in der Vergangenheit beobachteten Strukturen und Regelmäßigkeiten in Modellen und darauf folgender Simulation darstellen lässt.

Die häufig erwähnten neuronalen Netze hingegen arbeiten nicht mit einer statischen Modellvorgabe, sondern leiten kontinuierlich und adaptiv aus den Daten der Vergangenheit ein Erklärungsmuster ab, so auch im TRadeRoboter. Da dies über einige Zeit annähernd konstant bleibt, können somit Prognosewerte einfließen.

Neuronale Netze finden auch immer stärker Eingang in die technische Analyse. Die Idee, das Problem der kurzfristigen Kursprognose mit neuronalen Netzen anzugehen, begründet sich darauf, dass die Kurse der Wertpapiere im Grunde nichts anderes darstellen als „chaotische Zeitreihen“, welche sich durch die adaptive Lernfähigkeit der neuronalen Netze besser erkennen lassen.

Diese Netze gelten als eine Art „künstliche Intelligenz“, weil sie im Prinzip wie das menschliche Gehirn arbeiten. Sie sind lernfähig und können Entwicklungen voraussagen. Sie erweitern die Möglichkeiten der Computertechnik, wenn Daten vorliegen, die ungenau oder unvollständig sind, die sich teilweise widersprechen oder die eine Entwicklung verschieden stark beeinflussen.

Da zeigt sich der Vorteil auch gegenüber dem Menschen. Auf den Devisenhändler wirkt über Monitore, Telefone und Kollegen eine Flut von Informationen ein. Er ist gezwungen, selektiv, aufgrund seiner Erfahrung, eine subjektive Marktbeurteilung vorzunehmen. Die neuronalen Netze modellieren die Einflussfaktoren nicht gefiltert, aber gewichtet. Und mit einer weit komplexeren Datenbasis, als sie dem Händler mit der herkömmlichen Software zugänglich ist.

Linear	Gleichung $y = mx+b$
Logarithmisch	Gleichung $y = c\ln x+b$
Polynomisch	Gleichung $y = b+c_1x+c_2x^2+....+c_6x^6$ Zahl 2 – 6 im Feld Ordnung eingegeben.
Potenziell	Gleichung $y = cx^b$ Darstellung negativer Werte nicht möglich.
Exponenziell	Gleichung $y = ce^{bx}$ Darstellung negativer Werte nicht möglich.
Chaotisch	Gleichung $x_t = 4x_{t-1} (1-x_{t-1}c)+b$ Iteration mit bekanntem Anfangswert.

Tab. 3.1: Elementare Formeln eines Blackboards

Als weitere theoretische Verwendung, nebst den neuronalen Netzen und der Fuzzy Logic, gibt es die genetischen Algorithmen.

Eine Regel bezieht sich typischerweise auf etwa 70 bis 80 Bedingungen, enthält also im Bedingungsteil genauso viele Felder. Zentral ist nun, dass in diesen Feldern nicht nur die Werte 0 und 1 (Bedingung trifft zu/trifft nicht zu) stehen können, sondern auch ein Sternchen, sodass diese Bedingung für diese Regel keine Rolle spielt. Zu Beginn der Simulation sind Nullen, Einsen und Sternchen völlig zufällig auf den Bedingungsteil der ca. 60 Regeln jedes Akteurs verteilt.

Jede Periode läuft nun folgendermaßen ab: Als Erstes bestimmen die Akteure, welche Untermenge ihrer Regeln auf die aktuelle Marktsituation zutrifft, wobei man nur Regeln mit einer nicht negativen Stärke berücksichtigt. Von diesen wählen sie eine zufällig aus, wobei die Wahrscheinlichkeit der Wahl proportional zur Stärke der Regel ist.

Die gewählte Regel bestimmt nun, ob gekauft oder verkauft wird. Aus den angebotenen und nachgefragten Mengen aller Akteure berechnen sich die Marktpreise der Aktien. Als letzte Aktivität in einer Periode lassen sich die Stärken aller (nicht nur der aktivierten) Regeln aufdatiert. Hätte oder hat die Handlung der Regel unter der aktuellen Situation zu einem Verlust geführt, dann nimmt die Stärke ab, im umgekehrten Fall nimmt sie zu.

> *Bis jetzt sind diese Entscheidkriterien noch ziemlich statisch. Die Stärke erfolgreicher Regeln nimmt zwar zu und die katastrophalen Regeln rutschen allmählich ins Negative, aber sonst bleibt alles beim Alten.*

Erst wenn die genetischen Algorithmen ins Spiel kommen, wird das Ganze spannend. Man kann das Regelset nämlich als eine Art Gene betrachten und mit ihnen eine Minievolution veranstalten. Das heißt, von Zeit zu Zeit passieren spontane Mutationen: Neue Regeln ersetzen wenig erfolgreiche oder die Handlungsanweisung von schwachen Regeln wird umgedreht. Starke Regeln kreuzt man miteinander. Auch eine Duplikation von erfolgreichen Regeln mit anschließender Mutation einzelner Felder ist möglich. Das Resultat all dieser Mutationen ist eine allmähliche Verbesserung des anfänglich zufälligen Regelsets.

Das Blackboard im Gesamtkontext (wenn es so etwas gibt), ist dann in Abb. 3.48 zu sehen. Im Subsystem `DBTradeSystem` ist der Kern des Blackboard modelliert. Neuronale Netze, mit genetischen Algorithmen kombiniert, können auch zu folgendem Klassendiagramm führen, das dann im Subsystem `DBTradeSystem` enthalten ist:

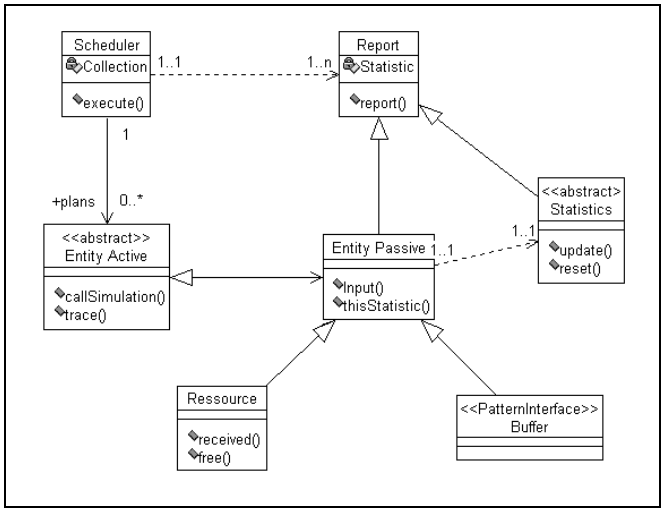

Abb. 3.46: Das Blackboard als Simulation erweitert

Zuerst wird die Simulation (Entity Active) mit Kursdaten aus der Klasse Ressource gefüttert. Die Simulation lässt sich dann mit dem Scheduler starten und steuern und die Zwischenergebnisse der Iterationen landen in der Klasse Buffer. Nach einer vom Scheduler bestimmten Zeit erfolgt eine statistische Berechnung mit entsprechenden Update-Intervallen für die Simulation, die als Gesamtes dann in einem Report resultiert. Die Abbruchbedingung legt die Methode `trace()` fest. Als Ergebnis erscheint bspw. der optimale Moving Average mit Buy/Hold-Signalen in einem Zeitfenster.

Die daraus entstehenden Durchschnittslinien (Durchschnitte oder gleitende Durchschnitte) lassen sich je nach Aktualität gewichten. Folgende Durchschnitte ergeben sich am häufigsten:

- 34-Tage-Schnitt = Kurzfristige Sichtweise (Fibonacci-Zahl)
- 100-Tage-Schnitt = Mittelfristige Sichtweise
- 200-Tage-Schnitt = Langfristige Sichtweise

Die Durchschnitte bieten die ersten Signale über den jetzigen und zukünftigen Kursverlauf. Steigt (oder fällt) der Schnitt und der Kursverlauf durchstößt die Durchschnittslinie von unten, dann spricht man von einer Stärke. Diese Stärke signalisiert Kaufen (in Abb. 3.48 als senkrechte grüne Linie markiert). Fällt (oder steigt) der Schnitt und der Kursverlauf durchstößt die Durchschnittslinie von oben, so spricht man ebenfalls von einer Stärke. Diese Stärke signalisiert jedochVerkaufen (rote Linie).

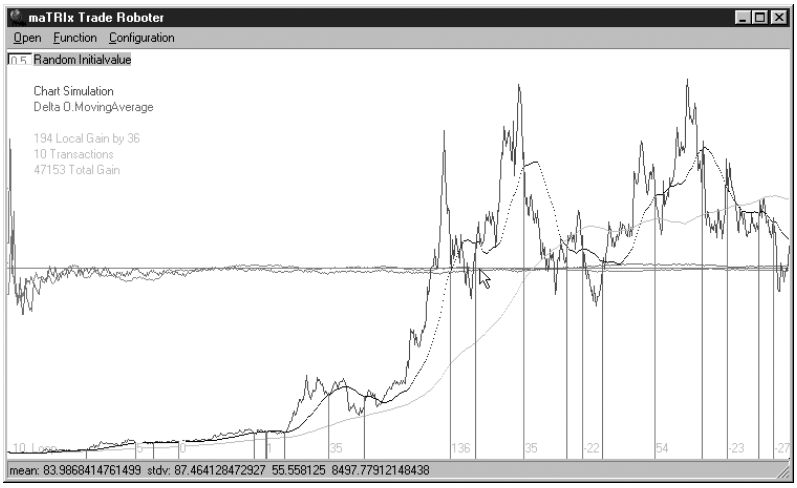

Abb. 3.47: Das Blackboard liefert den optimalen Moving Average

Mit einer vom amerikanischen Hydrologen H. E. Hurst entwickelten Methode lassen sich Trendstärken und Zykluslänge von Wellenbewegungen oder der Börsenentwicklung messen. Die Methode sieht wie folgt aus:

Man nehme die durchschnittliche Volatilität (erwartetes Schwankungspotenzial) der Kurse während verschieden langer Zeiträume und trage die so ermittelten Punkte in ein Diagramm ein. Die erhaltenen Punkte liegen dann approximiert auf einer steigenden Gerade, da die Volatilität mit der Zeit zunimmt. Bei einem reinen Random Walk hat diese Gerade eine Steigung von 50 %.

