1 Pattern-Techniken

1.1 Prozess-Patterns

1.1.1 Prozessmodelle

Der Einsatz von Patterns ist meist mit einem Tool und einem entsprechenden Vorgehen verbunden, sodass ein Prozess selbst schon ein Mustervorgehen sein kann. Oft hört man ja den Spruch vom musterhaften Vorgehen und meint doch eher: „Wenn schon nicht kreativ, dann doch vorbildlich.“ Und genau das Wort „vorbildlich“ ist die Kraft jedes Patterns, Sinnbild für ein Diagramm mit Wiederverwendbarkeit. Welches Vorgehen im Einsatz von Patterns ideal ist, soll ein erster Abschnitt klären.

Die weiteren Kapitel in Teil 1 gehen dann auf die Techniken und Tricks ein, die mit dem Einsatz von Patterns häufig zum Tragen kommen.

Die Unified Modeling Language (UML) bringt in der derzeitigen Version kein Vorgehensmodell (Prozessmodell) mit und wird auch keines mehr vorgeben wollen. Diese Einsicht trat schon im Juni '96 zu Tage, als das Gremium der Object Management Group (OMG) [i] die Unified Method in Unified Modeling umtaufte und so den Standard UML begründete.

Jeder Anwender dieses Standards ist deshalb gezwungen, seine eigene Vorgehensweise zu etablieren. Dies hat den Vorteil, UML in schweren (z. B. RUP) wie in leichten Prozessmodellen (z. B. XP) einsetzen zu können. Lange Zeit hat man in UML selbst einen Musterprozess vorgegeben, den „Objectory“, der sich aber nie wirklich etablieren konnte.

Sicher kennen Sie das klassische Wasserfallmodell, das den organisatorischen Ablauf analog zu einem Wasserfall beschreibt. Die einzelnen Projektzyklen sind in einer fest vorgegebenen Reihenfolge starr zu durchlaufen. Die jeweiligen Phasen laufen sequenziell ab, eine neue Phase wird erst begonnen, wenn die vorhergehende Phase abgeschlossen ist. Diese Art von Vorgehen findet man im Softwarebau höchstens noch auf dem Papier.

Auf der anderen Seite spricht man heutzutage von agilen Prozessen, welchen dann der Ruf von zu viel Freiheitsgraden anhaftet. Denn auf der einen Seite baut man bestimmte Praktiken der Softwareentwicklung weiter aus als andere, zum anderen lässt man einige etablierte Verfahren vollständig fallen. Diese Radikalität bringt z. B. XP häufig den Vorwurf ein, es mache die konzeptlose „Hackerei“ zum Prinzip.

Beim Versuch, ein Vorgehensmodell einzurichten, erscheint mir vor allem die Zuordnung von Aktivitäten zu Produkten (Ergebnisse) wichtig, denn nur so lässt sich ein konkreter Phasenplan aufstellen. Die im Voraus erwarteten Produkte (vor allem Dokumente, Diagramme und Code) sind Grundlagen von Reviews und dienen durch ständiges Hinterfragen auch der Qualitätssicherung.

Ein Prozessmodell stellt eigentlich eine vollständige Beschreibung des Software-Entwicklungsprozesses dar. Es definiert die Aktivitäten und Rollen, die bei der Entwicklung zu durchlaufen sind, und die Ergebnisse, die man dabei erzeugt.

Ferner gibt es über den Produktfluss Auskunft, d. h., es legt fest, welche Produkttypen der Entwickler in welchem Zustand als Eingangsinformation für seine nächste Aktivität erwartet und in welchem Zustand ein Produkttyp (z. B. ein Diagramm) aus einer Aktivität heraus dem Nächsten zur Verfügung steht.

Der Einsatz eines Prozessmodells bringt gleich mehrfachen Nutzen – innerhalb eines Projekts und über Projektgrenzen hinweg. In jedem Projektmanagement wirft man sich gerne so genannte W-Fragen an den Kopf, also etwa: Wer, Was, Wann und Wie? Ein Prozessmodell beantwortet also ständig die Frage: „Wer macht was wann und wie?“, indem es die Rollen aller Projektbeteiligten beschreibt und sie den Aktivitäten zuordnet.

Das heißt dann konkret, in der ersten Phase der Initialisierung beantwortet man sich die Frage, mit welchem Prozess und mit welchen Tools (Techniken) gelangt man vom Modell zum Code. Merke: Wer einen Prozess (zeitliches Vorgehen) und eine Technik einsetzt, der hat ein „methodisches“ Vorgehen.

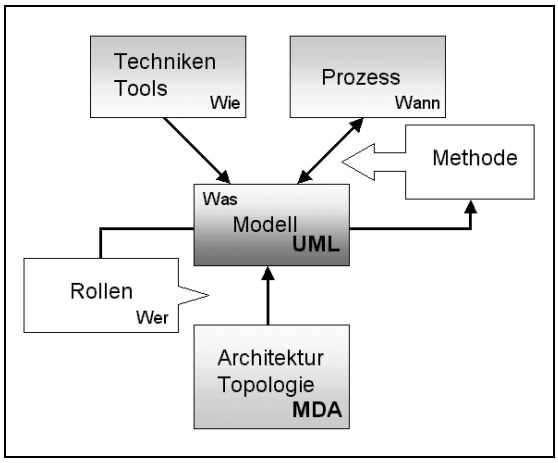

Abb. 1.1: Das Modellschema zum Prozess

UML bietet grafische Ausdrucksmittel an, um Anforderungen zu definieren sowie ein System fachlich und technisch zu entwerfen. Allerdings macht sie keine Aussage, wann, wie und von wem diese Mittel im Projektverlauf zum Zuge kommen. Während meinen Workshops taucht häufig die Frage auf, ob denn die Codegenerierung auch zu einem Prozess gehört.

Nun, wie man zum Modell/Code gelangt, ist eher eine Frage der Technik, z. B. eben mit Design Patterns, oder man generiert mit der MDA (Model Driven Architecture) schon in einer frühen Phase ganze Architektur-Patterns[ii]. Auch wenn es mittlerweile standardisierte Idiome wie getter- und setter-Methoden, IDL-Generierung oder Listenklassen gibt, die sich mit Code-Templates auch erzeugen lassen, hört die eigentliche Generierung innerhalb der Methode einer Klasse auf.

Bei der Implementierung einer Methode bzw. einer Member-Funktion verwendet man genau die algorithmischen Konstrukte, welche die bewährten Struktogramme oder Flow-

charts seit Jahren anbieten. Warum also nicht die klassischen Struktogramme zur Detailspezifikation von Methoden einsetzen, wenn es die Komplexität erfordert?

Auf der anderen Seite gibt es Prozesse, die ein modellgesteuertes Vorgehen vorsehen, sodass es z. B. aus dem Paketdiagramm auf der Stufe Design Patterns heraus möglich ist, Code fast konstruktionsfertig zu erzeugen. Wieweit diese Technik schon als Prozess genügt, zeige ich weiter unten innerhalb der MDA und UML 2.0 im Ansatz.

Jedes Pattern beschreibt eine Musterlösung für ein Problem, das immer wieder vorkommt. Dabei bezeichnet man mit dem Ausdruck Pattern sowohl das Ergebnis, das durch die Anwendung der Regel entsteht, als auch die Regel selbst.

> *So wenig wie ein Synthesizer[1] das Komponieren guter Musik garantiert, werden die neuen Architekturmethoden und Tools den Entwurf guter Bauten garantieren. Doch wie in der Musik werden sich auch die fähigen Entwickler der neuen Instrumente bedienen.*

In Tabelle 1.1 habe ich den Versuch unternommen, basierend auf dem Modellschema, eine Übersicht über die bekannten Prozesse herzustellen. Der Begriff „schwer“ ist im Gegensatz zu „agil“ zu betrachten. Fast allen gemeinsam ist der Einsatz der UML-Notation und Objektorientierung (OO), iteratives Vorgehen sowie ein vermehrtes Ausrichten auf Patterns. Auch die MDA ist eine Art prozessorientiertes Architektur-Muster:

Prozessmodell	**Techniken**	**Toolbeispiel**	**Modell**	**Organisation**
RUP	OO, MDA	Rose, XDE	UML	Schwer, Teams
V-Modell	OO, MDA	ObjectiF	UML	Mittelschwer
XP	OO, DUnit	ModelMaker	UML	Leicht, paarweise
SELECT	OO, SAD	S.Enterprise	SAD, UML	Schwer, Abteilung

Tab. 1.1: Prozessmodelle im Vergleich

1.1.2 RUP

Der Rational Unified Process (RUP) ist eine Sammlung von Best Practices, die sich in vielen Projekten bei Kunden und Ämtern bewährt haben. Er kombiniert Technologie und fachspezifische Anleitungen mit vielen Vorlagen und Beispielen.

Benutzer können Konfigurations- und Anpassungswerkzeuge der RUP-Plattform dazu nutzen, angepasste Projektvorgaben auf Basis ihrer spezifischen Anforderungen zu erstellen. Der RUP ist ein weit verbreiteter Standard, es zeigen sich aber deutliche Schwächen bei der Weiterentwicklung oder Änderung von Komponenten oder bestehenden Anwendungen, wenn die Organisation diese nicht von Anfang an mit RUP entworfen hat.

Im Zentrum von RUP steht die Bewältigung eines komplexen arbeitsteiligen, verteilten und lang andauernden Entwicklungsprozesses durch dessen Einteilung in definierte

[1] Synthesizer sind wie Compiler, einer baut, viele produzieren.

Produktionsschritte mit definierten Ergebnissen. Dieser tayloristische Ansatz entspricht dem Bestreben des Teams nach Kontrolle und Vorhersagbarkeit.

Der RUP[iii] beschreibt die internen Workflows mittels Aktivitätsdiagrammen. Es gibt sechs Kernaktivitäten, die das Vorgehen für Geschäftsmodelle, Anforderungsermittlung, Analyse und Design, Implementierung, Test und Verteilung festlegen (siehe Abb. 1.2). RUP legt bestimmte Rollen, Aktivitäten und Artefakte des Entwicklungsprozesses fest. Einer Rolle sind dann bestimmte Tätigkeiten und Artefakte (Produkte) zugewiesen.

> *Eine Aktivität ist ähnlich dem V-Modell eine Arbeitseinheit, die sich von einem Beteiligten in einer bestimmten Rolle bearbeiten lässt. Für viele der so genannten Artefakte liefert der RUP Vorlagen, z. B. um die Ergebnisse einer Systemabnahme festzuhalten.*

Auf den Vorwurf eines schweren Prozesses hat Rational reagiert. Rational und Object Mentor haben auf der Rational User Conference 2002 in Orlando ein Extreme Programming (XP) Plugin für den RUP vorgestellt. Die gemeinsame Entwicklung von Rational und des Vorreiters im XP-Bereich verknüpft Richtlinien und Best Practices von XP mit den Kernelementen des RUP.

Damit sollten Entwickler in der Lage sein, Anwendungen schneller und flexibler zu schreiben. Rational und Object Mentor gehören zu den Gründern der Agile Alliance [iv], die schnellere und unkompliziertere Softwareentwicklung fördert. Wobei die Gefahr besteht, dass die Leitplanken außer Sicht geraten, wie auch Franz Zucol meint: “Lieber einen umfangreichen, größeren Rahmen mit der nötigen Freiheit als die große Freiheit, um dann doch die Rahmen bestimmen zu müssen.“

Das Plugin kombiniert die Kernelemente der iterativen Entwicklung und die Nutzung komponentenbasierter Architekturen aus RUP mit der flexiblen und schnellen Methodologie von XP. Besonders kleinen und mittelgroßen Teams gibt das Plugin mit einer „ready-to-use“-Sammlung von Praktiken und Aktivitäten Starthilfe bei der Entwicklung.

„Mit dem RUP-Plugin für XP können Entwickler Software einfacher und schneller erstellen, da es die grundsätzlichen Prinzipien des RUP mit den Werten und Best Practices des Extreme Programming verbindet“, so Robert C. Martin, CEO, Präsident und Gründer von Object Mentor.

Die aktuellen Bemühungen von Rational, den RUP XP-kompatibel zu machen, und die Diskussionen im XP-Umfeld über Erweiterungen der Praktiken zeigen, dass sich die beiden Sichtweisen nicht widersprechen, sondern ergänzen können. Kombinationen und Experimente sind aus meiner Sicht nicht nur erlaubt, sondern notwendig und gefragt.

Vorteile des RUP:

- Die entsprechenden Tools sind verfügbar
- Bewährte Sammlung an Vorlagen
- Enge Anbindung an die UML-Notation
- Quasi Standard mit breitem Kurs- und Literaturangebot
- Auch für mehrjährige Projekte geeignet

Nachteile des RUP:

- Große Erfahrung in Teams nötig
- Gefahr der Degeneration nach der Einführung

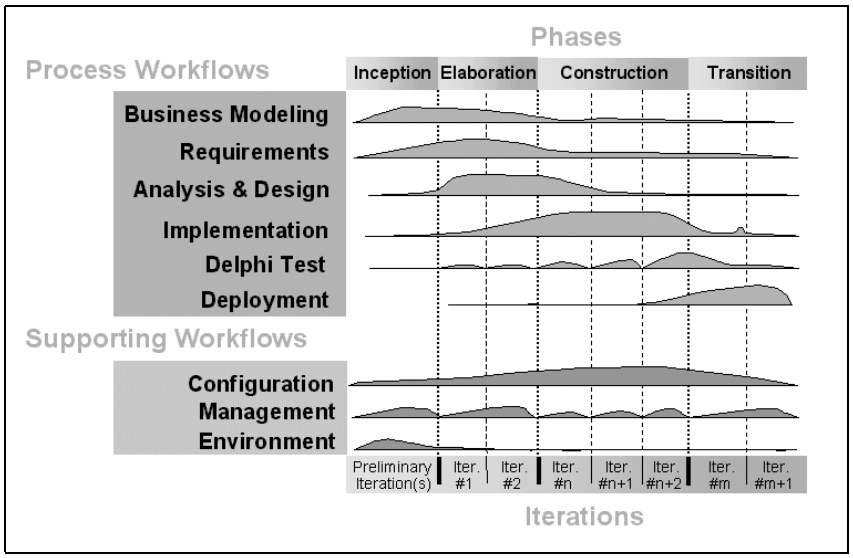

Abb. 1.2: Der RUP im Blick

1.1.3 V-Modell

Das V-Modell wurde ursprünglich in Deutschland im Auftrag des Bundesministeriums für Verteidigung entwickelt und ist dort seit 1992 verbindlich im Einsatz. Im Sommer 1996 wurde es, nach einer 1992 gestarteten Erprobungsphase, auch für den Einsatz im zivilen Verwaltungsbereich der Bundesbehörden empfohlen.

Damit existiert für die Entwicklung und Wartung von IT-Systemen ein einheitlicher Standard für den gesamten öffentlichen Bereich, der sich von öffentlichen Auftraggebern und privatwirtschaftlichen Auftragnehmern gleichermaßen nutzen lässt. Dieses Vorgehensmodell liefert die MID GmbH als vollständiges Referenzmodell aus und ist als Grundlage oder Instanz für ein entsprechendes Projekt anzuwenden.

Im eigentlichen V-Modell kann der Benutzer dann das Vorgehensmodell, das als Referenzmodell dient, an das konkrete Projekt anpassen. Hierzu kann wie in unserem Fall eine V-Matrix dienen. Dabei bedient man sich vorab spezifizierter Anpassungsregeln, die zu einem konsistenten und standardisierten Projekt führen.

Nebst der eigentlichen Softwareentwicklung (SE) gibt es noch drei andere so genannte Submodelle, die als Begleitaktivitäten zur eigentlichen Entwicklung zu betrachten sind. Es sind dies im Einzelnen:

- PM für Projektmanagement
- SE für Systementwicklung

- KM für Konfigurationsmanagement
- QS für Qualitätssicherung

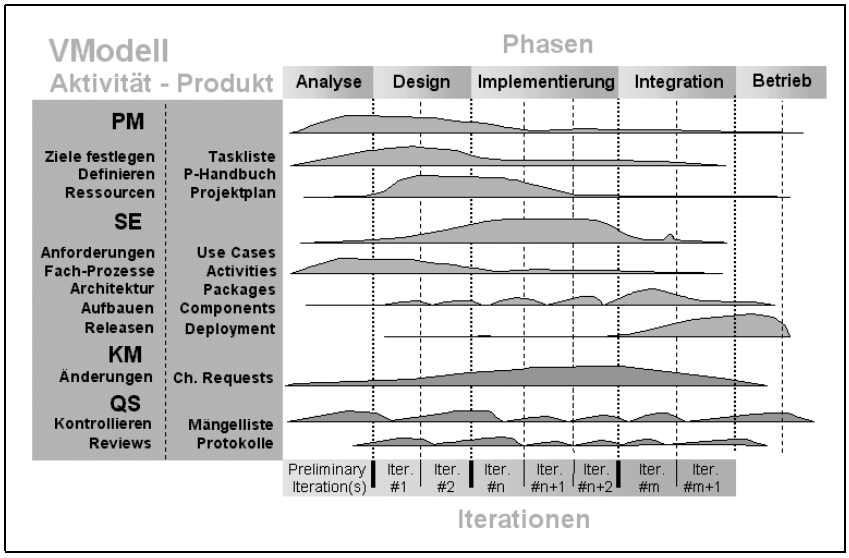

Abb. 1.3: Iteratives Vorgehen mit V

In den letzten Jahren hat sich in vielen Unternehmen die Einsicht durchgesetzt, dass man Softwarequalität verbessern muss. Wie in anderen Bereichen auch liegt der Schlüssel zum Qualitätsmanagement in der Kontrolle des Produktionsprozesses. Nur mit einem nachvollziehbaren Vorgehen ist es möglich, die Qualität der erzeugten Softwaresysteme sicherzustellen.

Die 1997 freigegebene Version des V-Modells ist methodenneutral. Während die Version von 1992 nur die Softwareentwicklung nach strukturierten Methoden unterstützte, ist es nun auch möglich, moderne Paradigmen, insbesondere objektorientierte Techniken mit UML, in den Entwicklungsprozess einzubinden.

- System strukturieren – Packages
- Systemarchitektur verteilen – Dekomposition
- Anforderungen definieren – Use Cases

Bei einer so genannten Elementarmethode in V stehen das Wie und Wann im Vordergrund. Das bedeutet, eine Elementarmethode wie Dekomposition ergänzt die Aussage darüber, was im Rahmen einer Aktivität geschehen soll, durch die Anleitung, wie und wann es zu tun ist.

Die Anwendung von Elementarmethoden ist im Allgemeinen mit dem Einsatz spezieller Darstellungsmittel (meist Diagramme) verbunden, die den Produkttypen zuzuordnen sind. Beim Einsatz der UML mit dem V-Modell ist jedes Darstellungsmittel eindeutig das Ergebnis einer Elementarmethode. Es reicht aus, den Produkttypen Darstellungsmittel zuzuordnen.

Die Produktflüsse des V-Modells halten dann fest, in welchen Aktivitäten das Produkt entsteht, und damit ist auch die Verwendung der entsprechenden Elementarmethoden vorbestimmt. Wünschenswert ist es, dass man nur Darstellungsmittel benutzt, die sich gegenseitig ergänzen, d. h., man bringt den UML-Diagrammen die Durchgängigkeit bei.

Dadurch ergeben sich weniger Reibungsverluste beim Erstellen neuer Produkte (Ergebnisse), die auf schon vorhandenen Ergebnissen aufbauen oder sie verfeinern.

Wie funktioniert nun das V-Modell? Der Projektleiter ordnet den Projektmitarbeitern auf der Grundlage des Modells Aktivitäten in einer Art Matrix zu. Indirekt ist diese zentrale Matrix auch eine Übersicht des Lieferumfanges und gehört ins Prozesshandbuch. Auf der CD-ROM befindet sich ein Template, genannt V-Matrix, welches als Ausgangslage für jedes V-Modell-Projekt dienen soll.

Die V-Matrix ist eine Art Zuordnungstabelle, die den Einsatz von Techniken und Tools in der entsprechenden zeitlichen Phase aufzeigt und gleichzeitig das Projekt initialisiert [v]. Konkret lässt sich in jedem Projekt eine Kopie der Matrix aus einem File-Server erstellen, mitsamt den Vorlagen, Checklisten und Templates.

In einem zweiten Schritt passt man die daraus benötigten Aktivitäten an oder streicht sie einfach. Dritter und letzter Schritt ist die Serialisierung der Aktivitäten, d. h., man muss eine vernünftige Reihenfolge auf der Zeitachse bestimmen. Diese Matrix ist dann im Prozesshandbuch für verbindlich zu erklären. Folgende Liste einer Phaseneinteilung der Diagramme in der Domäne SE sind nur exemplarisch als linear anzusehen, da ja in „real life“ eine iterative Spirale vorliegt:

Phase	**Modell**	**Instanzen**
Analyse	Use Case	Fach-Szenario
Analyse	Activity	Business-Units
Analyse/Design	Class Diagram	Objekte
Design	State Event	Zustandswerte
Implementierung	Sequence	Objektszenarios
Implementierung	Package	Versionen
Integration	Component	Binaries
Integration	Deployment	Devices

Tab. 1.2: Phasen im Modell mit den dazu gehörigen Instanzen

Mit in-Step als konkretem Tool (wird später noch kurz vorgestellt) ist Folgendes möglich: Es macht alle Aktivitäten eines Projekts mit ihren Ein- und Ausgabeprodukten und ihrer Mitarbeiterzuordnung sichtbar. Damit bietet es dem Einzelnen eine Art To-Do-Liste und dem gesamten Projektteam Orientierung. Jeder sieht auf einen Blick, welche Aktivitäten geplant, bereit, in Bearbeitung oder durchgeführt sind.

Diese Information wird über eine einprägsame Ampelsymbolik vermittelt. Gleichzeitig erkennt der Bearbeiter, welche Eingangsprodukte er für eine Aktivität benötigt und welche davon in welchem Zustand vorliegen. Ein Blick auf die Ausgangsprodukte zeigt,

welche Produkte man als Ergebnis einer Aktivität erwartet und ob sie bereits vollständig und im richtigen Zustand vorhanden sind.

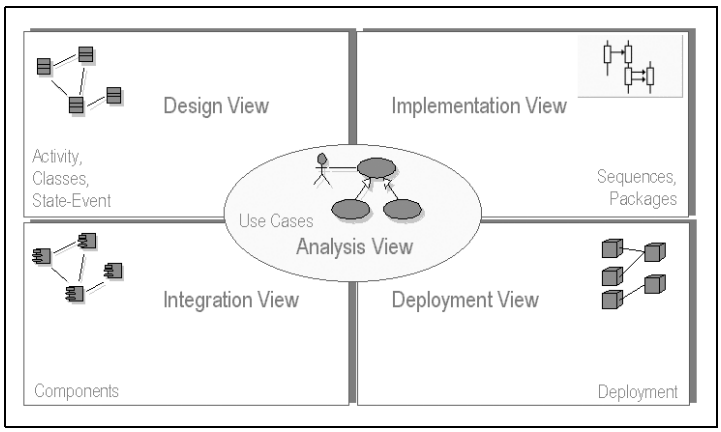

Abb. 1.4: Auch in V wie in RUP gibt es Phasen wie Sichten

1.1.4 XP

Seit etwas mehr als einem Jahr sorgt ein neues Schlagwort für Aufsehen unter den Entwicklern: Extreme Programming (XP) [vi]. XP ist in mancherlei Hinsicht extrem. Demgegenüber steht, wie erwähnt, der schwergewichtige Prozess der RUP. Mit dem RUP versucht Rational auch durch Marketinganstrengungen das dazugehörige Produkt zu verkaufen. Während viele Entwickler in flachen Hierarchien mit der schlanken und entwicklungsorientierten Vorgehensweise von XP sympathisieren, muss eine schwerfällige Organisation doch auch einen schweren Prozess haben, könnte man meinen!

XP kombiniert bekannte UML-Techniken zu einem neuartigen Prozess, der auch denen dienen soll, die bis anhin kein Vorgehen hatten. Die Entwicklung eines Lohnabrechnungssystems bei Daimler-Chrysler diente als Pilot. Spätestens seit Kent Becks Buch und seiner Keynote auf der OOP ‘99 in München ist das Thema auch in Deutschland bekannt.

Einen hohen Stellenwert bei XP hat die direkte Kommunikation zwischen den Mitgliedern, sowohl zwischen Entwicklern als auch zwischen Entwicklungsteam, Kunden und Anwendern. Dies wird durch kleine eingespielte Teams, vor allem paarweises Programmieren und der direkten Mitwirkung des späteren Benutzers gefördert.

> *Eingespielte Kommunikation soll bei XP die meisten Dokumente überflüssig machen, sodass XP Wert auf ein schlankes System legt.*

Nach einigen Recherchen habe ich in einem State Event (Abb. 1.6) die wichtigsten Schritte von XP modelliert. Die Applikation ist stufenweise in kurzen Iterationen erweiterbar. Vor jedem Zyklus stehen die Anforderungen des Kunden, aufgrund derer die Entwickler den weiteren Aufwand schätzen.

Der Kunde wählt dann die tatsächlich zu realisierenden Aufgaben aus. Das Ergebnis jeder Iteration ist eine lauffähige Software. Wichtig ist nun Folgendes: Das Entwicklerpaar schreibt vor dem Bau einer neuen Funktionalität einen Test, welcher das gewünschte Verhalten festlegt.

Mithilfe von Techniken wie DUnit[vii] kann man auf Knopfdruck jederzeit kontrollieren, ob der Test zum Erfolg führt. Zudem ist das Refactoring, d. h. die Umstrukturierung der Unit, ohne ihre Funktionalität zu verändern, ein wesentlicher Bestandteil von XP.

Durch meist kleine Schritte, wie Felder kapseln, Parameter extrahieren oder Vererbung durch Delegation ersetzen, wird der Code kontinuierlich klarer und verständlicher, d. h., der Code dokumentiert sich fast von selbst. Hier wissen wir alle, dass nur Superprofis einen Blick für solch selbsterklärenden Code haben.

Ist eine Klasse mit eigenen Zustandsvariablen bestückt oder will man die möglichen Ereignisse mit ihren Übergängen genauer festhalten, dann sind zusätzliche State Events oder Sequenzdiagramme immer erforderlich.

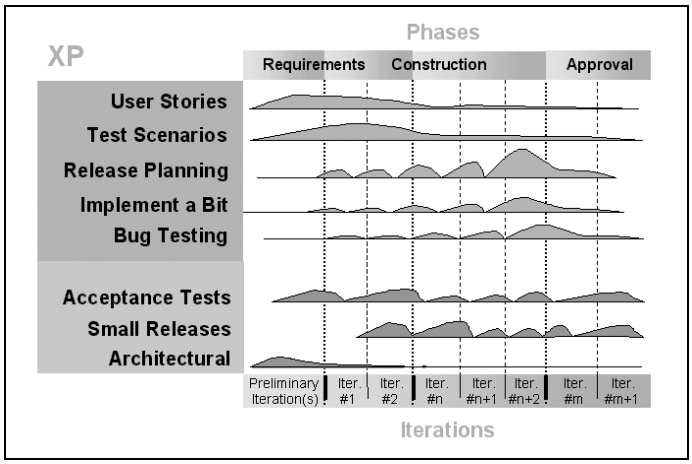

Abb. 1.5: XP mit schlanken Iterationen zum schnellen Testen

Unter bestimmten Bedingungen scheint XP das ideale Vorgehensmodell zu sein, weil es sowohl Entwicklern als auch Kunden und Benutzern entgegenkommt. Diese Bedingungen sind:

- Kunde verfügbar und vor Ort
- Mittlere Projektgröße (2 – 8 Entwickler)
- Kommunikative Mentalität

Erste positive Erfahrungen haben wir in einem Fall mit folgendem Vorgehen gemacht: Die XP-Release-Planung kann man durch Use Cases ersetzen, der Kunde priorisiert die Use Cases und die Iterationsplanung findet dann anhand der detaillierten Use Cases mit Aktivitätsdiagrammen statt. Natürlich nimmt die Gefahr zu, dass man spekulativ entwickelt, weil der Kunde nicht immer für offene Fragen zur Verfügung steht und das rudimentäre Design keine Antwort liefert.

„Trotzdem bleiben die Türme von Hanoi stehen“, meint Franz Zucol. Jedoch favorisieren die XP-Jünger den Einsatz von Design Patterns stark, da ein Testen und anschließendes Durchforsten des Codes die Verständlichkeit erheblich erhöht. Erst die Erfahrung der kommenden Jahre wird zeigen, wie weit XP sich bestätigt, ohne seinen eigentlichen Vorteil, leichtgewichtig zu sein, wieder einzubüßen, da ja gleichzeitig qualitativ hochwertige und kundenorientierte Software entstehen soll, die meistens sorgfältiges Design erfordert.

1.1.5 XP-Techniken

Die folgenden Techniken lassen sich auch in anderen Prozessen Gewinn bringend einsetzen, wie z. B. das Refactoring[viii] gerade eine Renaissance erlebt. Dem Refactoring ist ein eigenes Kapitel gewidmet. Im Refactoring lässt sich das Design fortlaufend in kleinen, funktionserhaltenden Schritten verbessern.

Finden zwei Entwickler Codeteile, die schwer verständlich sind oder unnötig kompliziert erscheinen, verbessern und vereinfachen sie den Code. Sie tun dies in disziplinierter Weise und führen nach jedem Schritt die Unit Tests [ix] aus, um keine bestehende Funktion zu zerstören.

> *Gewöhnlich wird jeder Funktionsblock durch einen Testfall untermauert. Die Unit Tests werden gesammelt, gepflegt und nach jedem Kompilieren ausgeführt. Jeder Testfall wird auf die einfachste denkbare Weise erfüllt. Es wird keine unnötig komplexe Funktionalität programmiert, die momentan nicht gefordert ist.*

Die Programmierer arbeiten stets zu zweit am Code [x] und diskutieren während der Entwicklung intensiv über Entwurfsalternativen. Sie rotieren in der Regel stündlich ihre Arbeit. Das Ergebnis ist eine höhere Codequalität, verbesserte Produktivität und erhöhte Wissensverbreitung. Der gesamte Code gehört dem Team.

Jedes Paar soll jede Möglichkeit zur Codeverbesserung jederzeit wahrnehmen und einbringen. Eine fortlaufende Integration nach mehrmals täglichem Build-Prozess stellt durch eine Versionsverwaltung sicher, dass man die Termine einhält. Damit erreicht man, die als „Timeboxing“ bekannte Unveränderlichkeit des Termins zu umgehen, da am Ende einer Iteration bereits die Detailplanung der nächsten Iteration steht.

1.1.6 Vergleich XP – RUP

Ein Vergleich birgt immer Zündstoff, zumal hinter RUP eine mächtige Industrie steht, die dem kostenlosen Kontrahenten XP Paroli bieten will. Im Gegensatz zu RUP wird bei XP eine vertikale Gruppeneinteilung vorgeschlagen, da in XP eine Spezialisierung unerwünscht ist; jedes Teammitglieder sollte alle Teile des Systems kennen. Wechselnde Programmierer betonen diesen Aspekt. Eine Rollenverteilung in RUP bevorzugt Spezialisierung (Analytiker, Designer, Codierer, Tester etc.).

Frappant ist der Unterschied in der Designstrategie. XP verwirklicht iteratives Design, das einfach beginnt und auf der Technik des Refactoring basiert. RUP startet mit einem möglichst vollständigen Design. Modell und Implementierung sind dann laufend synchronisiert (Roundtrip).

XP stellt Kommunikation über Dokumentation. Zusätzliche Reviews dienen dazu, fehlende Dokumentation durch Audits zu kompensieren. Das Management beschränkt sich in einem XP-Projekt auf wenige Aktivitäten. Diese umfassen hauptsächlich Kontrollen der Zeitschätzungen der Teammitglieder und das Anfertigen von Statistiken als Spiegel des Projektfortschritts. Im RUP kommt dagegen Detailplanung und Plankontrolle zum Einsatz.

In all diesen Eigenschaften zeigen sich die unterschiedlichen Schwerpunkte von RUP und XP: Im Zentrum von XP steht das Bestreben, die zentrale Tätigkeit des Softwarebaus im Team aufzuwerten und die Entwickler mit Tools und Techniken auszustatten, die diese gekonnt einsetzen können.