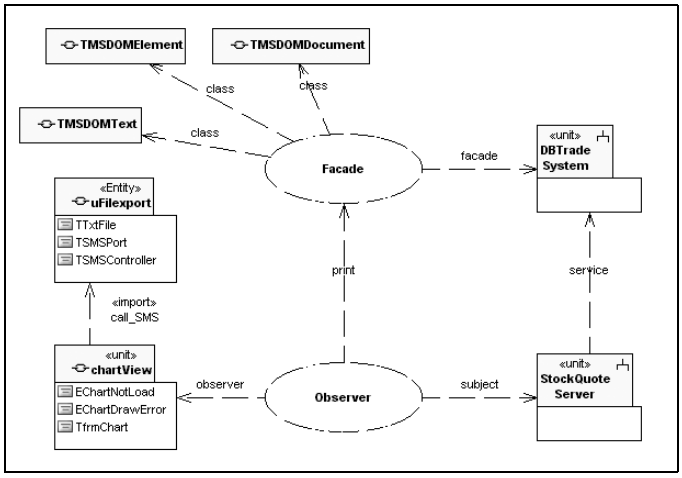

Abb. 3.48: In Kombination mit Subsystemen

Wenn sich aber der Kurs während bestimmter Zeiträume in eine bestimmte Richtung (Trendfolge) bewegt, dann zeigt sich die Gerade steiler. Je steiler dann die Ausprägung der Geraden ist, desto klarer die Trendbildung. Der Hurst-Exponent ist also eine Zahl zwischen 0 und 1, die angibt, wie stark sich eine Kursfolge (Zeitreihe) in der Zukunft überhaupt als Trend manifestiert. Wenn die Zahl aber zwischen 0 und 0.5 ist, sinkt die Wahrscheinlichkeit, dass ein Trend erkennbar wird. Untersuchungen an den Finanzmärkten haben Folgendes ergeben: 0.53 bei Devisenkursen, 0.72 bei Aktien und 0.81 bei der Konjunkturentwicklung.

Die Konjunktur ist also stark von ausgeprägten Trends begleitet, wohingegen Devisenkurse nahe an 0.5 eher einer Zufallsbewegung gleichen. Wenn nun der Hurst-Exponent mit dem optimalen Moving Average kombiniert wird, erhält man ein Trendfolgesystem mit klaren Kauf- und Verkaufssignalen.[x]

3.3.13 Terminal-Server

Ein kombiniertes Architekturmuster (Fernsteuern einer Applikation).

Zweck

Ermögliche die horizontale Struktur einer Anwendung, die ein vertikaler Client steuert. Dieser stellt die Verbindung zumTerminal-Server her.

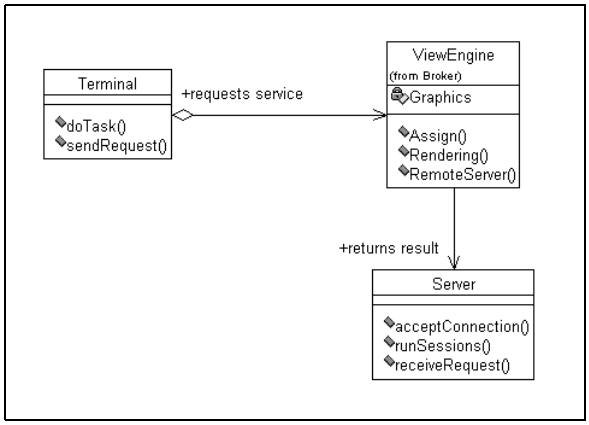

Abb. 3.49: Fernsteuern mit dem Terminal

Motivation

Nicht in jedem Falle ist eine Browserlösung oder ein Rich-Client (hieß früher mal Fat-Client) ideal. Der Unterschied zwischen Terminal Services und einer traditionellen Mainframe-Umgebung besteht darin, dass die so genannten „dummen Terminals“ des Mainframe nur über die text- oder zeichen-basierende Ein-/Ausgabe verfügen.

Ein grafischer Terminal-Server stellt mithilfe der Terminal Services gleichzeitig mehrere Client-Sessions zur Verfügung, auf welche sich die Benutzer von einem Terminal einloggen und auf dem Server jegliche 16- und 32-Bit-Applikationen betreiben können. Die Anwendungen laufen also effektiv auf dem Server ab, nur die Bild- und Toninformationen kommen zum Client.

Ein Terminal-Services-Client ist ein so genanntes Thin-Client-Gerät, welches über ein integriertes Betriebssystem verfügt. Dieser stellt die Verbindung zum Server her.

> *Der Terminal-Server-Emulator ist die Client-Software, welche auf einem Win, Mac- oder Linux-Client installiert wird, um sich auf den Terminal-Server einloggen zu können und den Verbindungsaufbau zu starten.*

Ein wichtiger Bestandteil des Terminal-Servers, der die Kommunikation zwischen Server und Client ermöglicht, ist das Protokoll. RDP bietet bspw. 64.000 Transportkanäle für „Serial Device Communication“ und „Presentation Data“ sowie verschlüsselte Maus- und Keyboard-Daten. Das Design von RDP ermöglicht Verbindungen über verschiedenste Protokolle wie zum Beispiel ISDN, IPX, Netbios usw. Die aktuelle Version unterstützt jedoch nur noch TCP/IP.

Analog dem OSI-Layer-Prinzip werden Daten von Applikationen und Services durch die verschiedenen Protokoll-Stacks in Pakete und Rahmen aufgeteilt, verschlüsselt und gepackt und schließlich adressiert und über das LAN/WAN zum Client geschickt. Die Antwort-Daten des Clients werden in umgekehrter Reihenfolge verarbeitet und wieder ausgepackt, sodass sie im Server wieder der Applikation zur Verfügung stehen.

Verwendung

In der Architekturschicht besteht die Visualisierung bei Thin-Clients nur noch aus der Darstellung und der Input-Steuerung (Terminal-Paradigma).

Ein so genannter „Terminal-Server-Listener“ erkennt eine Client-Anfrage und initialisiert eine neue Instanz des RDP-Stacks, welcher er dann die eingehende Verbindung übergibt um sogleich wieder den TCP-Port nach neuen Anfragen abzuhören.

> *Bei jeder neuen Session wird als Erstes der Verschlüsselungsgrad (vom Client her bestimmt) ermittelt, wobei der Terminal-Server deren drei anbietet: low, medium und high.*

Bei allen Clients werden Speicher zum Cachen von Bitmaps (Icons, Toolbars, Cursors) reserviert. Der Terminal-Server besitzt ebenfalls Buffer, um Bildschirm-Refreshes kontrolliert zu übermitteln (anstelle eines konstanten Bitstreams). Bei hoher Interaktivität der Benutzer (Keyboard- und Mausbewegungen) wird dieser Buffer ca. 20-mal pro Sekunde geflutet, bei ruhigem Interaktionsverhalten geht diese Rate zurück auf 10-mal pro Sekunde. Die Belastung derVerbindung wird also so niedrig wie möglich gehalten.

Seit geraumer Zeit bietet nun MS den „Terminal Services Advanced Client“ (TSAC) an. Dieser Release[xi] beinhaltet Komponenten bezüglich Konfiguration und Installation eines Terminal Services. Somit lässt sich theoretisch in jede Applikation ein Fenster einbauen, welches einen Zugriff auf einen Terminal-Server erlaubt. Als Erstes benötige ich das ActiveX Client Control, das in die Palette aufgenommen wird:

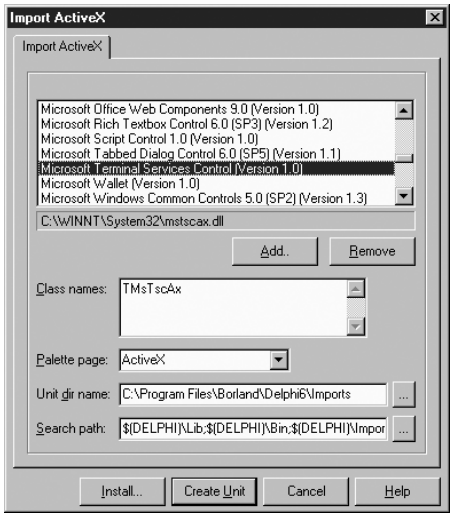

Abb. 3.50: Importieren des Terminal Toolkits

Dieses Control bietet volle Funktionalität einer Anwendung, wie bei einem installierten Terminal Services Client, wurde aber für eine autonome EXE oder eine Website mit ActiveX entworfen. Der Client benötigt noch die Datei *mstscax.dll*, die man zuvor mit regsvr32 registriert hat.

Im Folgenden will ich die konkreten Schritte aufzeigen, um diese Architektur einzusetzen:

Die ActiveX-Komponente bietet folgende Eigenschaften:

- Server: Der Terminal-Server-Name oder die TCP\IP-Adresse
- Domain: Benutzername mit Passwort oder die zugehörige Domain
- BitmapPeristence: die Einstellung des Cache

Mit dieser Information erstelle ich eine erste Startroutine:

```
procedure TfrmMain.btnConnectClick(Sender: TObject);
var pathToRun: string[255];
begin
pathToRun:='C:\WINNT\system32\clock.exe';
  with TfrmTerminal.Create(Application) do begin
    MsTscAx1.Server:= edtServer.Text;
    MsTscAx1.UserName:= edtUserName.Text;
    MsTscAx1.Domain:= edtDomain.Text;
    MsTscAx1.AdvancedSettings.BitmapPeristence:= 1;
    MsTscAx1.SecuredSettings.StartProgram:= pathToRun;
    ....
```

Und schon steuern wir die unter dem Pfad angegebene Uhr (clock.exe) auf dem Server fern. Die Verbindung selbst erfolgt über ein Event:

```
  MsTscAx1.Connect;
```

Durch die direkte Methode `disconnect` schließen wird den Kanal, aber die Session auf dem Server ist noch nicht geschlossen. Demzufolge muss ich zuerst den Client auf dem Server schließen (die ferngesteuerte Uhr), sobald ich ein `onDisconnected` erhalte, sodass ich auch mein Form beenden kann:

```
procedure TfrmTerminal.FormClose(Sender: TObject;
                                 var Action: TCloseAction);
begin
  if not frmMain.btnConnect.enabled then begin
    messageDlg('close server app', mtInformation, [mbOK], 0);
  Action := caNone;
 end;
end;

//Then we receive onDisconnected so we can close the form:
procedure TfrmTerminal.MsTscAx1Disconnected(Sender: TObject;
                                   DisconnectReason: Integer);
begin
  Close;
end;
```

3.3.14 Transaction

Ein nachrichtenorientiertes Architekturmuster (horizontales sicheres Befehlsmuster).

Zweck

Die Transaktionsunterstützung ermöglicht das Gruppieren bestimmter Aktionen in Transaktionen, die sich durch Quittieren bestätigen lassen.

Motivation

Werden in einer verteilten Datenbankanwendung beispielsweise die Datensätze mit einer Verbindungskomponente zwischen den Standorten gesendet, kann man die Methoden zum Hinzufügen und Löschen in dieselbe Transaktion aufnehmen. Dadurch wird entweder die gesamte Übertragung durchgeführt oder der vorherige Status wiederhergestellt. Transaktionen vereinfachen die Wiederherstellung nach einem Fehler in Anwendungen, die auf mehrere Datenbanken zugreifen müssen.

Transaktionen stellen Folgendes sicher:

- Innerhalb einer einzelnen Transaktion werden alle Aktualisierungen geschrieben oder rückgängig gemacht und auf ihren vorhergehenden Status gesetzt. Dies wird als Atomizität bezeichnet.
- Eine Transaktion ist eine korrekte Transformation des Systemstatus, wobei die Statusinvarianten beibehalten werden. Dies wird als Konsistenz bezeichnet.
- Parallel ausgeführte Transaktionen sehen gegenseitig nicht ihre unvollständigen und nicht geschriebenen Ergebnisse, was zu Inkonsistenzen des Anwendungsstatus führen könnte. Dies bezeichnet man als Isolierung.

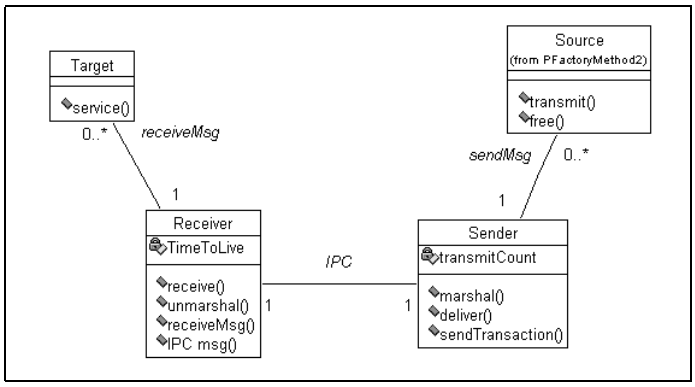

Abb. 3.51: Eindeutige Übermittlung zwischen Source und Target

Transaktionsdienste sind so robust, dass Vorgänge entweder immer vollständig ausgeführt oder rückgängig gemacht werden (der Server führt nie Teilaufträge aus). Die eingetragenen Aktualisierungen der verwalteten Ressourcen (z. B. Datensätze) überdauern selbst Kommunikations-, Prozess- oder Serverfehler. Dies bezeichnet man als Resistenz. Durch die Transaktionsprotokollierung ist es möglich, nach Ausfällen von Datenträgern den resistenten Zustand wiederherzustellen.

Manstelle sich einen Applicationloader vor, der die vier Aktionen:

Anfragen – Applikation suchen – Transportieren – Applikation starten

in einer Transaktion nach dem Motto „Alles oder Nichts“ ausführt. Diesem Loader gebe ich die Bezeichnung „Delphi Web Start“ (DWS). Somit sollten auf folgender Tabelle 3.2 Target und Source eine Brücke bilden:

Target	**Receiver**	**Sender**	**Source**
Programm	TCPClient	TCPServer	*.exe
Film	TCPClient	TCPServer	*.avi
Telnet	TCPClient	TCPConnection	Port: 9010
Form	DWSClient	DWSServer	Monitor

Tab. 3.2: Möglichkeiten einer Transaktion

Verwendung

Ich realisiere DWS mit den Indy-Komponenten, vor allem dem TCPServer. Diese Komponente kapselt einen vollständigen TCPServer (Transmission Control Protocol) mit Multithread-Unterstützung. Der TCPClient implementiert die Client-Funktionen des TCP-Protokolls einschließlich Sockets-Unterstützung. Dies kann man direkt oder in abgeleiteter Form für angepasste Client-Anwendungen verwenden.

> *Die Indy-Client-Komponenten sind einfach in der Handhabung, weil Sie Ihre Transaktionen in einer Folge schreiben und die Server multithread-fähig sind.*

Nachfolgend sei die konkrete Anwendung im Paketdiagramm aufgezeigt, die auch eine horizontale Struktur benötigt:

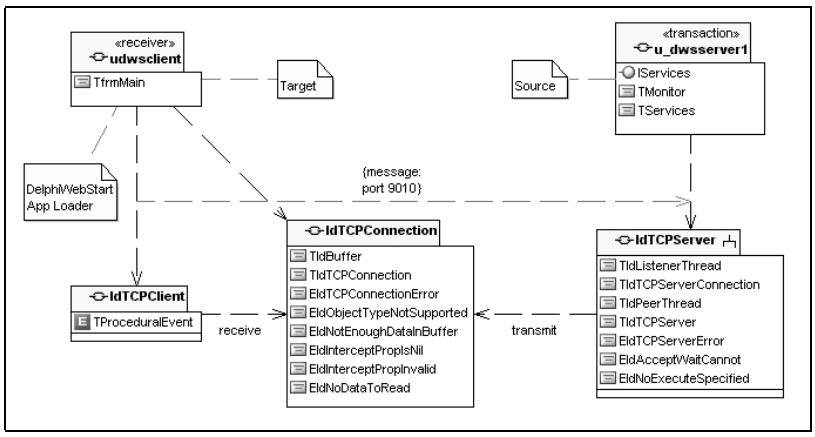

Abb. 3.52: Eindeutige Übermittlung zwischen Source und Target

Der DWS-Client erhält eine Liste der startfähigen Applikationen. Nach der Auswahl der gewünschten Anwendung startet der Benutzer die Transaktion, bis die Applikation auf dem Client transportiert ist und sich selbst startet.

Ein Marshaling zwischen Source und Target wird nicht benötigt, da DWS den Source ja nach dem Laden auf dem Target startet und betreibt.

> *Unter Marshaling wird der Mechanismus verstanden, der es einem Client ermöglicht, Aufrufe von Schnittstellenfunktionen von Remote-Objekten in einem anderen Prozessraum oder auf einem anderen Computer durchzuführen.*

Als Erstes definiere ich zwei so genannte „command handlers“, die als Nachrichten zwischen TCPServer und Client korrespondieren müssen:

```
CTR_LIST = 'return_list';
CTR_FILE = 'return_file';
```

Nach dem ersten Befehl erhalte ich die Liste der verfügbaren Dateien:

```
procedure TForm1.IdTCPServer1Execute(AThread: TIdPeerThread);
.....
// comes with writeline from client
if sRequest = CTR_LIST then begin
  for idx:= 0 to meData.Lines.Count - 1 do
  athread.Connection.
            WriteLn(ExtractFileName(meData.Lines[idx]));
  aThread.Connection.WriteLn('::END::');
  aThread.Connection.Disconnect;
.....
```

Zwei Threads sind durch Indy selbst in den Umlauf gekommen, erstens ein Listener, der auf eine Verbindung wartet, und zweitens der Servicethread, der den Transport einleitet. Diesbezüglich muss ich mich nicht um die Threadverwaltung kümmern, da der zugehörige Thread jeweils initialisiert übergeben wird:

```
IdTCPServer1Execute(AThread: TIdPeerThread)
```