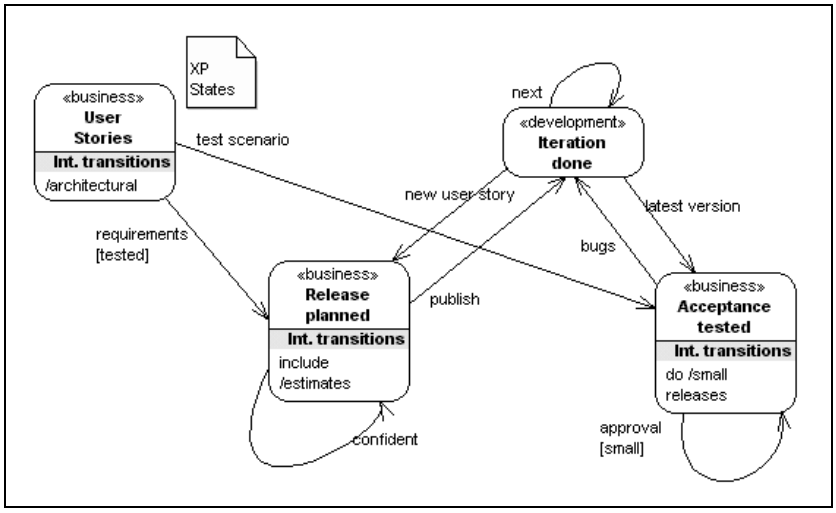

Abb. 1.6: Dynamischer Ablauf des XP-Prozessmodells

1.1.7 Generatoren

Jeder Prozess sollte mit einem Tool fähig sein, aus Diagrammen oder Patterns Code zu generieren. Ich erachte die Codegenerierung, nebst den kommunikativen Fähigkeiten, als künftig wichtigsten Teil im Entwicklungsprozess.

Wie weit die Möglichkeit heute gediehen ist, Prozesse zu automatisieren oder Code aus Patterns zu generieren, soll das Kapitel abschließend in einigen Beispielen aufzeigen.

Auf Basis der Scripting-Technik hat z. B. die ICG eine Anbindung von objectiF an Delphi realisiert. Somit steht dem OO Entwickler ein komplettes Round Trip Engineering zur Verfügung. Einerseits lassen sich neue Projekte in objectiF starten – das Tool generiert die Sourcen in OO – andererseits kann man bereits bestehende Delphi-Projekte per Reverse-Engineering in objectiF einlesen und dann modellbasiert weiterführen.

Wenn Sie ein Element löschen oder umbenennen, sorgt objectiF dafür, dass diese Änderungen an allen relevanten Stellen nachvollzogen werden. Das Ergebnis der Arbeit ist kompilierfähiger Code und lässt sich in der Datenbank von objectiF speichern. Ein mini-

maler Code-Editor ist mit von der Partie. Schon während der Eingabe nutzt der Code-Editor sein Syntax- und Kontextwissen und prüft die Eingabe auf Fehler.

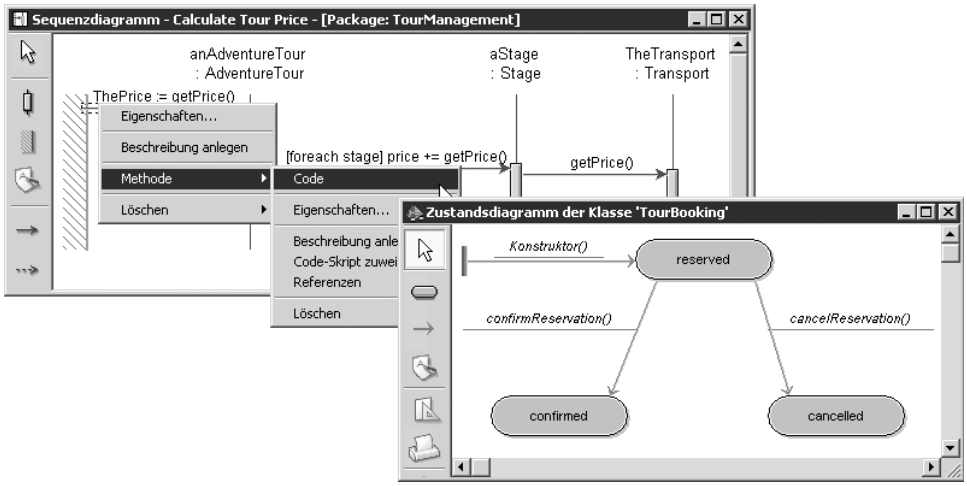

Abb. 1.7: objectiF mit Delphi-Anbindung

in-Step und actiF

actiF beschreibt den objektorientierten Entwicklungsprozess. Es definiert alle notwendigen Aktivitäts- und zu erstellenden Produkttypen, den Produktefluss zwischen den Aktivitätstypen und die Rollen aller Projektbeteiligten. Das vom V-Modell abgeleitete Prozessmodell ist durch Tayloring an eine spezifische Projektsituation anpassbar. Wie stehen nun UML und actiF zueinander?

Antworten auf diese Frage liefert actiF gleich selbst: Es beschreibt, welche Aktivitäten bei der objektorientierten Entwicklung wann durchlaufen werden, welche Produkte dabei entstehen und welche Mittel der UML dazu verwendet werden. Für alle Aktivitäten ist in actiF Tool-Unterstützung vorgesehen.

Es bietet für alle definierten Produkttypen auch Muster an (z. B. ein Word-Template), um eine Standardisierung nicht nur des Vorgehens, sondern auch der Ergebnisse zu erreichen. Wählt ein Projektmitarbeiter unter der Benutzeroberfläche ein ihm zugeordnetes Produkt einfach per Maus an, so wird es automatisch in der korrekten Version und mit dem richtigen Tool geöffnet, z. B. StarOffice oder MSProject.

Die Bandbreite der Werkzeuge kann von einer Tabellenkalkulation bis zum Compiler reichen. Da IT-Organisationen in der Regel über zahlreiche Tools verfügen, die bereits auf den spezifischen Bedarf abgestimmt sind, erlaubt ein Adapter, beliebige Tools unter einer gemeinsamen Oberfläche zu einer homogenen und durchgängigen Entwicklungsumgebung zu integrieren.

„Na, wie weit sind Sie denn? Geht es voran?" Diese Fragen rauben den Schlaf. Ob Auftraggeber, IT-Leiter, Geschäftsführung oder Entwickler, irgendwer stellt früher oder später in einem IT-Projekt diese Fragen.

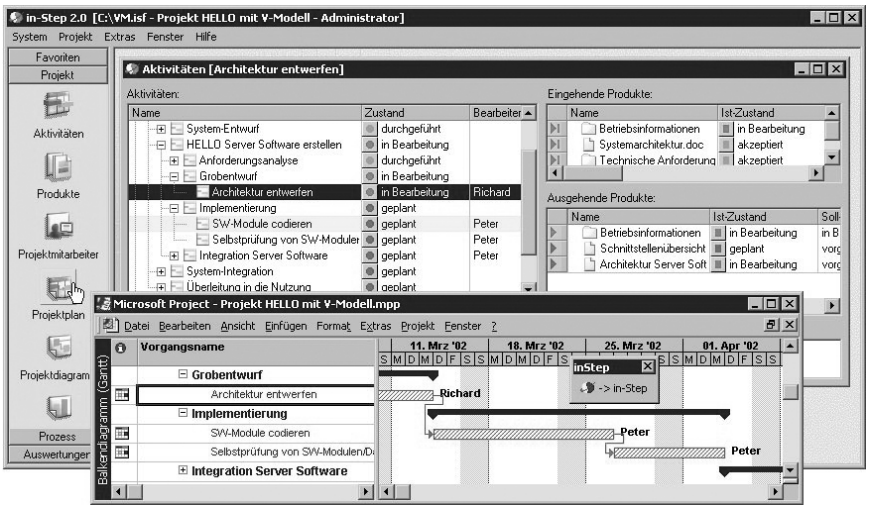

Abb. 1.8: Der Prozess bei objectiF

Nehmen Sie einmal an, Sie könnten so antworten: „Wir kommen zügig voran. Schauen Sie selbst." Und tatsächlich: Auf dem Bildschirm sind alle Aktivitäten und Produkte Ihres Projekts zu sehen (siehe Abb. 1.8). Farbige Markierungen zeigen ihre aktuellen Zustände. Sie erkennen sofort, wo das Projekt steht. Wenn Sie etwas länger hinschauen, erleben Sie den Projektfortschritt live: Sie sehen, wie die Markierungen der Aktivitäten von Rot/geplant über Gelb/bereit und Grün/in Bearbeitung auf Hellgrau/durchgeführt wechseln. Eine tolle Vision?

Es macht den Projektfortschritt in Echtzeit sichtbar. Es schafft Orientierung und hilft bei der Kommunikation – gerade in großen und verteilten Teams. Es unterstützt Sie bei der Aufgaben-, Termin- und Kostenplanung. in-Step steuert den Workflow im Projekt und sorgt auch dafür, mit einem integrierten Projekt- und Konfigurationsmanagement die Basis eines unternehmenseigenen Prozessmodells zu besitzen.

Patterns in XDE

Ein weiteres Tool ist XDE von Rational. XDE unterstützt nebst Codegenerierung auch Design Patterns mit einer umfangreichen Bibliothek. Die Muster eignen sich perfekt z. B. auch zu Schulungszwecken. Durch eine Pattern Engine hat man die Möglichkeit, neue Projekte mit einem automatischen Codegerüst zu versehen. Es ist auch möglich, Kombinationen von Patterns zu erzeugen, da z. B. ein Factory meist mit einem Singleton daherkommt. Erste Vorstufen einer MDA sind ersichtlich. Ein Mapping der Felder zu Attributen lässt sich direkt vornehmen.

Die Design Patterns sind von reicher Qualität und in abstrakter Notation vorhanden. Als Beispiel sei der Builder genannt, der ein Trennen der Konstruktion eines Objektes von seiner Repräsentation verlangt, sodass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann. Ein mächtiges Muster, welches nicht nur die visuellen Eigenschaften einer Konstruktion optimiert.

Man kann sich z. B. einen RTF-Leser vorstellen, der in Aggregation mit einem Konvertierer (als Builder) steht und RTF-Files interpretieren kann. Der Konvertierer selbst ist dann fähig, unterschiedliche Repräsentationen zu erzeugen, wie ASCII, Unicode, XML oder eine LaTeX-Datei für die Puristen unter uns.

Die meisten Entwicklungstools sind ja erstellt worden, um Code zu erzeugen, und nicht, um Modelle zu implementieren. Design Patterns lassen sich hier als eine Art Vermittler zwischen Modell und Code einsetzen, da man den Fokus auf die Architektur legt.

Den Schritt, ein vollständiges Tool mit Modellierung und Implementierung anzubieten, macht Rational wohl erst in der nächsten Version. Auch ein so genanntes Data Modeling ist mit von der Partie, welches speziell für die Modellierung von SQL-Datenbanken gedacht ist.

Modellgetrieben in StP

Software through Pictures nennt es „Architecture-Component-Development“(ACD)-Technologie. Auch unter dem Aspekt, dass die OMG dieses Thema unter dem eigenen Namen MDA weiterentwickelt, sind die Ideen hinter ACD und MDA identisch, mit dem Unterschied, dass sich Aonix diesem Thema schon seit drei Jahren widmet.

Einzigartig ist Aonix ACD-Technologie. ACD schlägt eine Brücke zwischen UML und realem Code, sodass mit einer Transformationsregel, welche wiederum mithilfe von Templates automatisch fast 70 % Code aus den Modellen generieren soll. Dahinter steckt eine spezielle Templatesprache, die man dem Transformator füttert.

StP/ACD ermöglicht die Trennung der fachlichen Anforderungen von den technischen Details der Implementierung. Die fachlichen Aspekte sind mit StP/UML-Modellen und die technischen Aspekte in Templates für den ACD-Transformator modellierbar. Dadurch erhält man einen sehr hohen Abstraktionsgrad in den UML-Modellen und ein hohes Maß an Wiederverwendung durch die technischen Musterlösungen in den Templates. Die Aktivitäten lassen sich durch diese Trennung wesentlich gezielter einsetzen. Mittels der ACD-Templatesprache, welche eine Art Abbildungsregeln definiert, soll der generierte Teil des Codes mehr als die Hälfte betragen.

Die Architektur von StP zeichnet sich durch ein Multiuser-Repository aus, das auch bei großen Projekten für entsprechende Skalierung sorgt. Durch Konfigurationsdateien lässt sich StP einfach an eigene Entwicklungsprozesse anpassen.

Die Integration mit weiteren Produkten wie Config-Management-System, IDEs für verschiedene Programmiersprachen und Testwerkzeuge sorgen für die Abdeckung des gesamten Softwarezyklus.

Spätestens beim Komponenteneditor bemerkt man die durchdachten Checks von Syntax und Semantik bezüglich der einzelnen Diagramme. Die weiteren Leistungsmerkmale von StP sind wie folgt:

- Speichert das semantische Modell in einer offenen Datenbank nach Wahl
- Verbesserte Konsistenzprüfung auf allen Ebenen und Sichten
- Integriert mit Tools automatische Dokumentengenerierung mit definierten Regeln
- Abbildungsregeln, die ein Modell in mehrere Klassen transformieren
- Eigene Templatesprache mit Transformator

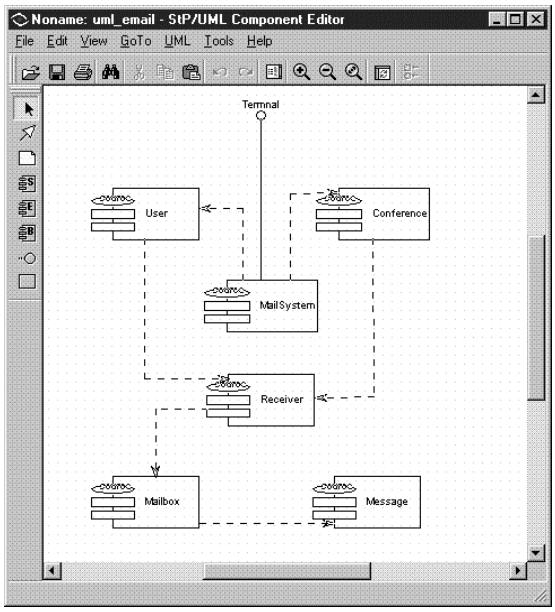

Abb. 1.9: Der Komponenteneditor von StP

Abschließend gilt noch zu bemerken, dass das früher bekannte Tool Select Enterprise in der Zwischenzeit an Aonix verkauft wurde. Alle Softwareprodukte aus diesem Bereich sowie die Mitarbeiter hat Aonix übernommen. Eine Zusammenlegung von StP und Select ist demnach nicht auszuschließen.

Auch Borland hat nach der Übernahme von Bold und Togethersoft mit dem Together Control Center Großes in der Pipeline. Es soll, gemäß Jason Vokes, eine gemeinsame IDE (C#Builder und Octane/Delphi) für verschiedene Programmiersprachen und Testwerkzeuge geben, welche die Abdeckung des gesamten Softwarezyklus inklusive UML und MDA beinhaltet.

1.2 Zeiger und Schnittstellen

Was ist eigentlich mit Pattern-Techniken gemeint? Viele Patterns benutzen gewisse Techniken immer wieder, die einen gemeinsamen Nenner haben, sie beruhen z. B. auf Schnittstellen und Listen. So nebenbei ist natürlich die Polymorphie oder eine darauf aufbauende Kollektion auch mit von der Partie. Daher die Idee, diese Techniken in den nächsten Kapiteln grundlegend zu durchleuchten. Grundlegend gibt es auch die drei Hauptsätze der OO-Technologie, welche beim Einsatz von Patterns zum Tragen kommen.

Die drei Hauptsätze der OO-Technologie:

1. Entwickle auf eine Schnittstelle hin, nicht auf die Implementierung.
2. Ziehe Objektkomposition der Klassenvererbung vor.
3. Setze Assoziation zur Laufzeit und Aggregation zur Entwicklungszeit ein.

Hauptsatz 1 und 2 sind Gegenstand dieses Kapitels, da beide auf Interfaces beruhen und im Hauptsatz 2 die Idee der Delegation mit Zeigern oder Referenzen daherkommt. Hauptsatz 3 manifestiert sich bei den Verhaltensmustern in den konkreten Klassendiagrammen, welche sich vor allem in Teil 2 öffnen.