Diese Technik ist mächtig, da der Client jederzeit eine Verbindung aufbauen kann, auch bei Dutzenden von bestehenden und aktiven Verbindungen zum Server. Der zweite Befehl `CTR_FILE` transportiert nun die Anwendung via Stream zum Client:

```
if Pos(CTR_FILE, sRequest) > 0 then begin
    iPos := Pos(CTR_FILE, sRequest);
    FileName := GetFullPath(FileName);
    if FileExists(FileName) then begin
      lbStatus.Items.Insert(0, Format('%-20s %s',
              [DateTimeToStr(now), 'Transfer starts ...']));
      FileStream := TFileStream.Create(FileName, fmOpenRead +
                                       fmShareDenyNone);
      aThread.Connection.OpenWriteBuffer;
      aThread.Connection.WriteStream(FileStream);
      aThread.Connection.CloseWriteBuffer;
      FreeAndNil(FileStream);
      aThread.Connection.Disconnect;
.....
```

Laden und Starten erfolgt auf Win wie auf Linux, denn DWS ist eine CLX-Anwendung. Unter Windows werden aber Textzeilen mit CR/LF (d. h. ASCII 13 + ASCII 10) abgeschlossen, unter Linux mit LF. Der Code-Editor kann mit diesem Unterschied zwar umgehen, aber wenn Sie Code aus Windows importieren, sollten Sie diesem Punkt besondere Aufmerksamkeit schenken:

```
   {$IFDEF LINUX}
      execv(pchar(filename),NIL);
```

```
   //libc.system(pchar(filename));
  {$ENDIF}
  {$IFDEF MSWINDOWS}
  with lbstatus.items do begin
    case  shellapi.shellExecute(0,'open', pchar(filename),
                               ',NIL,SW_SHOWNORMAL) of
      0: insert(0, 'out of memory or resources');
      ERROR_BAD_FORMAT: insert(0, 'file is invalid in image');
      ERROR_FILE_NOT_FOUND: insert(0,'file was not found');
      ERROR_PATH_NOT_FOUND: insert(0,'path was not found');
    end;
    Insert(0, Format('%-20s %s',
            [DateTimeToStr(now), filename + ' Loaded...']));
  end
  {$ENDIF}
```

Mit dem Marshaling wären zusätzlich Aktionen möglich, die parallel auf dem Server laufen könnten. Für jeden Schnittstellenaufruf schiebt der Client Argumente auf einen Stack und führt über den Schnittstellenzeiger einen Funktionsaufruf durch. Wenn der Objektaufruf nicht innerhalb des Prozesses erfolgt, wird er an den Proxy-Server übergeben.

Der Proxy-Server stellt die Argumente in ein Sequenzpaket und überträgt diese Struktur an das Remote-Objekt. Der Stub des Objekts „entpackt“ das Paket wieder, schiebt die Argumente auf den Stack und ruft die Implementierung des Objekts auf dem Server auf. Das Wesentliche bei diesem Vorgang ist, dass das Objekt den Aufruf des Clients in seinem eigenen Prozessraum erzeugt und ausführt.

Diese Architektur wäre dann mit dem in Abb. 3.53 gezeigten Paketdiagramm möglich.

Folgend sehen Sie noch den Client-Aufruf mit der einfachen Konfiguration von Port und Host, zusätzlich benötigt der Client noch die Datei *qtintf.dll*:

```
  with IdTCPClient1 do begin
    if Connected then DisConnect;
    showStatus;
    Host:= edHost.Text;
    Port:= StrToInt(edPort.Text);
    Connect;
    WriteLn(CTR_LIST);
...
```

3.3.15 Watchdog (Safety)

Ein nachrichtenorientiertes Architekturmuster (fehlertolerante Echtzeit-Systeme).

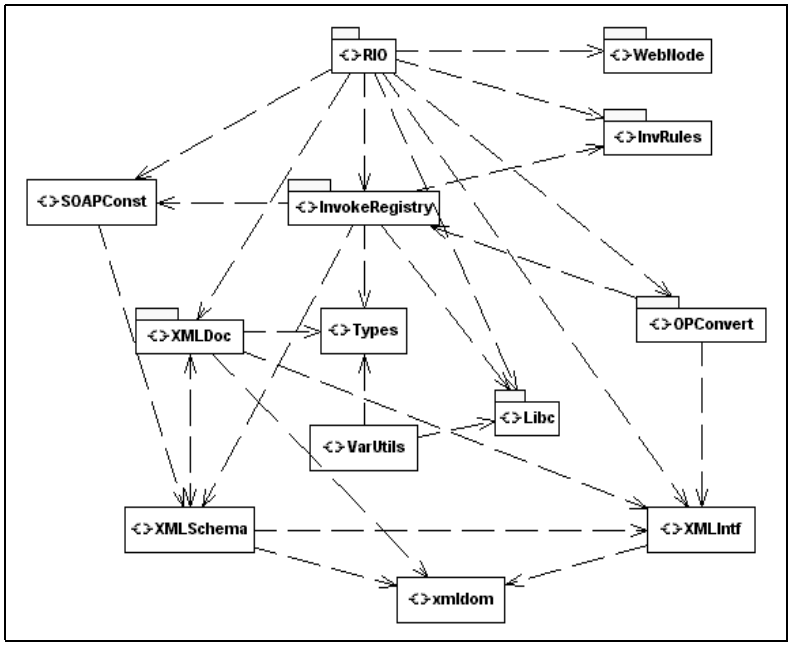

Abb. 3.53: DWS mit zusätzlichem Marshaling über SOAP

Zweck

Gegeben sei ein Mechanismus und eine Architektur, die kritische Systeme laufend auf Verfügbarkeit prüft und bei fehlender Reaktion Maßnahmen einleitet.

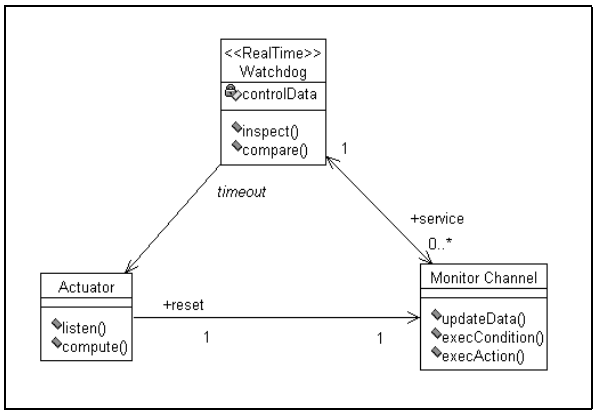

Abb. 3.54: Watchdog mit Recovery-Mechanismus

Motivation

Echtzeit- wie auch Embedded-Systeme kennen das Konzept des Watchdog, der vor allem der Sicherheit gewidmet ist und sowohl bei Herzschrittmachern wie auch bei der Flugüberwachung im Einsatz ist. Gewöhnlich wird bei solchen System auch jede Zeile Code durch einen Testfall motiviert, der zunächst fehlschlägt, d. h., man versucht mit allen Mitteln, eine Dysfunktion zu provozieren.

Hierarchische Systeme setzten sich typischerweise aus nur wenigen Arten von Subsystemen zusammen, in zahlreichen Kombinationen und Anordnungen. Ein komplexes System, das funktioniert, hat sich mit Sicherheit aus einem einfachen System entwickelt, das funktionierte. Ein komplexes System, das von Grund auf entworfen wird, funktioniert niemals und kann auch nicht dazu gebracht werden, zu funktionieren.

Wenn dann die Sicherheit zur Laufzeit hinzukommen muss, ist der Watchdog der ideale Partner. Ein Watchdog ist ein Subsystem, der in periodischen Abständen Signale oder Sequenzen von einem Monitor-Channel oder einem anderen Subsystem erhält. Ist ein Service oder eine Sequenznummer zu spät, aktiviert ein zugehöriger Activator Korrekturmaßnahmen wie reset, shutdown, Alarmieren oder ein Aufschalten eines anderen Subsystems.

Verwendung

Watchdogs sind stark auf einen Monitormechanismus angewiesen, der bspw. folgende Signale oder Kenndaten gemäß Real Time UML liefert:

- Timeouts durch andere Subsysteme
- Periodische Zusicherungen
- Kontinuierliche Build In Tests (BIT)
- Fehleridentifikationen durch Diagnosedaten

Diese Monitor-Architektur sei in folgendem Diagramm (s. Abb. 3.55) gezeigt (Bsp. IPC-Demos).

Schließlich muss sich der Watchdog auf die erhaltenen Daten verlassen können, so gehören Kritische Sektionen, ein Datenlogger, Ereignisverarbeitung und die hohe Verfügbarkeit mithilfe von Threads dazu. Falsche oder verloren gegangene Signale können aus statistischer Sicht verheerender sein als gar kein Signal.[8]

Ein einfacher Watchdog lässt sich schon mit einer DLL oder (SharedObject unter Linux) und einer Callback-Funktion realisieren.[xii] Ein Monitor übergibt der DLL als Subsystem eine Funktion, die dann periodisch aufgerufen wird. Der Monitor hat folgende fünf Elemente bezüglich Callback:

```
   interface...
1. TCallBackFunction = function(sig: integer):boolean;
2. function callME(sig: integer):boolean;
   implement...
```

[8] Schweigen ist Gold?

```
3. procedure TestCallBack(myCBFunction: TCallBackFunction);
                      register; external('watchcom.dll');
4. function callMe(sig: integer): boolean;
   begin
     //whatever you need to do, case of...
     showmessage('I was called with'+ inttostr(sig));
   end;
5. procedure TForm1.Button1Click(sender: TObject);
   begin
     testCallBack(callMe);  //subscribe function in DLL
   end;
```

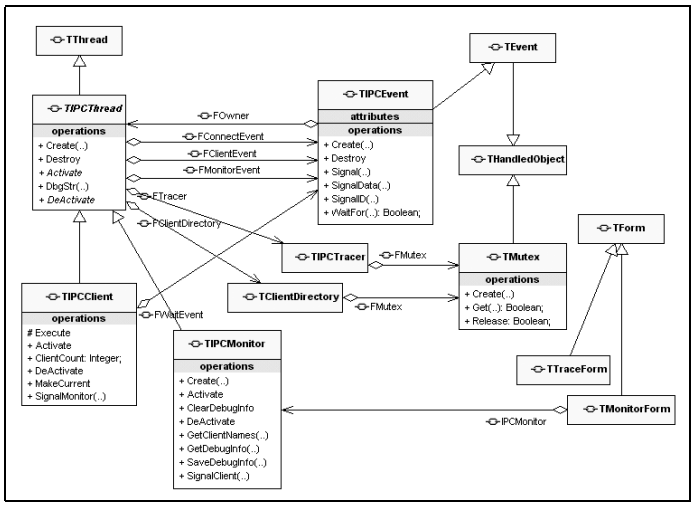

Abb. 3.55: Watchdog mit TCP-Monitor-Framework

In der DLL als Subsystem bietet sich dann die Möglichkeit, mit dem flexiblen Callback nicht eine fest verdrahtete Funktion aufrufen zu müssen. Somit hat auch der Monitor bei der Registrierung der Funktion Gewissheit, dass ein Subsystem existiert. Dieses sieht exemplarisch mit eingebautem Callback (Design by Contract) so aus:

```
Type
TCallBackFunction = function(sig: integer):boolean;

procedure TestCallBack(clientFunc: TCallBackFunction);
var sigAlive: boolean;
begin
  {timer stuff...
  set the signal...}
  if(clientFunc(55)) then sigalive:= true;
```

```
 end;
exports TestCallBack;
```

Im Gegensatz zu anderen Verbindungskomponenten sind über auf HTTP basierende Verbindungen keine Callbacks möglich. Als zusätzliche Sicherheitsmaßnahme muss der Monitor seine Verfügbarkeit auch für die Subsysteme garantieren (registrieren), die dann den Channel benutzen. Somit ist Sicherheit auch immerVerfügbarkeit!

3.4 Softwareklassen

3.4.1 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 eine Palette an, da die Nutzenbetrachtung voll zur Geltung kommt und die Verwaltung der Muster auch in Bibliotheken[9] 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. [xiii]

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 zugrunde liegenden Quelltext und deren Funktionalität. Diesen Vorgang bezeichnet man auch als Partitionieren von Applikationen, wie die Technik DLL+ im folgenden Diagramm zeigt:

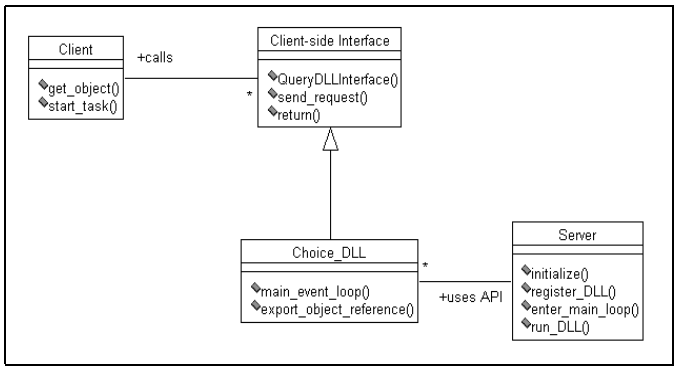

Abb. 3.56: Partitionieren mit DLL+ als OO-Library

[9] Bibliothek, Begriff aus einem Software-Kloster.

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 heraus entstanden sind. Solche nicht funktionalen Anforderungen wie Austauschbarkeit, Skalierbarkeit, Robustheit, Wartung oder Erweiterbarkeit der Software sind durch das Anwenden von Patterns und deren Bibliotheken implizit vorhanden.[10]

3.4.2 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: 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.*

[10] Patterns sind in Packages gespeichert.

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

3.4.3 Frameworks

„Programming for change“

Ein Framework ist ein wiederverwendbares, teilweise fertig gestelltes 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. 3.57), Crystal Reports, IntraWeb oder MetaBASE erwähnt . Das hat den Vorteil, dass der Entwickler sich auf die spezifischen Funktionen seiner Applikation konzentrieren kann. Frameworks kennt man eigentlich in jeder IDE, sofern eine Integration erfolgt ist.

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 stattdessen 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.

Aber auch externe Frameworks lassen sich in einer IDE verwenden. Beispiel eines externen speziellen Frameworks 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 strikt 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.

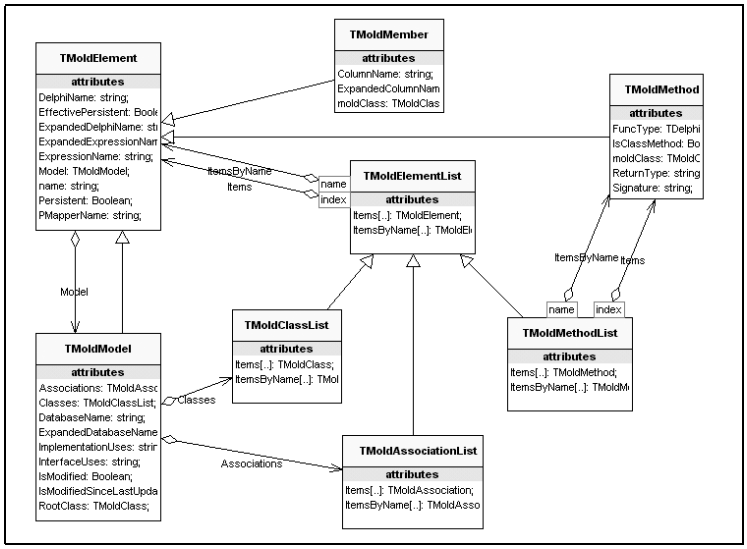

Abb. 3.57: Das Bold Framework der Metaklassen

3.5 Applikations-Framework

Microsoft hatte schon immer den Traum vom objektorientierten Betriebssystem. Als Steve Jobs 1988 mit seiner Next-Maschine und dem genialen, auf dem Mach-Kernel aufbauenden, NextStep die IT-Welt verblüffte, war Microsoft gerade dabei, etwas von Fenstern zu murmeln.

Jahre später entwickelten die Cracks aus Redmond ein eigenes Objektmodell, das Component Object Model (COM), und darauf aufbauend eine neue Version von OLE 2.0. COM ist keinesfalls auf OLE beschränkt, sondern universell einsetzbar, und wurde mit DCOM sogar netzfähig.

Die Schnittstellen sowie die Sprache von Delphi sind an der COM-Spezifikation ausgerichtet. Delphi implementiert Objekte entsprechend den COM-Spezifikationen durch eine Anzahl von Klassen, das sog. Delphi ActiveX Framework (DAX).

> *Auch die ActiveX-Komponenten sind aus dem COM abgeleitet. ActiveX Controls stellen eine schlankere Weiterentwicklung der OLE (OCX)-Controls für das Internet und die verteilten Welten dar.*

COM wurde ursprünglich konzipiert, um eine Kernfunktionalität für die Kommunikation zur Verfügung zu stellen und Erweiterungsmöglichkeiten zu bieten. Durch die Definition spezialisierter Schnittstellen für ganz bestimmte Zwecke wurde die Kernfunktionalität von COM selbst erweitert.

In Delphi wird der Begriff Transaktions-Objekte für Objekte benutzt, die von den Transaktionsdiensten, der Sicherheit und dem Ressourcenmanagement profitieren, die der Transaktions-Server MTS von MS (bis Win 2000) oder COM+ (ab Win 2000) bereitstellt. Diese Objekte sind für die Arbeit in einer großen verteilten Umgebung ausgelegt. Für Clients ist der Unterschied zwischen MTS und COM+ nicht von Bedeutung.

Die Idee, die Objektmodellen wie COM, CORBA oder OpenDoc zugrunde liegt, ist die gleiche wie bei einem Baukasten: Um große Anwendungen zu erhalten, konstruiert man zunächst eine Vielzahl von universell einsetzbaren **Grundbausteinen**.

Es geht schlussendlich darum, die Klassenbibliotheken und Komponenten der Compiler wie Delphi, C# oder C++ auf die Stufe Betriebssysteme zu bringen!

3.5.1 CLX und .NET

Viele Technologien zur Softwareentwicklung buhlen momentan um die Gunst der Entwickler. Java und das Komponentensystem EJB, auf der anderen Seite C# mit der .NET-Architektur und zu guter Letzt die Traditionalisten mit CLX, Delphi und C++. Vielfach wird auch die Echtzeit schlichtweg vergessen.[xiv] Die Gemeinsamkeiten sind in XML und UML und einer Service-basierten Architektur zu finden. Die benötigten Protokolle wie SOAP, IIOP, RMI basieren (oder sollten) längerfristig alle auf dem Prinzip eines Softwarebuses, der die skalierbare Kommunikation zwischen den Frameworks und Komponenten erlaubt.