Wo Licht ist, ist auch Schatten. Techniken sind von der zu implementierenden Sprache abhängig, und das Einzige, was Delphi nicht beinhaltet, ist die multiple Vererbung und so genannte Statics, wobei multiple Vererbung in keinem Pattern vorkommt und es einen würdigen Ersatz für Statics gibt (Klassenmethoden). Also doch kein Schatten.

> *Ein Static ist übrigens ein Klassenattribut, das in jedem von der Klasse stammenden Objekt wie eine Konstante als Objekt-Attribut vorkommt.*

1.2.1 Generisch wie ein Zeiger

Eine Zeigertyp-Variable (englisch Pointer) enthält keinen Wert, sondern die Speicheradresse einer dynamischen Variablen eines Basistyps. Die dynamische Variable, auf die eine Zeigervariable zeigt, wird wie im Codebeispiel durch Schreiben des Zeigersymbols `^` vor der Variablen referenziert.

Um auf Werte der dynamischen Variablen zuzugreifen, muss man den Zeiger dereferenzieren, d. h. das Zeigersymbol `^` wie in `ptrAdresse^.strOrt:= ptrString^;` nach der Zeigervariablen anfügen.

Bevor man Zeiger dereferenzieren kann, muss man sie allerdings erst mit der Prozedur `New` erzeugen, wie dies nach dem `begin` geschieht. Nach Gebrauch oder spätestens bei Programmende sollte man den für die dynamische Variable reservierten Speicher mit der Prozedur `Dispose` wieder freigeben.

Typen- und Variablendeklaration, Initialisierung, Zerstörung und Verwendung für Zeigertypen sehen folgendermaßen aus:

```
Type
  TptrAdresse = ^TrecAdresse;
  TrecAdresse = record
    strName : string;
    strOrt : string;
  end;
var
  ptrString : ^string;
  ptrAdresse : TptrAdresse;

begin
  new(ptrString);
  new(ptrAdresse);
  ptrAdresse^.strName:= 'Paul Pattern';
  ptrString^:= 'CODESIGN';
  ptrAdresse^.strOrt:= ptrString^;
```

```
  lblTest.caption:= ptrAdresse^.strName
    + ', ' + ptrAdresse^.strOrt;
  dispose(ptrAdresse);
  dispose(ptrString);
end;
```

Wenn Sie einen Zeiger auf einen eigenen Typ erstellen wollen, dann müssen Sie wie `ptrString: ^string[30];` diesen Typ innerhalb der gleichen Typendeklaration definieren.

Das reservierte Wort NIL bezeichnet eine Zeigerkonstante, die auf nichts zeigt. Da z. B. ein leerer String keine gültigen Daten enthält, entspricht eine Indizierung eines leeren Strings einem Zugriff auf NIL – das Ergebnis ist eine Zugriffsverletzung.

Zeiger gehören durchaus zu den fehleranfälligsten, aber auch zu den mächtigsten Techniken. Der Einsatz von Zeigern ist von Haus aus kompliziert und erfordert solide Kenntnisse. Zeiger sind meistens die einzige Chance, den technischen Horizont massiv zu erweitern (Pointers are the realm of programmers). Begleiten Sie mich auf dem ersten Rundgang im Interface-Kapitel und erleben Sie, wie Delphi mit Zeigern umgeht oder eben die Zeiger mit uns.

Einsatzgebiet

Die Bedeutung der Zeigertypen ist in Object Pascal aus erster Sicht wohl rückläufig, zumal der Compiler die Objekte – und seit Delphi 2 auch Zeichenketten – von sich aus dynamisch und für den Entwickler fast unbemerkt verwaltet (transparent, wie es so schön heißt). Auch die Selbstdeklaration von Zeigern hat abgenommen, denn es stehen mit Objekten neue Sprachelemente zur Verfügung, welche die Zeiger **kapseln**.

In anderen Bereichen, eben aus zweiter Sicht, wie beim Zugriff auf APIs, Verwalten von Instanzen zur Laufzeit, Aufbau von eigenen Datenstrukturen und Klassenhierarchien oder dynamischer Verwaltung von Multimedia-Daten in Listen oder Kollektionen ist der Einsatz von Zeigern immer noch unentbehrlich. Sogar ein Interface als eine typisierte Tabelle von Funktionszeigern werkelt im Hintergrund nur mit Pointern. Auch im Komponentenbau fängt mit Zeigern der Spaß erst an.

> *Wenn Sie eine Struktur oder einen Datentyp in Object Pascal definieren, ist die Empfehlung, immer auch einen Zeiger auf diesen Datentyp zu definieren. Viele höhere Programmiertechniken, wie etwa verkettete Listen dynamisch verwalteter Datensätze, erfordern die Verwendung solcher Zeiger anstatt der Variablen selbst, zudem entfällt das Kopieren größerer Speicherblöcke.*

Zeigerinhalt

Jeder Zeiger enthält zwei Informationen: die Adresse als Zeiger und die Bedeutung des Inhalts der Speicherstelle, worauf der Zeiger zeigt. Die Anweisung, wie die durch die Zeigeradresse gefundenen Bits interpretiert werden sollen, ist im **Basistyp** des Zeigers enthalten. Ein Zeiger auf eine ganze Zahl (pointer to an integer) teilt dem Compiler mit, dass der Inhalt der adressierten Speicherstelle als ganze Zahl interpretiert werden soll.

> *Typen lassen sich generell in einfache, String-, strukturierte, Zeiger-, prozedurale oder variante Typen einteilen. Auch Typenbezeichner selbst gehören zu einem speziellen Typ, da man sie als Parameter an bestimmte Funktionen (z. B. High, Low und SizeOf) übergeben kann.*

Versuchen Sie sich vorzustellen, dass eine Speicherzelle einfach nur Bits, Hi und Lo enthält, die an keine Interpretation gebunden sind. Erst die Decodierung dieser Bit-Muster nach einem definierten Schema gibt den Nullen und Einsen einen Sinn.

Diese Indirektion ist natürlich fehleranfällig. Oft ist es einfach, einen Fehler zu finden, aber schwierig, ihn zu korrigieren. Zeiger sind da anders. Zeigerfehler entstehen meistens dadurch, dass der Zeiger auf eine Adresse zeigt, die eigentlich verboten ist. Nun kann bis zum Schreiben in den falschen Speicherbereich Zeit vergehen, wie eine tickende Zeitbombe im Hintergrund. Das Tückische daran ist, dass die Speicherverwaltung die Schutzverletzung nicht sofort meldet (z. B. wenn zufällig in diesem Speicherbereich nichts Bedeutendes steht), der Fehler selbst ist aber mit tödlicher Präzision immer noch vorhanden. Erinnern Sie sich an den letzten „Demo Effekt“?

Objekterzeugung

Das neue Objektmodell basiert auf dem Typ `TObject` und stellt gegenüber der OWL in Borland Pascal 7.0 eine Vereinfachung dar. Den Typ `object` gibt es aber seit Delphi 7 nicht mehr. Bei hausgemachten Strukturen ist die Geschichte jedoch gleich geblieben (auch unter Pascal gibt es so eine Art Investitionsschutz). Beim alten Modell wurde explizit eine Referenz auf dem Heap angelegt:

```
type
PTheObject = ^TTheObject;
TTheObject = object
  constructor Init;
  destructor Done;
end;
var TheObject: PTheObject; //Zeigerbelegung von 4 Byte
```

Im neuen Modell genügt die Klassenangabe als Referenz:

```
type
TTheObject = class(TObject)
  constructor Init;
  destructor Destroy;
end;
var TheObject: TTheObject; //Zeiger von 4 Byte;
```

Dadurch fällt der Umweg über die Pointer-Deklaration weg und die Initialisierung ist auch einfacher zu handhaben. Auch die Zuweisung von `TheObject` ohne einen Konstruktor ist nicht mehr so ohne weiteres möglich. Somit entsteht mehr Sicherheit. Es ist

aber möglich, mehrere Referenzen auf eine Instanz zu haben. Wo aber steckt nun das Objekt in den unendlichen Weiten des Adressraumes[2]:

Folgende vier Fälle lassen sich genauer untersuchen:

1. Referenzieren mit
   ^TTheObject — Zeigt auf den Basistyp
2. Adressoperator mit
   @TheObject — Adresse der 4Byte Zeigervariablen
3. Pointerinhalt
   pointer(TheObject) — Inhalt der Zeigervariablen
4. Dereferenzieren mit
   TheObject^ — TheObject, eigentliches Objekt mit Inhalt

Alle Klassen (und deshalb auch alle Komponenten) sind im Grunde Zeigertypen. Der Compiler dereferenziert Klassenzeiger automatisch, sodass Sie sich im Normalfall nicht damit befassen müssen. Der Status von Klassen, die als Zeiger fungieren, wird in dem Augenblick bedeutsam, in dem Sie eine Klasse als Parameter übergeben wollen. Generell sollten Sie Klassen **als Wert** und nicht als Referenz übergeben.

Der Grund dafür liegt darin, dass Klassen bereits Zeiger sind. Wenn Sie eine Klasse als Referenz übergeben, übergeben Sie also eigentlich eine Referenz auf eine Referenz.

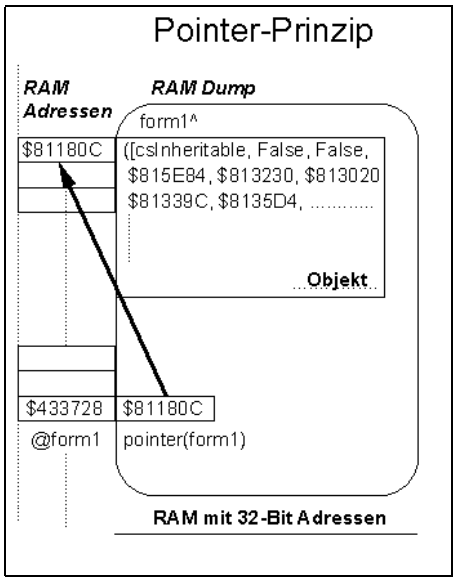

Abb. 1.10: Die Übersicht der vier Fälle verdeutlicht das Pointer-Prinzip

[2] Man spricht von Adressraum und Rechnergrenzen.

Zeiger sind typisiert. Sie geben also auch die Art der Daten an, auf die sie zeigen. Der Allzwecktyp Pointer repräsentiert einen Zeiger auf beliebige Daten, während spezialisierte Zeigertypen nur auf bestimmte deklarierte Datentypen zeigen. Zeiger belegen immer vier Bytes Speicherplatz.

Analysieren und betrachten wir nun die Geschichte der vier Fälle der Reihe nach, anhand eines Form-Objektes beginnen wir mit Fall 1, welches Delphi fast immer im Alleingang erzeugt, nämlich z. B. `Form1` von `TForm1`, das von `TForm` geerbt wurde. So erspare ich Ihnen und Sie mir zusätzliche Tipparbeit, zudem ist die konkrete Konstruktion eines Form-Objektes den meisten schon bekannt.

Beim eigentlichen **Referenzieren** fasst Delphi die Schritte `PForm1=^TForm1` und `TForm1=object` zusammen. Delphi generiert in einem Schritt `TForm1= class(TForm)`. Das eigentliche Objekt instanziert der Compiler mit

```
var  Form1: TForm1;
  Form1:= TForm1.create(self);
```

Es ist wichtig zu wissen, dass jede Definition einer Klassenvariablen vom Typ `class` einen Zeiger auf diese zukünftige Klasseninstanz liefert. Gegenstand unserer weiteren Betrachtung wird nun `Form1` sein. Unter Fall 2 holen wir die Adresse von `Form1` heraus, d. h., wir fragen uns, an welcher Speicherstelle die 4-Byte-Zeigervariable `Form1` liegt. Hier kommt uns der Adressoperator @ zu Hilfe, welcher folgendes Ergebnis liefert:

Abb. 1.11: Auch ein Zeiger hat sein Zuhause

Das Ergebnis $433728 ist aber nicht die Adresse des Objekts, sondern eben die Adresse der Zeigervariablen `Form1` selbst. Zum gleichen Ergebnis kommen Sie auch mit `addr(Form1)`. Jetzt wissen wir, wo die Zeigervariable im RAM liegt, aber kennen noch nicht deren Inhalt, welcher ja die eigentliche Adresse auf das Objekt darstellt (Fall 3). Weil `Form1` einen Zeiger darstellt, müssen wir durch ein Typecasting den Compiler überlisten, die Ausgabe ungeprüft als Pointertyp darzustellen, um so die Adresse unseres Objekts zu erfahren:

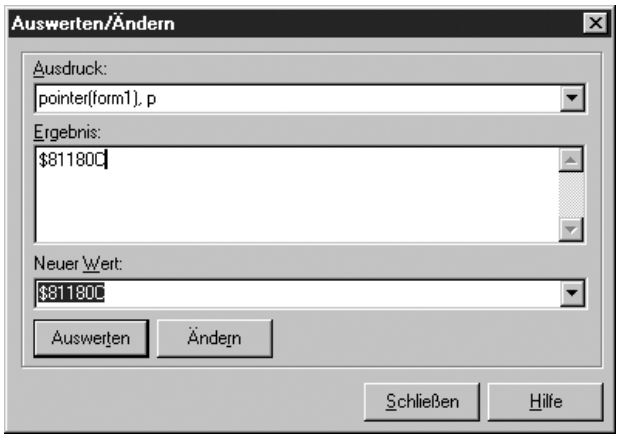

Abb. 1.12: Hier sehen wir, wohin der Zeiger zeigt

Der Formatanweisungsparameter p ist nicht unbedingt nötig, da ein Typecast die Sache ja erzwingt, kann aber sehr nützlich sein, denn er stellt Werte von Zeigern als 32-Bit-Adressen dar, mit Informationen über die Zieladresse, Angaben über den Speicherbereich, in welchem sich der Zeiger befindet, und den Namen der Variablen an der Offset-Adresse.

Hier sollte man ein wenig experimentieren, ja, das Ganze wirkt manchmal wie ein Mikroskop (Debug-Sitzung) mit Blick auf den binären Adressraum und den „daherschwimmenden“ Molekülen (Pointer). Das Atom wäre demnach ein Bit, oder eben bitte ein Bit;).

Mit „@form1, 4m“ erreichen wir dasselbe Ziel. 4m zeigt als Speicherauszug 4 Bytes ab der Adresse des gegebenen Ausdrucks. Im Code sind Zeiger am besten mit der Formatanweisung als String darstellbar:

```
FObj:= Format('%p',[pointer(self)]);
```

Jedes Byte wird standardmäßig als eine Folge zweier hexadezimaler Ziffern dargestellt. Die gefundene Adresse von `Form1` ist somit $81180C.

Das Objekt der Begierde

Bei Fall 4 wollen wir nun dem eigentlichen Objekt auf die Spur kommen. Während ja bei allen anderen Variablen der Compiler Speicher für diverse Daten reserviert, enthalten Zeigervariablen lediglich die Adresse eines solchen Speicherbereichs. Und dieser Speicherbereich, unser Form-Objekt, erreichen wir mit einer Dereferenzierung, die sich nach dem neuen Objektmodell direkt ansprechen lässt:

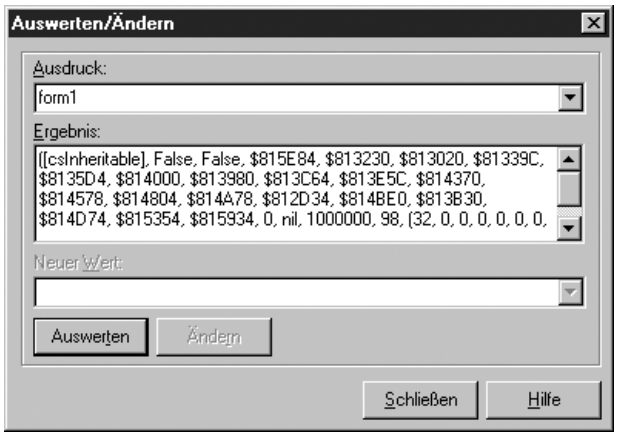

Abb. 1.13: Das Objekt mit seinen Methoden und Eigenschaften

Jede Methode besitzt noch den zusätzlichen Parameter `Self`. Bei diesem handelt es sich um eine Referenz auf die Instanz oder Klasse selbst, für welche man die Methode innerhalb der Klasse aufruft. Der Parameter Self wird immer als 32-Bit-Zeiger übergeben, sofern das Betriebssystem mitmacht. Deshalb sollte (je nach Breakpointstand bei Auswerten) auch mit Self das Objekt im Ergebnis erscheinen.

Die weiteren Zeiger, die wir im Objekt `Form1` sehen, sind übrigens Methodenzeiger (method pointer), die auch Ereignisse sein können. In Borland Pascal auch schon unter dem Begriff Prozedurtypen bekannt (nicht ganz ein method pointer), können Sie diesen speziellen Variablen nicht nur Daten zuweisen, sondern auch Prozeduren und Funktionen. Eigentlich speichert der Code in diesem Fall nur ein Zeiger auf die Prozeduren oder Methoden. Ein schneller Blick in die Delphi-Hilfe bestätigt:

> *Ein prozeduraler Typ, der mit der Klausel of object deklariert ist, wird als Methodenzeiger bezeichnet. Ein Methodenzeiger kann eine Prozedur oder Funktion eines Objekts referenzieren. Er wird als zwei Zeiger codiert, von denen der erste die Adresse der Methode speichert und der zweite eine Referenz auf das Objekt, zu dem die Methode gehört.*

Ein Beispiel dazu, welches im Mediator Pattern noch vorkommt:

```
Type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Typecasting

In Fall 3 mussten wir den Compiler mit einem Typecasting überlisten, welches ihn zwingt, etwas ungeprüft (ohne Typenprüfung) auszuwerten. Der Compiler geht also davon aus, dass wir die Typenprüfung übernehmen. Sollten Sie sich jedoch irren, dann folgt

die Überraschung. Bei einem Typecasting mit einem Pointer ist das nicht Kühnheit, sondern es sind schlicht pragmatische Gründe.

Die Technik des Typecastings lässt sich auch als explizite Typumwandlung einsetzen, da sich z. B. nach der Umwandlung ein typisierter Zeiger dereferenzieren lässt. Variablen des **untypisierten Typs** Pointer lassen sich nicht dereferenzieren, es ist somit ein Fehler, einer solchen Variable das Zeigersymbol ^ nachzustellen.

Über das Dereferenzierungssymbol ^ zeigen Sie Delphi, dass Sie nicht am Zeiger interessiert sind, sondern nur an den Daten, auf die der Zeiger zeigt. Bei einer Typumwandlung wird eine Variablenreferenz eines bestimmten Typs in eine Variablenreferenz eines anderen Typs umgewandelt. Für die Gültigkeit der benötigten Umwandlung ist auch hier der Entwickler verantwortlich:

```
var
  roherZeiger: Pointer;
  typisierterZeiger: PByte;
begin
  roherZeiger:= typisierterZeiger; //ohne Typumwandlung
  //ein gut erzogener Zeiger
  typisierterZeiger:= PByte(roherZeiger);
```

Der NIL-Zeiger

Zeiger, die man noch nicht durch eine Zuweisung initialisiert hat, zeigen ins Nirwana.[3] Daher sollte man sich angewöhnen, für nicht initialisierte Zeiger den vordefinierten Wert NIL zu verwenden. Intern belegt der Compiler NIL durch vier Null-Bytes. Das Schreiben von Werten an eine NIL-Adresse führt zwar immer noch zu Fehlern, jedoch kann man **vor** der Verwendung des Zeigers prüfen, ob dieser auf NIL zeigt. Wenn nein, lässt sich bei defensiver Codierung davon ausgehen, dass der Zeiger korrekt initialisiert ist. Defensiv bedeutet unter anderem, dass man den Zeiger nach der Entwertung auf NIL setzen sollte:

```
destructor TKreditKonto.destroy;
  begin
    dataList.Free;
    Transakt_Obj.Free;
    ctr.kreditKonto:= NIL;
  end;

if buffer <> NIL then begin  //if assigned(buffer)
    freeMem(buffer);
    buffer:= NIL;
end;
```

[3] Nirwana verkörpert die morbide Ästhetik der Informatik.

NIL bedeutet aber nicht, dass die Zuweisung von NIL den Speicher des Objektes gleich freigibt. Die Zuweisung von NIL ist nichts anderes, als den Pointer auf die Adresse 32bit (0000) zeigen zu lassen. Auch nach der Freigabe des Speichers ist kein automatisches Setzen von NIL zu erwarten, deshalb das explizite Setzen von NIL.

Abschließend noch einige Beispiele für Zeiger im Einsatz.

Bei COM oder DCOM wird ja ein Aufruf via Netzwerk auf den entfernten Rechner geleitet und dort das Server-Objekt mit einer CLSID_X in der Registry lokalisiert. Nach dem Start gibt das entfernte Server-Objekt einen Zeiger auf sein Interface mittels der COM-Bibliothek an den Client zurück. Der Client ist nun in der Lage, das Objekt mittels des Zeigers direkt aufzurufen.

Beim Aufruf eines OLE-Objektes erzeugt ein Client meistens mittels des Typs `Variant` eine Instanz. Varianten lassen sich bevorzugt dann einsetzen, wenn der eigentliche Datentyp, der verarbeitet werden muss, entweder wechselt oder zum Zeitpunkt der Compilierung noch unbekannt ist. Im Grunde genommen handelt es sich hier um einen typisierten Zeiger auf das Objekt.

Im Folgenden finden Sie einen Auszug als Beispiel, in dem ein Client eine OLE-Automatisierungsmethode aufruft. Beachten Sie, wie die Anweisung mit der Funktion `CreateOleObject` aus der Unit `OleAuto` eine Variante erzeugt, die eine Referenz auf ein OLE-Automatisierungsobjekt enthält, mit der wir dann Methoden wie `FileNew` aufrufen können:

```
var Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');  //Referenz zurück
  Word.FileNew('Normal');
  Word.Insert('Dies ist die erste Zeile'#13);
  Word.FileSaveAs('g:\delphi7\temp\ole\maxtest.txt', 3);
end;
```

Wie schon das Handling bei den COM-Schnittstellen gezeigt hat, sind auch die IDAPI-Funktionen (ja, es gibt sie immer noch) in hohem Maße von Zeigern geprägt. Angenommen uns interessiert eine Datensatznummernanzeige von Paradox-Records. Beim Durchstöbern der IDAPI-Funktionen sind wir auf die

```
function DbiGetSeqNo (hCursor: hDBICur; var iSeqNo: Longint):
DBIResult stdcall;
```

gestoßen. Beim formalen Parameter `iSeqNo` erfahren wir, dass es sich um einen Pointer handelt, gut getarnt als `Longint`. Diese Tarnung, den `Longint` auch als Pointer zu gebrauchen, ist nahe liegend und gebräuchlich, da ein Longint 4 Byte verbraucht. `DbiGetSeqNo` erwartet also zwei Parameter, das Cursor-Handle (Cursor meint übrigens „Current Set of Records“ und hat nichts mit dem sonstigen Cursor am Hut) und als zweiten Parameter unseren Zeiger auf eine 32-Bit-Variable, in der die aktuelle Datensatznummer zurückkommt. Die aktuelle Datensatznummer wird tatsächlich in der 32-Bit-Variablen `RecID` abgelegt:

```
procedure GetRecordID(ATable: TTable; var RecID: Longint);
begin
  with ATable do begin
    UpdateCursorPos;   // sync BDE with Delphi
    Check(DbiGetSeqNo(Handle, RecID));
     else
       raise EDatabaseError.Create('Not a IDAPI table');
    CursorPosChanged;
  end;
end;

//Aufruf z. B. mit
GetRecordID(TableKonto, RecID);
AktRecID:= RecID
Panel1.Caption:= Format('Datensatz %d von %d',[AktRecID,
                                  TableBiblio.Recordcount])
```

Am Ende der Zeigerei stoßen wir noch kurz in die tiefen Weiten der Win-API-Library vor und betrachten das explizite Laden einer DLL. Dieses explizite Laden hat den großen Vorteil, dass eine Applikation die DLL nur bei Bedarf holt und nicht „Alle von Anfang an“. Jedoch muss man einen prozeduralen Typ deklarieren und der Adressoperator @ bekommt eine neue Bedeutung:

Wird der vielseitige Adressoperator @ auf eine prozedurale Variable oder einen Prozedur- oder Funktionsbezeichner angewendet, verhindert die Laufzeitumgebung den Aufruf der Prozedur und wandelt gleichzeitig das Argument in einen Zeiger um.

Die von Windows definierte Funktion `GetProcAddress` (in der Unit `WinProcs`) gibt beispielsweise die Adresse einer in eine DLL exportierten Funktion als untypisierten Zeigerwert zurück. Mit dem Operator @ können Sie dann das erhaltene Ergebnis eines `GetProcAddress`-Aufrufs einer prozeduralen Variablen zuordnen:

```
Type
  TStrComp = function(Str1, Str2: Pchar): Integer;
var
  StrComp: TStrComp;
  hDLL: THandle
begin
 ...
  @StrComp:= GetProcAddress(hDLL, 'getStrComp');
end.
```

> *Wenn der Adressoperator @ sozusagen in einer Prozedurvariablen zweckentfremdet wird, wie lässt sich dann sein Ort ausfindig machen? Mit dem doppelten Adressoperator @@ können Sie anstelle der in einer prozeduralen Variable gespeicherten Adresse die physikalische Speicheradresse der Variable selbst ermitteln! Es wurde eben an fast alles gedacht.*

Mit diesen drei Beispielen sollte noch einmal die Bedeutung und Vielfalt der Zeigerei unterstrichen werden. Ohne zusätzliche Tools wie Bounds Checker, MemProof oder HeapWalker kann Windows mit den Zeigern des Todes schnell zum Alptraum degenerieren. Als „Zeigertraining“ empfehle ich öfter den Studenten, erst mal eine gehörige Portion Minesweeper zu spielen, da man auch hier mit der Zeigerei am weitesten kommt. Und weiter geht's innerhalb der Pattern-Techniken mit Properties.

1.2.2 Properties

Die Elementliste eines Interfaces (später mehr) darf aus Designgründen nur Methoden und Properties enthalten. Felder sind in Schnittstellen nicht erlaubt. Da für Schnittstellen keine Felder verfügbar sind, können die Zugriffsattribute für Properties (read und write) eigene Methoden sein. Doch ein Property kann auch direkt ohne getter und setter auf ein Feld referenzieren ist deshalb von universaler Bedeutung, weshalb ein vertieften Blick auf die Properties sinnvoll erscheint.

Wenn wir jetzt künftig von **Properties** sprechen, sind eigentlich auch **Eigenschaften** gemeint. Wobei Eigenschaften auch etwas mit Feldern gemeinsam haben. Definieren wir: Eigenschaften sind ein genereller Begriff für Felder oder Properties. Doch Properties sind eine der besonders wichtigen OO-Techniken von Delphi und verkörpern sozusagen das OOP-Paradigma, welches besagt:

> *Ein Property einer Instanz sollte man nur über die dazugehörige Zugriffsmethode manipulieren können.*

Und genau dazu dienen die Properties. Denn die Daten sind in den Properties mit definierten Methoden verbunden, die diese Daten verändern können. Jedesmal, wenn Sie einem Property einen Wert zuweisen oder das Property nur lesen (z. B. den Wert einer anderen Variablen zuweisen) **feuert** Delphi automatisch die entsprechende Methode, sodass diese auf jede Änderung reagieren kann. Somit sind Properties also nur vereinfachte Methodenaufrufe.

Properties vermitteln dem Anwender die Illusion, dass er den Wert einer Variablen in der Komponente schreibt oder liest. Dem Entwickler hingegen ermöglichen sie, die zugrunde liegende Datenstruktur zu verbergen und globale Seiteneffekte (Side Effects) des Zugriffs zu vermeiden.

Der Einsatz von Properties bietet dem Entwickler mehrere Vorteile:

- Properties können Werte oder Formate prüfen, während der Anwender diese eingibt und zuweist. Die Validierung der Anwendereingaben beugt Fehlern vor, die man durch ungültige Werte verursachen kann.
- Das Programm kann bei Bedarf geeignete Werte kontrolliert erzeugen.
- Wechselwirkungen lassen sich ausschließen, da ein Property wie ein Schalter wirkt.

Der vielleicht häufigste Fehlertyp ist das Referenzieren einer Variablen, welche keinen Anfangswert besitzt. Indem Sie den Wert zu einem Property erklären (default), können Sie jedoch sicherstellen, dass der aus dem Property ausgelesene Wert immer gültig ist.

Im Gegensatz zu einem Feld kann ein Property Details der Implementierung vor dem Benutzer verbergen. Beispielsweise lassen sich Daten intern verschlüsseln, für das Setzen

oder Lesen des Propertys aber unverschlüsselt anzeigen. Obwohl der Wert eines Propertys sich nach außen nur als einfache Zahl darstellt, könnte die Komponente diesen Wert in einer Datenbank abfragen oder mit komplizierten Verfahren errechnen.

Properties bieten auch erhebliche Vorteile im Komponentenbau, sowohl für Sie als Komponentenentwickler als auch für die Benutzer Ihrer Komponenten. Properties lassen sich in Komponenten im Entwurfsmodus (weil sie persistent sind) des Objekt-Inspektors anzeigen (`published`), darin liegt ihr offensichtlichster Vorzug.

Properties deklarieren

Die Deklaration eines Propertys und ihre Implementierung ist einfach und konsistent.

Die nötige Deklaration eines Propertys enthält drei Dinge:

1. Den Namen des Propertys
2. Den Typ des Propertys
3. Methoden für das Lesen und/oder Schreiben des Wertes

Die Properties einer Komponente sollten wenigstens im Abschnitt `public` deklariert sein, was ein einfaches Lesen und Schreiben des Propertys von außerhalb der Komponente zur Laufzeit ermöglicht. Es macht auch Sinn, Properties ausschließlich abgeleiteten Klassen als `protected` zu deklarieren.

So sieht eine typische Propertydeklaration aus:

```
TKonto = class(TBusObj)
  private
    FKreditLimit: double; //Feld für interne Speicherung
    FKontostand: double;
  protected
    function getKreditLimit: double;
    procedure setKreditLimit(newLimit: double);
  public
    constructor createKonto(cust_no: longInt; std_acc: byte);
    destructor destroy; override;
    property KreditLimit: double
          read getKreditLimit write setKreditLimit;
end;
```

Die Methode `read` eines Propertys ist eine Funktion, die keine Parameter annimmt und einen Wert vom Typ des Propertys zurückliefert. Gemäß Konvention ist der Name dieser Funktion `Get`, gefolgt vom Namen des Propertys. Die read-Methode für ein Property mit dem Namen `KreditLimit` würde also `getKreditLimit` heißen.

Die Regel „keine Parameter“ hat allerdings eine Ausnahme, nämlich bei Array-Properties, die ihre Indizes als Parameter übergeben (siehe Array-Properties).

Wenn Sie keine read-Methode deklarieren, erlaubt die Eigenschaft nur Schreibzugriff. Properties mit ausschließlichem Schreibzugriff sind sehr selten und in der Regel auch nicht sinnvoll.

```
function TKonto.getKreditLimit;
begin
  result:= FKreditLimit;
end;
```

Die Methode `write` eines Propertys kann eine Prozedur (oder ein Feld) sein, die genau einen Parameter annimmt. Der Parameter muss vom gleichen Typ sein wie das Property. Er lässt sich durch call by value oder über call by reference übergeben und kann jeden beliebigen gültigen Namen haben. Gemäß Konvention lautet der Name der Prozedur `Set`, gefolgt vom Namen des Propertys. Die write-Methode für das Property `KreditLimit` würde also `setKreditLimit` heißen.

Der im Parameter übergebene Wert wird verwendet, um den neuen Wert des Propertys zu setzen. Daher muss die write-Methode fähig sein, alle erforderlichen Behandlungen und Checks der Werte durchzuführen, um sie in der richtigen Form in das interne Feld `FKreditLimit` zu schreiben.

Wird keine write-Methode deklariert, gestattet die Eigenschaft nur Lesezugriff. Üblicherweise wird vor dem Setzen des Wertes überprüft, ob sich der neue Wert vom aktuellen unterscheidet. Auf die weiteren Business-Rules, die sich in meiner write-Methode befinden, gehen wir in Teil 2 mit Patterns genauer ein.