> *Eine Komponentenbibliothek besteht aus Objekten, die in verschiedene Teilbibliotheken getrennt sind und jeweils einem bestimmten Zweck dienen.*

Der Aufbau von CLX und ihre Schnittstellen sind sehr ähnlich zur VCL, die Implementierung ist jedoch plattformunabhängig. Während die VCL auf den Windows-Steuerelementen aufsetzt, dient als Basis für CLX die bekannte Qt-Bibliothek von Troll Tech. Dadurch läuft CLX fast problemlos sowohl unter KDE als auch unter Gnome.

Alle CLX-Klassen sind von `TObject` abgeleitet. `TObject` führt Methoden ein, die grundlegende Verhaltensweisen wie Konstruktion, Destruktion und Nachrichtenverarbeitung implementieren. Die CLX-Hierarchie ist ähnlich strukturiert wie die VCL, allerdings werden die Steuerelemente hier als Widgets bezeichnet (TWinControl entspricht TWidgetControl usw.).

Da die Klassen, Eigenschaften und Methoden von CLX weit gehend identisch mit denen der VCL sind, sollte es keine großen Schwierigkeiten machen, bestehende VCL-Anwendungen nach Linux zu portieren bzw. sie plattformunabhängig zu machen. Voraussetzung ist natürlich, dass die Möglichkeiten der VCL studiert wurden und keine Windows-API-Funktionen direkt verwendet werden.

Neben den neuen Komponenten kommt mit Kylix auch auf dem Gebiet des Datenbankzugriffs etwas Neues. Die altehrwürdige BDE wird als einheitliche Zugriffsschicht durch

dbExpress ersetzt. Borland stellt mit dbExpress eine neue, leichtgewichtige Datenbanktechnologie vor, die einerseits sicherer und schneller und andererseits weniger aufwändig zu installieren ist.

Die wichtigsten CLX-Einheiten sind:

- BaseCLX: Betriebssystemnahe Klassen und Routinen, die für alle CLX-Anwendungen verfügbar sind. Es umfasst die Laufzeitbibliothek bis einschließlich der Classes-Unit.
- VisualCLX: Plattformübergreifende GUI-Komponenten und Grafik. VisualCLX nutzt die übergreifende Widget-Bibliothek (Qt).
- WinCLX: Klassen, die man nur auf Windows nutzen kann. Dazu gehören Steuerelemente für systemeigene Steuerelemente, Komponenten für den Datenbankzugriff, die mit Mechanismen wie BDE, COM oder ADO arbeiten, die unter Linux nicht zur Verfügung stehen.

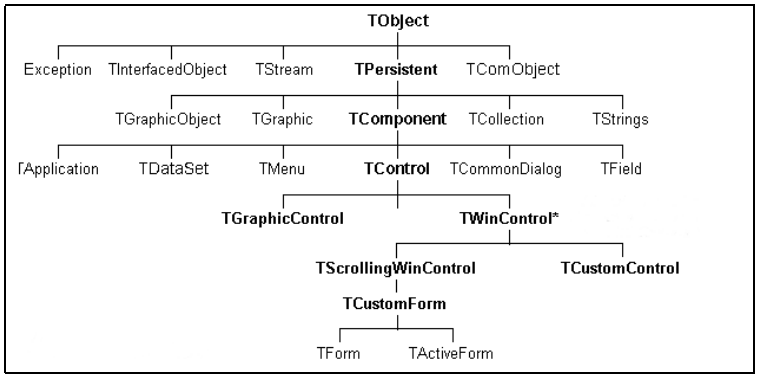

Abb. 3.58: CLX wie .NET sind sich in der Hierarchie ähnlich

Auch Klassen, die keine „Komponenten“ sind (d. h. CLX-Klassen, die von `TObject`, aber nicht von `TComponent` stammen), sind für verschiedene Aufgaben brauchbar. In der Regel dienen diese Klassen dem Zugriff auf System-Objekte (z. B. Dateien oder die Zwischenablage) oder zur Erledigung temporärer Aufgaben (z. B. Speichern von Daten in einer Liste).

Auf der anderen Seite steht .NET. Das MS.NET Framework und die Common Language Runtime (CLR) bieten eine Laufzeitumgebung, in der Komponenten, die man in den .NET-Sprachen programmiert, nahtlos miteinander interagieren können. Ein Compiler für eine .NET-Sprache gibt keinen systemeigenen Maschinencode aus. Stattdessen wird die Sprache in ein plattformneutrales Zwischenformat mit der Bezeichnung „MS Intermediate Language“ (MSIL oder IL) kompiliert. Die Module mit dem IL-Quelltext werden verknüpft und bilden eine Assembly. Eine Assembly kann aus mehreren Modulen bestehen oder nur eine einzelne Datei sein.

> *Wenn man diese beiden Applikations-Frameworks nun kreuzt, entsteht Octane (Delphi 8), das sowohl CLX wie auch .NET Support beinhaltet. Hier gilt: Zusammenarbeit bei den Spezifikationen – Wettbewerb bei den Implementierungen.*

Kriterium	CLX	.NET	J2EE
Technologie-Typ	Produkt	Produkt	Standard
Sprache	Delphi, C++	Diverse	Java
Middleware Vendors	Borland	Microsoft	Mehr als 40
Web-Technik	WebSnap/IW	ASP	JSP, Servlet
Business-Logic	DataSnap	Managed COM	EJB
Datentechnik	dbExpress	ADO	JDBC
Interpreter	Compiler	CLR	JRE
WebServices	Ja	Ja	Ja

Tab. 3.3: Applikations-Frameworks im Vergleich

Bezüglich Unabhängigkeit der Geräte ist ASP.NET zu erwähnen, das GUI Systeme ohne erneutes Codieren zu alternativen GUI-Systemen rendert. Sonst sind die Features von .NET denen von COM+ bezüglich der Laufzeit fast gleichzusetzen.

Eine Assembly ist die grundlegende Einheit für die Weitergabe in der .NET-Umgebung, und die CLR verwaltet das Laden, die Compilierung in den systemeigenen Maschinencode sowie die nachfolgende Ausführung dieses Codes. Anwendungen, die vollständig im Kontext der CLR laufen, lassen sich als „Managed Code“ bezeichnen.

3.5.2 Plattformen

Wie weit sind eigentlich Komponenten plattformunabhängig? War Java wirklich der Anfang? Eine gute Idee hat doch seinen Ursprung in Pascal! Nein, im Ernst, was die Rechnerunabhängigkeit von Software betrifft, gab es tatsächlich schon einen Versuch, der mir bekannt ist.

In meinen EDV-Frühtagen bewunderte ich des Öfteren die vielen dicken Bücher von UCSD-Pascal, die mich an eine Mischung zwischen Okkultismus und Fiktion erinnerten. UCSD-Pascal war eine Realisierung der University of California, San Diego.

> *Dieses Entwicklungssystem arbeitete mit einem Zwischencode, ähnlich dem Byte-Code von Java, für den es dann für alle Computertypen und Rechnerarchitekturen Interpreter geben sollte.*

Der Kommerzialisierung stand dann aber doch die fehlende Bereitschaft zur Verbreitung der «Virtual-Machines» entgegen. Zu dieser Zeit (1985) war die Vernetzung noch gering und das Web hatte noch nicht mal seinen Namen gefunden.

In der Regel ist die Funktionalität von Win-Anwendungen (VCL) und plattformübergreifenden Anwendungen (CLX) dieselbe. Allerdings können einige Win-spezifische Features nicht direkt auf Linux-Umgebungen übertragen werden. So sind beispielsweise

ActiveX, BDE, COM und ADO von der Win-Technologie abhängig und stehen unter Kylix nicht zurVerfügung.

Die Serveranwendung ist normalerweise für Linux oder Windows entwickelt, kann aber ebenfalls übergreifend sein. Die Clients können sich auf beliebigen Plattformen befinden. Bezüglich Web sind einfache CGI-Anwendungen und Anwendungen, die Web Broker oder WebSnap verwenden, sowohl unter Win als auch unter Linux einsetzbar.

Portierungen

Was gibt es für Portierungstechniken?

Plattformspezifische Portierungen sind häufig zeitaufwändig, teuer und produzieren ein Ergebnis, das nur für ein einziges Ziel vorgesehen ist. Sie erzeugen unterschiedliche Code-Grundlagen, wodurch sie insbesondere schwierig zu warten sind. Jede Portierung ist jedoch auf ein ganz spezielles Betriebssystem ausgelegt und kann plattformspezifische Funktionsmerkmale nutzen. Die resultierende Anwendung ist deshalb normalerweise schneller.

Plattformübergreifende Portierungen sparen in der Regel Zeit, da man die portierten Anwendungen auf mehreren Plattformen einsetzen kann. Allerdings ist der Arbeitsaufwand bei der Entwicklung übergreifender Anwendungen stark vom bereits existierenden Code abhängig. Wurde Code ohne Berücksichtigung der Plattformunabhängigkeit entwickelt, können Situationen entstehen, in denen die unabhängige Logik und die plattformabhängige Implementierung vermischt sind.

Der übergreifende Ansatz ist die zu wählende Methode, weil Business-Logic auf plattformunabhängige Weise ausgedrückt wird. Einige Dienste werden hinter einer internen Schnittstelle abstrahiert, die auf allen Plattformen sehr ähnlich aussieht, aber jeweils spezifische Implementierungen aufweist.[xv]

Ein Beispiel dafür ist die Laufzeitbibliothek. Die Schnittstelle ist auf beiden Plattformen sehr ähnlich, während sich ihre Implementierung wesentlich unterscheiden kann. Man sollte übergreifende Komponenten isolieren und dann, darauf aufsetzend, spezifische Dienste implementieren. Insgesamt stellt dieser Ansatz eine optimale Lösung dar, weil durch die größtenteils gemeinsam genutzte Codebasis weniger Wartungskosten anfallen und eine verbesserte Anwendungsarchitektur entsteht.

Mit dem Aufkommen von Octane wird sicher ein neues Kapitel um die Schönste im Lande eröffnet. Wie sagte schon Goethe: „Die Sprache ist das Kleid der Gedanken“, demzufolge hat Borland wirklich das schönste Kleid. ;)

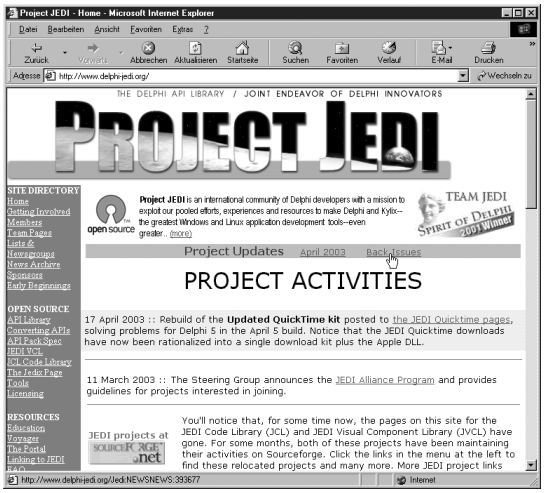

Abb. 3.59: Object Pascal the World

3.6 Anhang

3.6.1 Architektur-Idiome

Die fünf Architektur-Idiome befinden sich auf der CD-ROM.

3.6.2 Pattern-Buch-Overview

Kontext	Name	Typ	Verwendung
Prozess	V-Modell	Verhalten	TRadeRoboter
Prozess	RUP	Verhalten	MIL Sport
Prozess	XP	Verhalten	SMS-Gate
Design	Abstract	Erzeugung	Scripting
Design	Builder	Erzeugung	TForm
Design	Factory	Erzeugung	MTS Pooling
Design	Singleton	Erzeugung	Application
Design	Adapter	Struktur	WebSnap
Design	Bridge	Struktur	Sockets
Design	Composite	Struktur	Directory Dienst
Design	Decorator	Struktur	Stream Adapter

Kontext	Name	Typ	Verwendung
Design	Facade	Struktur	Tool API
Design	Flyweight	Struktur	Charts Fractal
Design	Proxy	Struktur	CORBA Stub Proxy
Design	Wrapper	Struktur	Data Link
Design	Chain	Verhalten	Exception Handling
Design	Command	Verhalten	Fraktale
Design	Interpreter	Verhalten	XML-DOM Parser
Design	Iterator	Verhalten	WebSnap
Design	Lock	Verhalten	Listenstrukturen
Design	Mediator	Verhalten	OLE DB
Design	Memento	Verhalten	TRecall
Design	Observer	Verhalten	SQL-Monitor
Design	State	Verhalten	Delegates
Design	Strategy	Verhalten	Decision Cube
Design	Template	Verhalten	Stream Classes
Design	Visitor	Verhalten	CASE Tool
Architektur	Automation	Kombiniert	OPC-Server
Architektur	Broker	Verarbeitung	SOAP, dbExpress
Architektur	Container WP	Verarbeitung	DLL+, Rates
Architektur	Layers	Kombiniert	Broker Bank
Architektur	Master-Slave	Nachrichten	Sort Multithreading
Architektur	Microkernel	Verarbeitung	Base Component
Architektur	Monitor	Daten	Scanner, E-Commerce
Architektur	MVC	Nachrichten	Boldsoft
Architektur	PAC	Kombiniert	Boldsoft
Architekur	Prototype	Kombiniert	HTML, COM
Architektur	Provider	Daten	Data Set Provider
Architektur	Simulator	Kombiniert	Trade Simulation

Kontext	Name	Typ	Verwendung
Architekur	Terminal-Server	Kombiniert	MS TSAC
Architektur	Transaction	Nachrichten	DelphiWebStart
Architektur	Watchdog	Nachrichten	TCP Monitoring

Tab. 3.4: Alle Patterns in der Übersicht

3.6.3 UML Overview

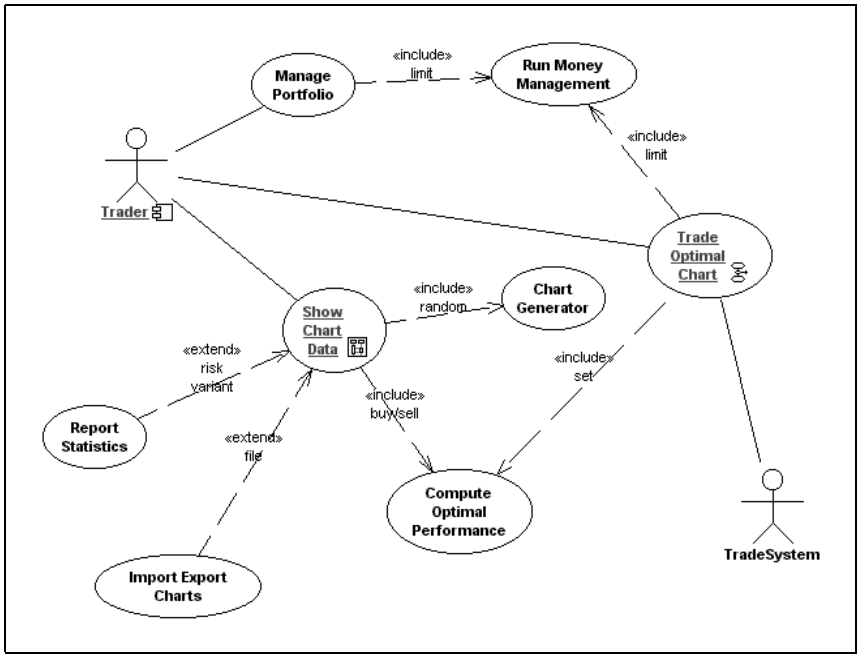

Abb. 3.60: Use Case in der Analyse

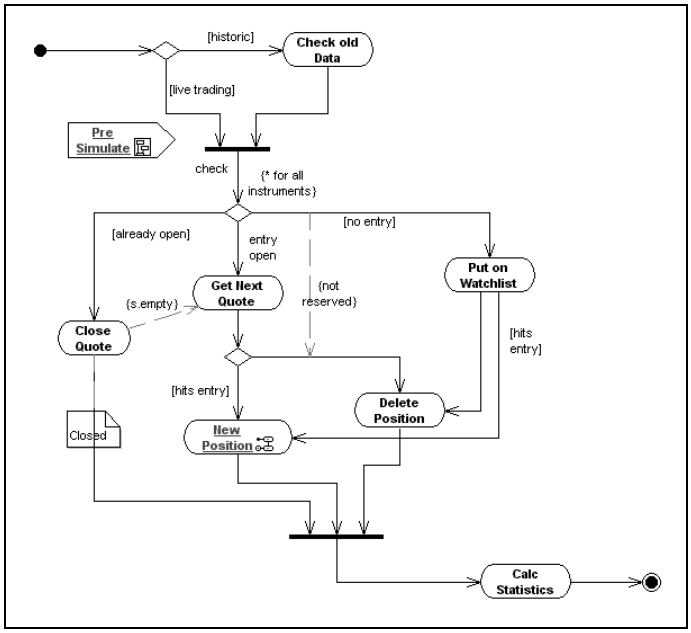

Abb. 3.61: Activity in der Analyse

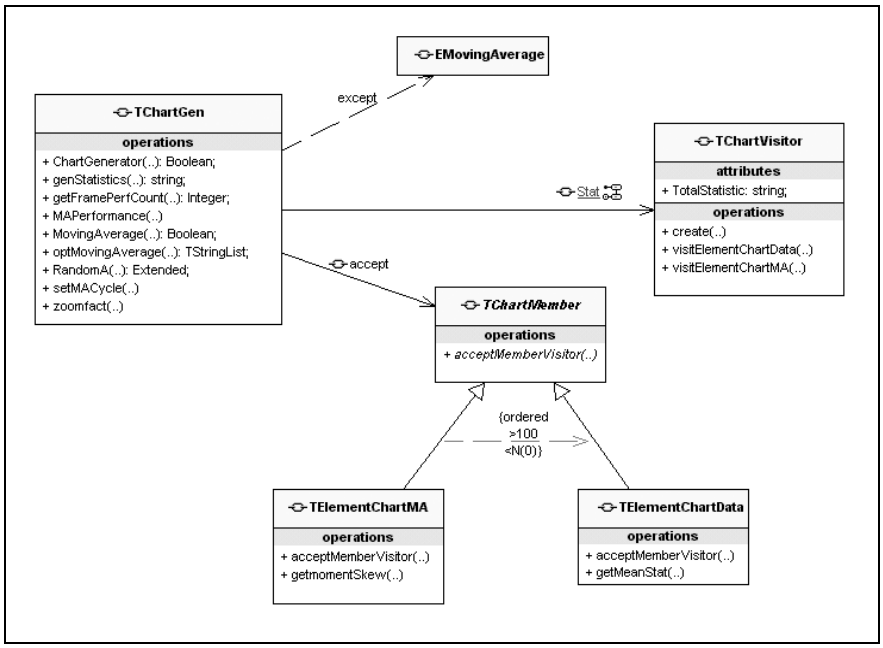

Abb. 3.62: Class im Design

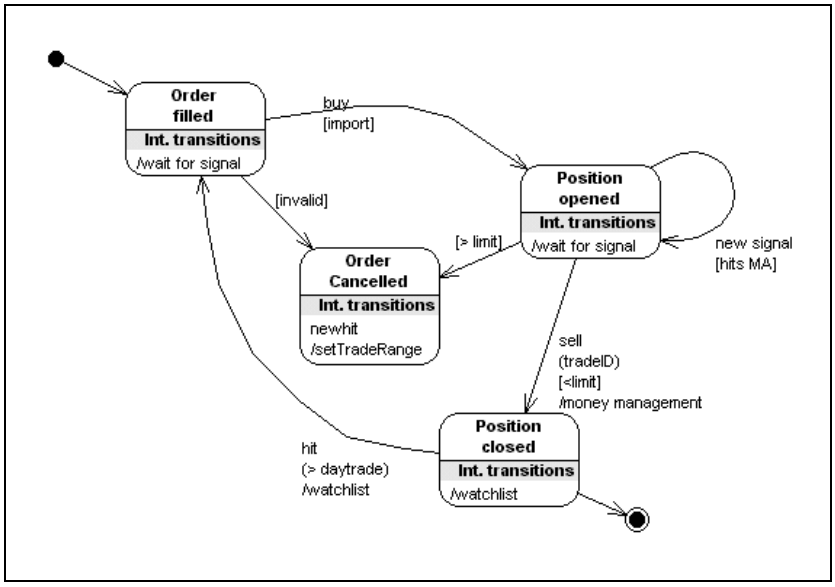

Abb. 3.63: State Event im Design

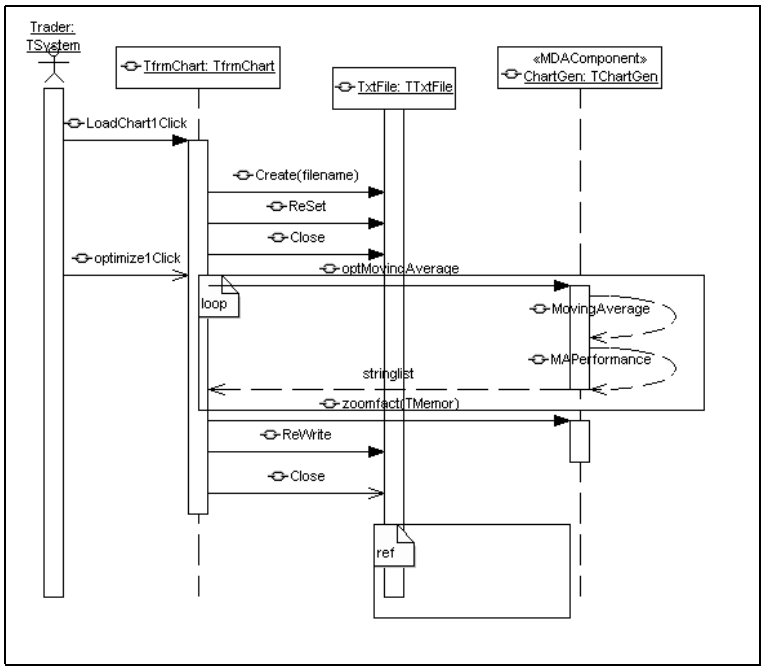

Abb. 3.64: Sequence in der Implementierung

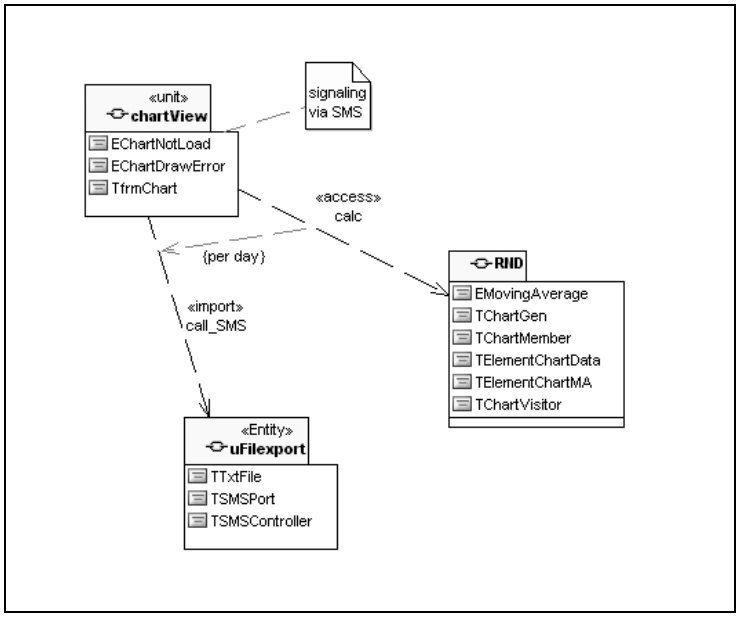

Abb. 3.65: Packages in der Implementierung

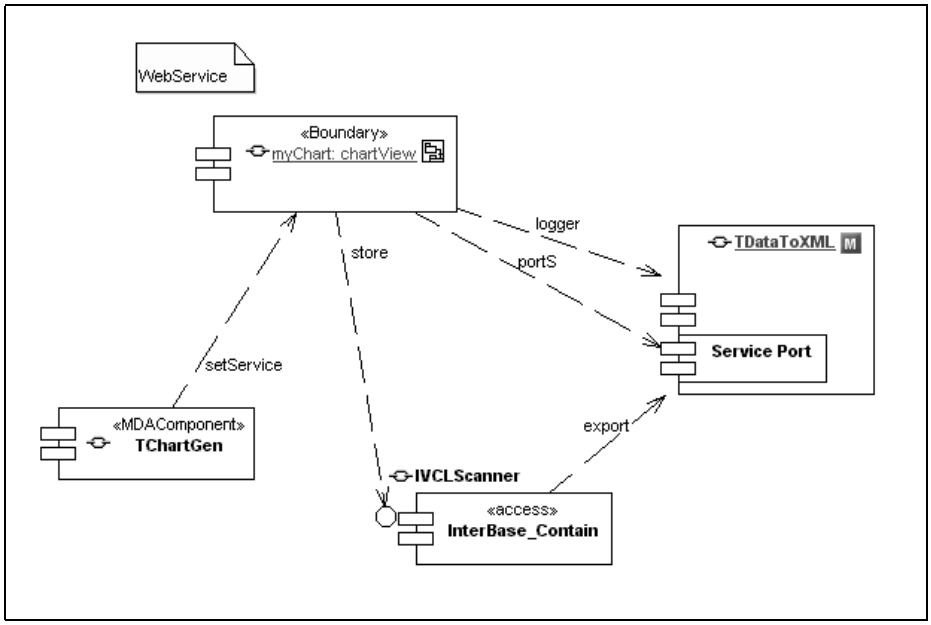

Abb. 3.66: Components in der Integration

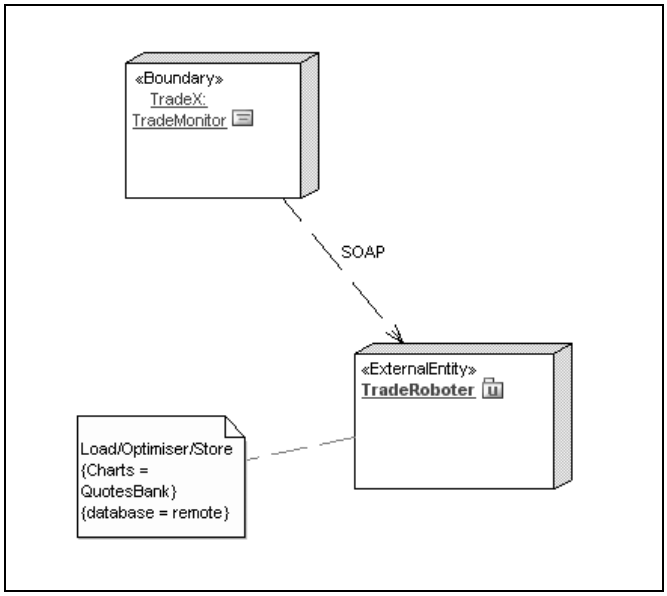

Abb. 3.67: Deployment in der Integration

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

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

[iii] Götz, Mende: MSR mit Delphi, Franzis Verlag, 2001

[iv] Fowler, M.: Analysis Patterns – Reusable Object Models, Addison-Wesley, 1997

[v] Jacobson, I., Ericcson, M.: The Object Advantage – Business Process Reengineering with Object Technology, Addison-Wesley, 1994

[vi] Douglas B.P.: Real-Time UML, Addison-Wesley, 1998

[vii] Struts: http://jakarta.apache.org/struts/

[viii] Chappell: Understanding ActiveX and OLE, A Guide for Developers & Managers, ISBN 1-57231-216-5

[ix] Kleiner, M.: Cash oder Crash?, c't 12/94, S.194 ff.

[x] Kleiner, M.: Ein optimaler Moving Average, UNI Bern, 1995

[xi] Terminal: http://www.microsoft.com/windows2000/downloads/recommended/TSAC/

[xii] Harel, David: Algorithmics, Addison Wesley Longmann, 1993

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

[xiv] Ward P.T and Mellor S.J: Structured Development for Real Time Systems, Vol 1-3, Prentice Hall (Yourdon Press), 1985

[xv] Booch, G.: Object Oriented Analysis and Design with Applications, 2nd Ed., Benjamin Cummins, 1993