```
procedure TKonto.setKreditLimit(newLimit: double);
 begin
   if FKreditLimit <> newLimit then begin
     if getPassword(frmTrans.password) then
       FKreditLimit:= newLimit
     else MessageDlg(frmTrans.strLit[2]+'  '+ //multilang
          frmTrans.strLit[3], mtWarning,[mbok],0);
   end;
 end;
```

Standardwerte von Properties

Beim Deklarieren eines Propertys lässt sich optional ein Standardwert deklarieren. Der Standardwert für die Eigenschaft einer Komponente ist der Wert, der für diese Eigenschaft im Konstruktor der Komponente gesetzt ist. Wenn Sie zum Beispiel eine Komponente aus der Komponentenpalette in ein Formular einfügen, erzeugt Delphi die Komponente durch Aufrufen des Konstruktors, der den Anfangswert des Propertys der Komponente festlegt.

> *Eigentlich ist es nicht der Konstruktor direkt, da der Standardwert (Default) ja aus dem Form (DFM) gelesen wird.*

Zum Deklarieren eines Standardwerts für ein Property fügen Sie den Befehl default an die Deklaration (oder Neudeklaration) des Propertys an, gefolgt vom Standardwert.

```
property KreditLimit: double
   read getKreditLimit write setKreditLimit; default 5000.00;
```

Die Deklaration eines Standardwerts in der Deklaration setzt das Property **nicht** automatisch auf diesen Wert. Sie als Entwickler müssen sicherstellen, dass der Konstruktor der Klasse das Property tatsächlich auf den eingegebenen Wert setzt.

Erzeugen eines Array-Property

In der Regel haben read-Methoden keine Parameter oder Standards, hier folgt nun die mögliche und flexible Ausnahme:

Einige Properties bieten sich für eine Indizierung an, im Stil von Arrays. Sie bestehen dann aus mehreren Werten, die einer Art von Indexwert entsprechen. Ein Beispiel bei den Standardkomponenten ist das Property `Lines` der Memo-Komponente. `Lines` ist eine indizierte Liste von Strings, die den Text des Memos ausmachen, und diese Liste lässt sich wie ein String-Array behandeln. In diesem Fall gibt das Array-Property dem Benutzer Zugriff auf ein bestimmtes Element (einen String) in einer größeren Datenmenge (dem Memotext).

Array-Properties funktionieren genauso wie andere Properties und lassen sich im Wesentlichen in der gleichen Weise deklarieren. Ein Array-Property funktioniert nach außen hin wie ein normales Array, Sie greifen also beispielsweise mit `color[0]:= clWhite`; auf das nullte Element des Properties `Color` zu (eigentlich müsste es jetzt genau genommen Colors heißen[4]). Bei der Deklaration unterscheidet das fehlende of und das Anreihen von Dimensionen das Array-Property von einem normalen Array:

```
property Color[n: Byte]: TColor read GetColor write SetColor;
```

Ein Array-Property hat eine indizierte Eigenschaft. Sie werden beispielsweise für die Einträge eines Listenfeldes, die untergeordneten Objekte eines Steuerelements oder die Pixel einer Bitmap-Grafik verwendet. Ein anspruchsvolleres Beispiel folgt der Tatsache, dass ein Interface keine Felder enthalten darf, somit ist ein später zu implementierendes Array-Property die erste Wahl.

Ein großer Vorteil ist der Zugriffstyp auf das Array, der eigentlich jeden gültigen Typ darstellen kann. Auch der Indextyp selbst lässt sich bestimmen. Im Gegensatz zu Arrays, bei denen nur ordinale Indizes erlaubt sind, können die Indizes von Array-Eigenschaften einen beliebigen Typ haben. Zudem benötige ich die zugehörigen getter und setter und habe dann Wahlfreiheit, wie die eigentliche Datenstruktur daherkommt. Als Erstes definieren wir eine Klasse als Interface:

```
IChaosBase = Interface(IUnknown)
['{C6661345-26D1-D611-9FAD-C52B9EAAF7C0}']
  function getScales(index: integer): TDouble; stdcall;
  procedure setScales(index: integer; scale:TDouble); stdcall;
  property scales[index: integer]: TDouble
                   read getScales write setScales;
end;
```

[4] Den Plural setzt man für die Collections ein.

Das Ziel ist nun, für das Array-Property `scales` die Implementierung in einer eigenen Klasse bereitzustellen und die dazugehörige Datenstruktur zu definieren. Die Datenstruktur besteht aus den vier Werten eines Koordinatensystems:

```
 scaleX1: double;
 scaleX2: double;
 scaleY1: double;
 scaleY2: double;
```

Diese Argumente lassen sich in ein dimensioniertes Array packen. Sie sehen, einzig der Index ist in diesem Fall eine ganze Zahl, auf den Inhalt des Propertys lässt sich ein beliebiger Typ definieren. Die Zugriffsbezeichner sind keine Felder, sondern Methoden. Die Methode in einem read-Bezeichner muss eine Funktion sein, bei der Anzahl, Reihenfolge und Typ der Parameter mit der Indexparameterliste des Propertys identisch sind und der Ergebnistyp mit dem Typ des Propertys übereinstimmt.

Die Methode in einem write-Bezeichner muss eine Prozedur sein, bei der Anzahl, Reihenfolge und Typ der Parameter, wie bei `read`, mit der Indexparameterliste des Propertys identisch sind. Die implementierende Klasse hat nebst den Zugriffsmethoden auch die Datenstruktur als `TDoubleArray`, diese kann nebst einem Array auch eine Liste, Map oder ein Hash sein, die mit den Zugriffsmethoden wiederum übereinstimmen müssen:

```
TDoubleArray = array[1..4] of TDouble;

TChaosBase = class(TInterfacedObject, IChaosBase)
protected
  myscales: TDoubleArray;
  function getScales(index: integer): TDouble; stdcall;
  procedure setScales(index: integer; scale: TDouble); stdcall;
public
  property scales[index: integer]: TDouble
                      read getScales write setScales;
end;
```

Die eigentlichen setter und getter legen den Typ fest, das Property hingegen verbirgt den Typ, hier sind nur die erforderlichen read- und write-Bezeichner ersichtlich. Zugriff auf die interne Objektstruktur erfolgt dann über den Index:

```
function TChaosBase.getScales(index: integer): TDouble;
begin
  result:=myscales[index];
end;

procedure TChaosBase.setScales(index: integer; scale: TDouble);
begin
  myScales[index]:=scale;
end;
```

Nach außen ist nur das öffentliche Property ersichtlich, sodass auch hier der Index von Bedeutung ist, der am Beispiel des Lesens endlich das Array belästigt:

```
scaledX:= (X-scales[1])/(scales[2]-scales[1]);
scaledY:= (Y-scales[4])/(scales[3]-scales[4]);
```

Auf die Array-Standardeigenschaft einer Klasse ist mit der Kurzform Objekt[Index] der Zugriff direkt möglich, ansonsten müsste dort jedesmal `X-ObjBase.scales[1]` stehen. Diese Anweisung ist mit Objekt.Eigenschaft[Index] identisch.

Ereignisse sind auch Properties

Komponenten verwenden auch Properties, um ihre Ereignisse zu implementieren. Anders als die meisten anderen Properties verwenden Ereignisse (Events) keine Methoden, um ihre Funktionen `read` und `write` zu implementieren, weil im Property eigentlich schon die Methode steckt. Ereignisproperties arbeiten stattdessen mit einem privaten Feld vom gleichen Typ wie die Properties.

Zum Beispiel speichert der Compiler den Zeiger der Methode `OnClick` in einem Feld namens `FOnClick` des Typs `TNotifyEvent`. Die Deklaration des Ereignis-Property `OnClick` ohne Zugriffsmethoden sieht wie folgt aus:

```
type TControl = class(TComponent)
//Feld deklarieren, das den Methodenzeiger enthalten soll
private
  FOnClick: TNotifyEvent;
protected
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;
```

Wie bei jedem anderen Property können Sie auch den Wert eines Ereignisses zur Laufzeit setzen oder ändern. Der Hauptvorteil dafür, dass Ereignisse auch Properties sind, liegt jedoch darin, dass Sie Ereignissen Behandlungsroutinen (Eventhandler) mit dem Objekt-Inspektor im Entwurfs- und Laufzeitmodus direkt zuweisen können. Einige Patterns beruhen auf dieser Technik, welche zudem viele OO-Sprachen innehaben.

Der Botschaftsbehandlung liegt folgendes Prinzip zugrunde: Die Klasse empfängt Botschaften und ruft – in Abhängigkeit von der Art der Botschaft – eine bestimmte Methode aus einer Methodengruppe auf, die vorher für die Botschaftsbehandlung definiert wurde, und gibt diese Botschaft oder Nachricht als Ereignis an die Anwendung weiter. Die Anwendung reagiert dann, sofern man einen Eventhandler zugewiesen hat. Falls für eine Botschaft keine entsprechende Behandlungsmethode existiert, erfolgt ein Aufruf der Standard-Behandlungsroutine.

```
Botschaft: WMLButtonDown → Ereignis: OnClick → Eventhandler:
Button1Click(Sender: TObject)
```

Die Windows-Umgebung benutzt ja Fenster als grundlegende Bausteine der Benutzerschnittstelle. Eigentlich sind alle visuellen Elemente, beispielsweise auch Schaltflächen

oder Editboxen, in Windows Fenster. Bei der visuellen Programmierung ist der Entwicklung ein neuer Schritt vorangestellt. Bevor man nämlich mit der eigentlichen Codierung beginnt, ist im Drag-und-Drop-Verfahren eine visuelle Benutzerschnittstelle zu gestalten.

Die Grundbausteine sind Formulare. Formulare sind spezielle Fenster, die andere visuelle und nicht visuelle Elemente enthalten können. Wie jedes Windows-Fenster enthalten Formulare mindestens verstellbare Ränder und eine Titelleiste mit Systemmenü und Schaltflächen, um das Formular zu minimieren, zu vergrößern oder zu schließen.

Unter Windows werden Systembenachrichtigungen in Form von Windows-Nachrichten (auch Botschaften genannt) direkt an eine Anwendung und deren Steuerelemente gesendet. Dieses Verhalten wird von CLX-Anwendungen nicht unterstützt, weil CLX eine Bibliothek für plattformübergreifende Anwendungen ist und Windows-Nachrichten unter Linux nicht verwendbar sind. CLX reagiert deshalb mit einer plattformneutralen Methode, unter KDE oder GNOME, auf Systembenachrichtigungen.

In CLX entspricht den Windows-Nachrichten ein auf der Widget-Ebene beruhendes Benachrichtigungssystem. Wie in der VCL, wo Windows-Botschaften entweder vom Betriebssystem oder durch die von VCL gekapselten Windows-Steuerelemente aktiviert sind, unterscheidet auch die Widget-Ebene von CLX diese beiden Meldungsarten. Handelt es sich um eine Widget-Nachricht, wird sie als Signal bezeichnet.

Stammt sie dagegen vom Betriebssystem, heißt sie Systemereignis. Die Widget-Ebene reicht Systemereignisse als Signale des Typs `event` an die CLX-Komponenten weiter.

Die folgende Tabelle listet den Vergleich auf, da einige Interface-Techniken auf das Weiterreichen von Ereignissen angewiesen sind:

	Windows	**CLX auf Linux**
Library	Win32 API /Tcontrol	C-Lib /Qt /TWidget
Botschaftsname	Messages (Nachrichten)	System_event
Ereignisname	Event	Signal
Bearbeiter	On_Eventhandler	On_Slot_Event

Tab. 1.3: Ereignissteuerung in Windows versus CLX

1.2.3 Vererbung

Schnittstellenvererbung

Objektorientierte Sprachen wie Design Patterns zeichnen sich u. a. durch Vererbung aus. In OO-Einführungen wird dies im Allgemeinen so erklärt, dass konkrete Klassen von allgemeineren abstammen und deren Code erben. Dies ist aber nur die halbe Wahrheit: Etwas tiefer gehende Abhandlungen zu objektorientierter Programmierung unterscheiden zwischen Implementierungs- und **Schnittstellenvererbung**.

Eine Schnittstelle erbt wie eine Klasse alle Methoden ihres Vorfahren. Schnittstellen implementieren aber im Gegensatz zu Klassen keine Methoden.

Eine Schnittstelle erbt die Verpflichtung zur Implementation von Methoden. Diese Verpflichtung geht auf alle Klassen über, welche die Schnittstelle unterstützen.

Die Implementierungsvererbung ist das, was man im Allgemeinen als Vererbung kennt. Für einen Onlinebroker mit verschiedenen Anlagen könnte das so aussehen: Eine konkrete Klasse, z. B. SuperIBM, stammt von der Klasse TEDVAnlage ab und erbt von dieser gewisse Methodenimplementierungen, z. B. getPreis. Die Klasse TEDVAnlage kann nun wiederum eine Unterklasse der noch allgemeineren Klasse TAnlage sein und getPreis ebenfalls bereits erben.

Eingeschränkt wird das Konzept der Vererbung in allen modernen objektorientierten Sprachen (Delphi, Java, Ada) dadurch, dass eine Klasse nicht von mehreren Klassen erben darf [xi]. Mehrfachvererbung wie in C++ ist nicht erlaubt. Allerdings gilt dies nur für die Implementierungsvererbung. Schnittstellen kann eine Klasse von mehreren Überklassen erben.

Java und Delphi unterscheiden hier zwischen **inherits** für Implementierungsvererbung und **implements** für Schnittstellenvererbung, was von den Begriffen her eher gewöhnungsbedürftig, aber doch entscheidend ist. In C# wird die Syntax wie bei Delphi im Klassenkopf nicht unterschieden. Hier hilft nur eine konsequente Umsetzung der Namenskonvention, dass Interfaces immer mit I beginnen, wenn man die zwei unterscheiden will. Eine Bilddatei könnte also von einer Dateiklasse Code erben und von weiteren Interfaces die Schnittstelle. Im Multikulti-Code sieht dies so aus:

```
Java: class ImageFile inherits File implements IfileOperation
Delphi: TImageFile = class (TFileInfo, IFileOperation)
C# : class ImageFile : FileInfo, IFileOperation
```

Nun wissen Sie zwar, wie man Interfaces im Code benutzt, aber vielleicht haben Sie immer noch nur eine rudimentäre Vorstellung davon, was Schnittstellenvererbung denn eigentlich bedeutet. Dies wird in den nächsten Stufen verdeutlicht.

Polymorphie

Nun geht es um einen faszinierenden Pfeiler der OOP (Kapselung, Vererbung), um die Polymorphie (Vielgestaltigkeit). Diese Technik wird (muss sogar) auch von Interfaces oder abstrakten Klassen intensiv genutzt. Dahinter verbirgt sich die Idee, virtuelle **Methoden mit gleichem Bezeichner** auf alle Instanzen einer vererbten Objekthierarchie anwenden zu können, obwohl sie innerhalb dieser Methoden unterschiedlich reagieren müssen oder können.

Diese abstrakten Gesetze treffen wir auch in der Wirklichkeit an, wo z. B. jedes Fortbewegungsmittel eine Methode Fahren kennt, jedoch anders realisiert. Eben diese Fähigkeit, gleichartige Prozesse in unserer Umwelt zu erkennen und aus ihnen ein abstraktes Prinzip herzuleiten, ist die Stärke der Polymorphie. Die Vielgestaltigkeit soll also dem Entwickler mit virtuellen Methoden helfen, die fachlichen Operationen näher an die Realität zu bringen.

Diese zur Laufzeit generierte Flexibilität ist aber nur möglich, wenn die Referenz auf eine Methode nicht schon beim Kompilieren, sondern erst während der Laufzeit in Abhängig-

keit der Typen festgelegt wird. Delphi (eigentlich auch alle anderen OO-Sprachen) löst das Problem des „late binding“ mithilfe der so genannten Virtual Method Table (VMT), die ein Teil jeder Klasse ist.

Das ist eine Liste mit Methodenzeigern, die man für jede Klasse mithilfe des Konstruktors automatisch anlegt, sofern die Klasse über virtuelle Methoden verfügt. Konkret zur Sache geht es dann in der Umsetzung der Design Patterns in Teil 2 des Buches, wo dank der **Vererbung**[5] jede Kontoklasse mit der virtuellen Methode `Buchen` ein eigenes Verhalten zeigt:

```
TKonto = class(TBusObj)
  private
    dataList: TStringList;
    FName: string;
    FStammCode: longInt;
    FKreditLimit: double;
    FBasisGebuehr: double;
    FKontostand: double;
    FDiffKonto: byte;
    FTransakt_Obj: TTransaction; //aggregation
  protected
    function getKreditLimit: double;
    procedure setKreditLimit(newLimit: double);
    procedure Buchen(betrag: double); virtual;
    .....
```

Das erste 4-Byte-Feld eines jeden Objekts ist ein Zeiger auf die Tabelle der virtuellen Methoden (VMT) der Klasse. Es gibt nur eine VMT pro Klasse (und nicht eine für jedes Objekt). Zwei verschiedene Klassentypen können eine VMT jedoch nicht gemeinsam benutzen.

> *VMTs erstellt der Compiler automatisch und lassen sich nie direkt von einem Programm bearbeiten. Ebenso speichern die Konstruktoren die Zeiger auf VMTs automatisch in den erstellten Objekten.*

Die folgende gekürzte Tabelle aus dem Borland-Archiv zeigt die Struktur einer VMT. Bei positiven Offsets besteht eine VMT aus einer Liste mit 32-Bit-Methodenzeigern. Für jede benutzerdefinierte virtuelle Methode des Klassentyps ist ein Zeiger vorhanden. Die Zeiger sind in der Reihenfolge der Deklaration angeordnet. Jeder Eintrag enthält die Adresse des Eintrittspunktes der entsprechenden virtuellen Methode.

Dieses Layout ist zur V-Tabelle von C++ und zu COM kompatibel. Bei negativen Offsets enthält eine VMT die Anzahl der Felder, die in Delphi intern implementiert sind. In einer Anwendung sollten diese Informationen mit den Methoden von TObject abgerufen werden, da sich dieses Layout bei künftigen Implementierungen von Delphi ändern kann.

[5] Polymorphie ohne Vererbung ist sinnlos.

Offset	Typ	Beschreibung
-76	Ptr	Ptr auf virtuelle Methodentabelle
-72	Ptr	Ptr auf Schnittstellentabelle
-68	Ptr	Ptr auf Informationstabelle zur Automatisierung
-64	Ptr	Ptr auf Instanzen-Initialisierungstabelle
-60	Ptr	Ptr auf Informationstabelle des Typs
-56	Ptr	Ptr auf Tabelle der Felddefinitionen
-52	Ptr	Ptr auf Tabelle der Methodendefinitionen
-48	Ptr	Ptr auf die Tabelle der dynamischen Methoden
-44	Ptr	Ptr auf String, der den Klassennamen enthält
-40	Card.	Instanzengröße in Byte
-36	Ptr	Ptr auf die übergeordnete Klasse (oder NIL)
-12	Ptr	Eintrittspunkt der Methode NewInstance
-08	Ptr	Eintrittspunkt der Methode FreeInstance
-04	Ptr	Eintrittspunkt des Destruktors Destroy
+00	Ptr	Eintrittspunkt der ersten benutzerdefinierten virtuellen Methode
+04	Ptr	Eintrittspunkt der zweiten benutzerdefinierten virtuellen Methode

Tab. 1.4: Struktur der VMT

1.2.4 Einsatz von Interfaces

Zu was dienen eigentlich Interfaces in Object Pascal? Dieses Leistungsmerkmal haben die Borländer zur Unterstützung von COM erschaffen (seit Delphi 4), es wurde aber unabhängig von COM weiter implementiert, d. h., die Mechanismen funktionieren sogar in Kylix und lassen sich unter Windows losgelöst von COM in Design und Integration Gewinn bringend einsetzen. Zeit also, einen Blick hinter die Kulissen der Zwischengesichter zu werfen, sozusagen „Face to Face“.

Schnittstellen bieten einige Vorteile der Mehrfachvererbung, umgehen aber deren semantische Probleme. Außerdem sind sie bei der Verwendung von verteilten Objektmodellen von größter Bedeutung (z. B. CORBA und SOAP). Benutzerdefinierte Objekte, die Schnittstellen unterstützen, können dabei mit Objekten interagieren, die man mit C++, Java oder anderen Programmiersprachen entwickelt hat.

In einem Interface wird nur die Schnittstelle einer Klasse definiert, und zwar ohne den dazugehörigen Code. In Java besteht die Schnittstelle aus den öffentlichen Methoden, in

Delphi wie in C# kommen Eigenschaften (Properties) und Indexer dazu. Wozu dient das nun? Schnittstellenvererbung kann man sich vorstellen wie einen **Vertrag**, den eine Klasse mit einer sie aufrufenden Klasse abschließt.

> *Wenn eine Klasse ein bestimmtes Interface implementiert, dann garantiert sie der aufrufenden Klasse, dass sie alle Methoden und die weiteren öffentlichen Elemente dieses Interface implementiert.*

Die aufrufende Klasse braucht sich um den Typ und die innere Struktur dieser Klasse nicht zu kümmern, sie arbeitet nur mit dem Interface. Dies ermöglicht es zum Beispiel, eine Liste von Objekten abzuarbeiten und bei allen in der Liste enthaltenen Objekten dieselbe Methode aufzurufen, obwohl die Objekte verschiedene Typen haben. Dies funktioniert, solange sie alle dasselbe Interface implementieren.

Dafür gibt es auch Beispiele in der realen Welt: Um einen ICE zu fahren, muss ich nicht Fahrstunden mit genau diesem Modell genommen haben. Es genügt, dass ein Fahrzeug die Schnittstelle „ICE-Hochgeschwindigkeitszug“ erfüllt, d. h. via Beschleuniger und Führerstandsanzeige und noch ein paar Instrumente dazu Methoden wie Anfahren, Beschleunigen oder Bremsen zurVerfügung stellt, damit ich den Zug führen kann. ;)

1.2.5 Designfragen

Das Object Pascal-Schlüsselwort `interface` erlaubt das Erstellen und Verwenden von Schnittstellen in Anwendungen. Definieren wir mal:

> *Ein Interface ist eine Liste von Zeigern auf Methoden.*

Schnittstellen erweitern das Modell der Einfachvererbung der VCL, indem einzelne Klassen mehr als eine Schnittstelle implementieren können. Gleichzeitig können mehrere Klassen mit unterschiedlichen Basisklassen gemeinsam auf ein Interface zugreifen. Wozu das gut sein soll? Schnittstellen sind hilfreich, wenn beispielsweise Stream-Operationen, Transferoperationen oder Sicherheitsfunktionen für viele verschiedene Objekte erforderlich sind, die aber wegen Typenverträglichkeit nicht in den jeweiligen Basisklassen enthalten sind.

Außerdem bilden Schnittstellen eine der Grundlagen der erwähnten verteilten Objektmodelle DCOM, CORBA und SOAP und sind auch bekannte Konstrukte anderer Sprachen, da ja ein registriertes Interface schon als Type-Library gilt.

> *Eine Type-Library ist eine editierbare Datei mit der Extension *.TLB, die ein Interface definiert.*

Beim Design umfangreicher Komponenten oder z. B. eines Open Tool API, gelangt man schon mal an die Grenzen einer linearen Objekthierarchie und lernt den Einsatz verschiedener Basisklassen schätzen. So lassen sich universelle Methoden mehrfach verwenden, ohne immer von der gleichen Hierarchie abhängig zu sein oder ständig in die Bibliothek eingreifen zu müssen.

Ich habe das Borland-Beispiel in ein Klassendiagramm umgesetzt (Abb. 1.14), um auf einen Blick die Stärken von Interfaces darzustellen. Es sind zwei Basisklassen vorhan-

den, die `TSquare` und `TCircle` unterstützen. Nun erscheint es vernünftig, das `TCircle` zwar `paint` implementiert, aber nicht `Rotate`, da es bei einem Kreis wahrlich nichts zu drehen gibt.

Die beiden Methoden sind so universell, dass man sie aus Designgründen, wie der besseren Granularität und Wiederverwendung, nicht einfach in eine Basisklasse wie `TPolygonObject` stecken will. Gut, solche Konstrukte fallen nicht einfach vom Himmel, sie entstehen langsam nach diversen Reviews oder Refactoring im Schweiße des Zwischengesichtes.[6]

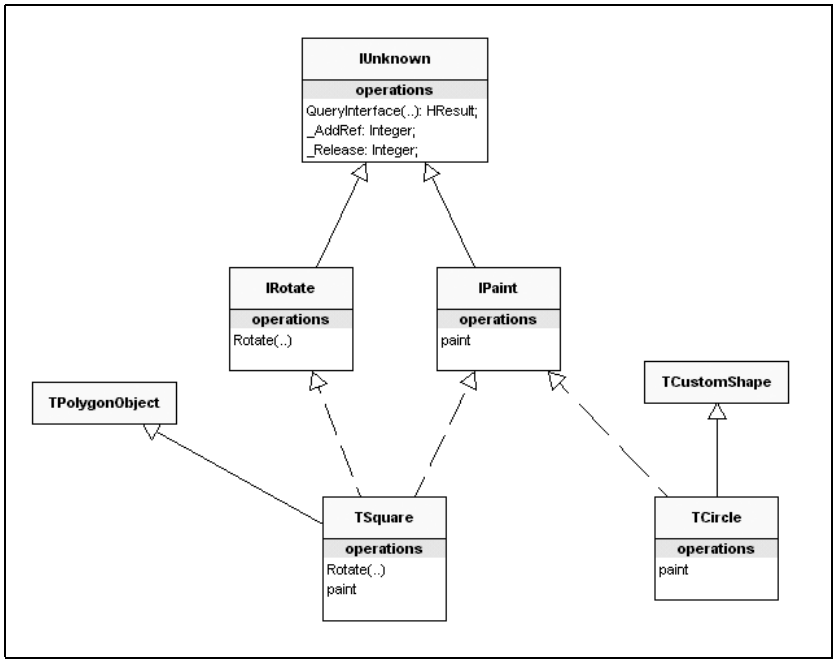

Abb. 1.14: Das Borland-Beispiel als Klassendiagramm

Wir sehen aber gleich praktisch mit dem Beispiel einer Zinsratenberechnung, dass Schnittstellen auch ohne kompliziertes Design oder dem Umsetzen von bewährten und standardisierten Patterns ein paar tolle Merkmale aufweisen, die sich schon im Einzelfall pro Klasse lohnen können:

- N-Objekte sind zuweisungskompatibel, wenn sie dasselbe Interface unterstützen
- Automatische Speicherfreigabe durch Referenzzähler
- Bequemes Typecasting mit eingebauten Funktionen
- Interface-Name wird durch ein GUID weltweit eindeutig
- Verbergen der Implementierungsdetails

[6] A Trainstation is a station where a train stops, what about a workstation?

Eine Schnittstelle ähnelt einer Klasse, die nur abstrakte Methoden und eine genaue Definition ihrer Funktionalität enthält, jedoch besitzt sie selbst kein Grundverhalten. Solange die Schnittstelle von `IUnknown` erbt und nicht von `IDispatch` sind wir noch unabhängig von der COM-Jacke (ich habe nichts von einer Zwangsjacke gesagt).

```
unit income1;
IIncomeInt = interface (IUnknown)
    ['{DBB42A04-E60F-41EC-870A-314D68B6913C}'] //GUID
    function GetIncome(const aNetto: Currency):Currency; stdcall;
    function GetRate: Real;
    function queryDLLInterface(var queryList: TStringList):
                                        TStringList; stdcall;
    procedure SetRate(const aPercent, aYear: integer); stdcall;
    property Rate: Real read GetRate;
  end;
```

Eine mit einem Schnittstellentyp deklarierte Variable wie im folgenden `incomeIntRef` kann Instanzen jeder Klasse referenzieren, die man von der Schnittstelle implementiert. Das Objekt muss einfach die Methoden der Schnittstelle unterstützen. Das Typische an einer Schnittstellenvariablen in der Deklaration ist eben, die Referenz auf die Schnittstelle und nicht auf die Klasse direkt zu setzen:

```
private
  incomeIntRef: IIncomeInt;
begin
   //in einer DLL+
  incomeIntRef:=createIncome;
   //in einer DCU
  incomeIntRef:=TIncomeRealIntf.create;
```

Das Schema für das Vorgehen sieht folgendermaßen aus:

1. Definieren der Schnittstelle
2. Implementieren der Schnittstellenmethoden in der Klasse
3. Schnittstelle in Unit referenzieren
4. Deklaration als Member in einem Client: Variable von Schnittstelle
5. Definition im Konstruktoraufruf: Objekt an Variable zuweisen

Eine mit einem Schnittstellentyp deklarierte Variable kann Instanzen jeder Klasse referenzieren, die man von der Schnittstelle implementiert. Mithilfe solcher Schnittstellenvariablen lassen sich Schnittstellenmethoden auch aufrufen, wenn zur Compilierzeit noch nicht bekannt ist, wo die Schnittstelle implementiert wird, jedoch sind Namen und Signaturen schon festgelegt.

In der Teamarbeit ein unschätzbarer Vorteil. In unserem Beispiel bauen wir auf der Technik von DLL+ auf [xii] [DE01], d. h., die Implementation befindet sich in einer DLL, welche die Referenz des Objektes `TIncomeRealIntf` exportiert. Sonst bleibt vom Prinzip her alles gleich, d. h., Sie entscheiden ob Implementierung und Interface in derselben

Datei sind, einzeln in einer separaten DCU oder eben ausgelagert in einer DLL. Ich empfehle aber, die Schnittstellen immer in eine eigene Datei zu legen, wie `unit income1` in Abb. 1.15 es demonstriert.

Schnittstellen sind wie Klassen nur im äußersten Gültigkeitsbereich eines Programms oder einer Unit, nicht aber in einer Prozedur oder Funktion zu deklarieren. Eine Schnittstellendeklaration ähnelt in weiten Teilen einer Klassendeklaration. Es gelten jedoch folgende Einschränkungen:

- Die Elementliste darf nur Methoden und Eigenschaften enthalten. Felder sind in Schnittstellen nicht erlaubt. Da für Schnittstellen keine Felder verfügbar sind, sind Properties und die entsprechenden Methoden (getter, setter, letter) einzusetzen
- Strukturen lassen sich nur über Array-Properties definieren und ansprechen.
- Man benötigt die Technik der Referenzzählung mit Konsequenzen, da man die gewohnten Free- oder Releasemechanismen überdenken muss.
- Alle Elemente einer Schnittstelle sind als public deklariert. Sichtbarkeitsattribute und Speicherattribute sind nicht erlaubt. Es lässt sich aber ein Array-Property mit der Direktive default als Standardeigenschaft deklarieren.
- Schnittstellen haben keine Konstruktoren oder Destruktoren, haben keine Grundfunktionalität und lassen sich im Gegensatz zu **abstrakten Klassen** auch nicht instanzieren, ausgenommen durch die Klassen selbst, welche die Methoden implementieren.
- Methoden-Bindings wie `virtual, dynamic, abstract` oder `override` sind nicht erlaubt. Da Schnittstellen keine eigenen Methoden implementieren, haben diese Bezeichnungen auch keine Bedeutung.

  Tipp: Wenn eine Methode in einer Vorfahrklasse als abstract deklariert ist, gilt sie als abstrakte Klasse. Abstrakte Klassen können jedoch konkrete Methoden enthalten. Die abstrakten Methoden müssen Sie (durch Neudeklaration und Implementierung) in einer abgeleiteten Klasse implementieren.

Delphi kann wie bei Schnittstellen keine Instanzen einer Klasse erzeugen, die abstrakte Elemente enthält.

1.2.6 Interface konkret

Genug von Theorie und Design, ich komme von der Schnittstelle zur Baustelle, d. h. zur weiteren Implementierung. Unser Beispiel einer Zinseszinsberechnung in Abb. 1.16 besteht aus mindestens drei Units:

Die Library *income.dpr* beinhaltet als DLL die eigentlichen konkreten Methoden und die zu exportierende Referenz als Funktion.

Die Unit *income1.pas*, beinhaltet die Schnittstelle `IIncomeInt`, weitere Interfaces lassen sich hinzufügen.

Und es gibt eine Unit, die den Client als `TfrmIncome` mit der Referenz darstellt.

Die Library *income.dpr*, die ja von der Schnittstelle, d. h. von der Klasse `IIncomeInt`, erbt, implementiert die Methoden nach Wunsch und Anforderung in einer DLL. Wie

gesagt, die Implementation lässt sich auch in eine DCU oder ein Package packen. Die Diskussion DLL versus Packages soll hier jedoch nicht geführt werden.

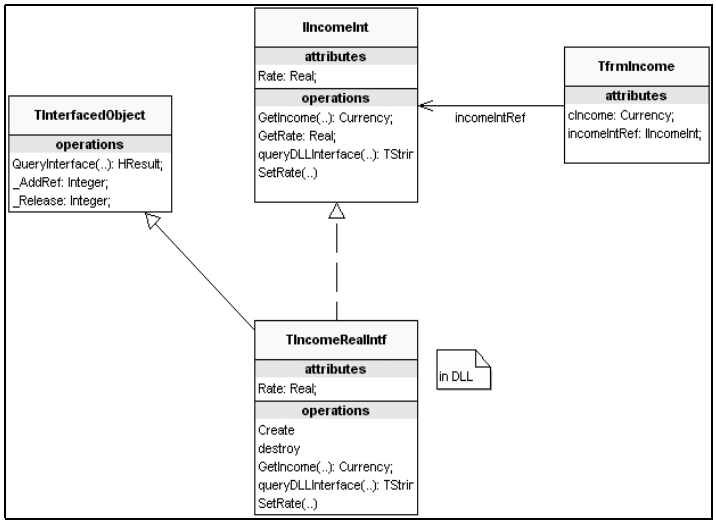

Abb. 1.15: Das Klassendiagramm mit dem Interface

Ein Package ist eine spezielle Art von DLL, die von Anwendungen, der IDE oder von beiden verwendbar ist. Spezifisch für die Klasse ist nun, dass von der Schnittstelle einerseits und von einer Basisklasse andererseits geerbt wird. Auf diese Basisklasse gehe ich nun näher ein.

Über allem steht in Delphi die Deklaration von `TInterfacedObject` aus der Unit System, da der Vorfahr ja `TObject` sein muss und zugleich `IUnknown` jeweils implementiert. Der Name `IUnknown` rührt daher, weil die Schnittstelle ihre unterstützenden Klassen nicht kennt.

`TInterfacedObject` implementiert also die Schnittstelle `IUnknown`. Daher deklariert und implementiert `TInterfacedObject` für uns jede der drei Methoden `QueryInterface`, `_AddRef` und `_Release` von `IUnknown` und eignet sich aus diesem Grund als Basis für jede weitere Klasse, die eine Schnittstelle implementiert. Hier ein Blick in diese Hilfsklasse:

```
TInterfacedObject = class(TObject, IUnknown)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj):
                                              HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    class function NewInstance: TObject; override;
```

```
    property RefCount: Integer read FRefCount;
  end;
```

Das eigentliche Arbeitspferd ist die konkrete Klasse mit der Implementierung in der DLL, die von der Basisklasse als Hilfsklasse wie auch von der Schnittstelle ja die Verpflichtung erbt, zu realisieren, und dies ganz ohne Erbschaftssteuern ;):

```
TIncomeRealIntf = class(TInterfacedObject, IIncomeInt)
    FRate: Real;
    function Power(X: Real; Y: Integer):Real;
  protected
    function GetRate:Real;
  public
    constructor Create;
    destructor destroy; override;
    function GetIncome(const aNetto: Currency):
                                  Currency; stdcall;
    .....
    property Rate: Real read GetRate;
  end;
```

Wenn Sie mit DLL+ arbeiten, muss noch die Referenz zum Exportieren befähigt werden. Der eigentliche Export befindet sich in einer globalen Funktion, die als Rückgabewert die Referenz über den Konstruktor der Klasse zurückgibt. Dann benötigt man nur noch das Schlüsselwort `exports`.

```
function CreateIncome: IIncomeInt; stdcall;
begin
  result:= TIncomeRealIntf.Create;
end;
exports CreateIncome;
```

1.2.7 Speicher und Referenz

Da die Klasse `TInterfacedObject` die Methoden von `IUnknown` implementiert, führt sie automatisch die Referenzzählung und die Speicherverwaltung für Schnittstellenobjekte durch.

Eines der grundlegenden Konzepte beim Entwurf von Schnittstellen besteht in der Sicherstellung der Referenzverwaltung für die Objekte, welche die Schnittstellen implementieren. Die IUnknown-Methoden `_AddRef` und `_Release` ermöglichen die Implementierung dieser Funktionalität. Sie überwachen die Existenz eines Objekts mithilfe eines Referenzzählers. Dieser lässt sich durch die Funktion jedesmal erhöhen, wenn eine Schnittstellenreferenz an einen Client übergeben wird. Sobald der Referenzzähler den Wert Null erreicht, wird das Objekt freigegeben.

Bei einem Objekt, das ausschließlich durch Schnittstellen referenziert wird, ist eine manuelle Freigabe nicht mehr nötig. Seine Freigabe erfolgt automatisch, wenn der Refe-

renzzähler den Wert Null erreicht. In unserem Beispiel erfolgt das Erzeugen der **lokalen** Schnittstellenreferenz (`incomeIntRef:= createIncome`) direkt auf Mausklick, wir erzwingen also bei jedem Durchlauf die Berechnung der Zinsen und stellen fest, das Objekt wird bei jeder neuen Berechnung wieder mit `destroy` aus dem Speicher entfernt.

Warum? Weil bei jedem neuen Konstruktoraufruf `IUnknown` automatisch das alte Objekt entfernt, obwohl kein `_Release` oder Ähnliches im Spiel ist! Denn beim Verlassen der Prozedur wird der Zähler reduziert oder auf null gesetzt. Auch beim Schließen des Clients erfolgt der gleiche Freigabemechanismus und auch hier wird `destroy` automatisch aufgerufen.

```
destructor TIncomeRealIntf.destroy;
begin
  messagebox(0,'automatic release','TIncomeRealIntf',mb_OK)
end;
```

Im Experiment lässt sich der Automatismus mit dem expliziten Setzen von `AddRef` auch mal austricksen, d. h., wir treiben den Zähler künstlich in die Höhe und geben vor, dass mindestens eine Referenz noch im Spiel ist, d. h., wir verhindern also ständig die Freigabe. Hier hilft nur ein erneutes Release, sodass je nach Design und Dynamik die eingebauten Automatismen nicht immer von Vorteil sind:

```
procedure TfrmIncome.BitBtnOKClick(Sender: TObject);
begin
  incomeIntRef:=createIncome;
  with incomeIntRef do begin
    if QueryInterface(IIncomeInt, incomeIntRef) = S_OK
    then begin
      //_addRef;to test
      SetRate(strToInt(edtZins.text),
                       strToInt(edtJahre.text));
.....
```

Man könnte sogar so weit gehen, alle Delphi-Objekte automatisch entfernen zu lassen. Das heißt, ich implementiere über den Reference-Counter-Mechanismus eine allgemeine Garbage Collection, indem jedes neue Objekt vorgängig dem Konstruktor einer Schnittstelle übergeben wird, die dann automatisch die Freigabe des Objektes übernimmt.

Diese Umlenkung hat auch klare Einschränkungen zur Folge, die alle Komponenten mit owner-Eigenschaft betreffen. Hier ist ein Konflikt mit Delphi vorprogrammiert. Dennoch, ein erster Ansatz solch einer konstruierten Garbage Collection könnte so aussehen:

```
Type
  ISelfDestroy = interface

TSelfDestroy = class (TInterfacedObject, ISelfDestroy)
  private
    FObject: TObject;
```

```
  public
    constructor Create(aObject: TObject);
    destructor destroy; override;
    .....
constructor TSelfDestroy.create(aObject: TObject);
begin
  FObject:= aObject);
end;
```

Noch ein Wort zur Polymorphie. Wenn die Implementierung und Signatur der Klasse der Schnittstellendefinition entspricht, ist die Schnittstelle vollständig polymorph: Zugriff und Verwendung der Schnittstelle ist also für alle späteren Implementierungen dieser Schnittstelle identisch.

Wir könnten also noch eine zweite Klasse, z. B. `TIcomeReal2Intf i`mplementieren, die eine andere Art der Berechnung durchführt. Dann einfach Referenz im Client ändern, und es steht eine andere Implementierung derselben Methoden im Einsatz. Dies machen sich, wie in Teil 2 zu sehen ist, einige der Entwurfs-Muster zu Nutze.

Mit Schnittstellen eröffnet sich eine neue Möglichkeit zur Trennung von Einsatz und Implementierung einer Klasse. Zwei Klassen können also dieselbe Schnittstelle verwenden, auch wenn sie keine Nachkommen derselben Basisklasse sind.

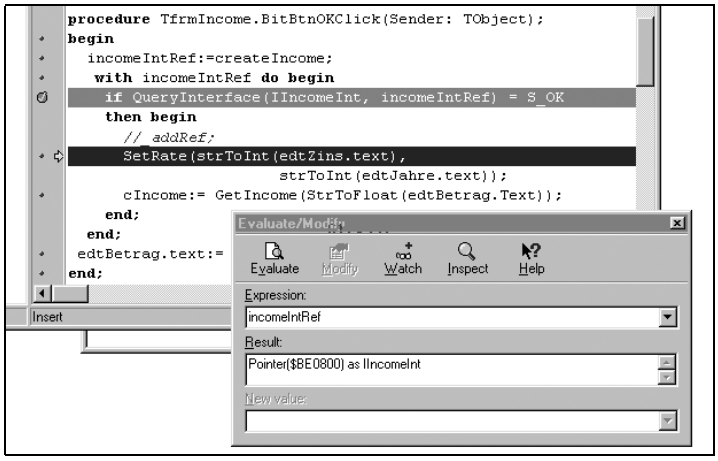

Abb. 1.16: Sicheres Typecasting mit QueryInterface

Wenn Klassen Schnittstellen implementieren, können sie den Operator `as` für die dynamische Bindung an die Schnittstelle verwenden. Spätestens hier benötige ich eine IID als GUID, welche sich direkt im Code-Editor mit SHIFT+CTRL+G fabrizieren lässt und etwas „Einzigartiges“ darstellt.

> *Ein GUID (Schnittstellen-ID) ist ein binärer 16-Byte-Wert, der eine Schnittstelle weltweit eindeutig bezeichnet (nach Wahrscheinlichkeitsrechnung).*

Wenn eine Schnittstelle ein GUID hat, können Sie über eine Schnittstellenabfrage Referenzen auf ihre Implementationen abrufen.

Mithilfe des Operators `as` können Schnittstellenumwandlungen durchgeführt werden. Dieser Vorgang wird auch als Schnittstellenabfrage bezeichnet und macht dann Sinn, wenn so mehr als drei Interfaces im Spiel sind. In Abb. 1.16 ist ersichtlich, dass beim Aufruf von `QueryInterface` immer ein `as` im Spiel ist. Eine Schnittstellenabfrage generiert auf der Basis des (zur Laufzeit vorliegenden) Objekttyps aus einer Objektreferenz oder einer anderen Schnittstellenreferenz einen neuen Ausdruck vom Typ einer Schnittstelle. Die Syntax für eine Schnittstellenabfrage lautet:

```
Object as Interface //as Operator
```

Objekt ist entweder ein Ausdruck vom Typ einer Schnittstelle oder Variante, oder er bezeichnet eine Instanz einer Klasse, die eine definierte Schnittstelle implementiert. Also angenommen wir haben zwei Interfaces und rufen den Konstruktor für die erste Schnittstelle auf.

Diese Instanz möchte wir nun auf die zweite Schnittstelle `IUserAccount` anwenden und erhalten eine Umwandlung unserer ersten Instanz, ohne erneuten Aufruf des Konstruktors. Beispiel gefällig:

```
var incomeIntRef: IIncomeInt;
    userRef: IUserAccount;
begin
  incomeIntRef:= TIncome.create
  incomeIntRef.setRate();
  userRef:= incomeIntRef as IUserAccount
  userRef.setWAPAccount()
```

Im Gegensatz zu einer direkten Typumwandlung mit `as` (die bei Fehler eine Exception auslöst) lässt sich mit `QueryInterface` einfach und locker nachfragen, ob ein definiertes Interface vom Objekt unterstützt wird:

```
  if QueryInterface(IIncomeInt, incomeIntRef)
                       = S_OK then begin
```

Wenn ein Objekt den Wert `NIL` hat, liefert die Schnittstellenabfrage als Ergebnis `NIL`. Andernfalls wird das GUID der Schnittstelle an die Methode `QueryInterface` übergeben. Wenn `QueryInterface` keinen gültigen Vergleich anbringen kann, wird NIL gesetzt. Ist der Rückgabewert dagegen Null oder gültig (d. h., das Objekt unterstützt mit seiner Implementation die Schnittstelle), ergibt die Schnittstellenabfrage eine Referenz auf das Objekt zurück.

Das heißt aber nicht, dass `QueryInterface` in irgendeiner Art Objekte instanziert, ein erstes Objekt muss in jedem Fall vor dem Aufruf von `QueryInterface` instanziert worden sein. Wenn die Funktion einen Schnittstellenzeiger zurückgibt, wird automatisch der Referenzzähler erhöht, wie folgender Blick in die Sourcen verdeutlicht.

```
function TClassFactory.QueryInterface(const iid: TIID;
                                      var obj): HResult;
begin
  if IsEqualIID(iid, IID_IUnknown) or
           IsEqualIID(iid, IID_IClassFactory) then begin
    Pointer(obj):= Self;
    AddRef;
    Result:= S_OK;
  end else begin
    Pointer(obj):= NIL;
    Result:= E_NOINTERFACE;
  end;
end;

function TClassFactory.AddRef: Longint;
begin
  Inc(FRefCount);
  Result:= FRefCount;
end;
```

Abschließend die wichtigsten Tipps zu Interfaces:

- Die Standard-Aufrufkonvention für Schnittstellen ist `register`. Schnittstellen, die man von verschiedenen Modulen gemeinsam benutzt, sollten die Methoden mit `stdcall` deklarieren. Dies gilt insbesondere, wenn diese Module in verschiedenen Programmiersprachen erstellt wurden.
- Wenn Sie die Funktion `QueryInterface()` mit einer Schnittstelle einsetzen, muss diese eine zugeordnete IID besitzen, mit dem Operator `as` ist dies nicht nötig.
- In jedem Interface ist unterstellt, dass die Basisklasse oder ein Vorfahr die Methoden von `IUnknown` zur Referenzzählung implementiert.
- Die einfachste Art, diese Methoden zu implementieren, besteht darin, die Klasse von `TInterfacedObject` in der Unit *System* abzuleiten.
- Wenn es sich beim Objekt um eine VCL-Komponente oder ein Steuerelement handelt, unterliegt dieses Objekt der Speicherverwaltung von `TComponent`. Mischen Sie also nicht VCL-Referenzverwaltung mit der Referenzzählung.
- Die Funktion `GUIDToString()` konvertiert ein GUID in einen String aus druckbaren Zeichen. Jedes GUID wird in einen eindeutigen String umgewandelt.
- Im Unterschied zu einer **abstrakten Klasse** enthält eine Schnittstelle überhaupt keine implementierte Methode, alle Methoden sind abstrakt. Abstrakte Klassen lassen sich verwenden, wenn Grundverhalten in den abgeleiteten Klassen vorhanden sein soll.

1.2.8 Delegation

Eine Möglichkeit der Wiederverwendung von Quelltext in Schnittstellen besteht darin, ein Objekt in die Schnittstelle aufzunehmen bzw. die Schnittstelle als Objekt in eine an-

dere einzufügen. Die VCL verwendet zu diesem Zweck Properties, die Objekttypen darstellen. Zur Unterstützung dieser Struktur für Schnittstellen stellt Object Pascal das Schlüsselwort `implements` bereit, mit dem die Implementierung einer Schnittstelle ganz oder teilweise einem untergeordneten Objekt übertragen werden kann.

So erreicht man eine zentralisierte Instanzierung mit einem Objekt. Dies nennt man Delegation. Es gibt aber ein Delegieren vom Typ einer Schnittstelle (Beispiel 1) und ein Delegieren vom Typ einer Klasse (Beispiel 2):

```
Type //Beispiel 1
IMyInterface = interface
  procedure P1;
  procedure P2;
end;
TMyClass = class(TObject, IMyInterface)
  FMyInterface: IMyInterface;
  property MyInterface: IMyInterface
              read FMyInterface implements IMyInterface;
end;
```

Die folgende Aggregation ist eine andere Möglichkeit, Code über diese Mechanismen, vor allem zur Laufzeit, wiederzuverwenden (wie beim Proxy). Bei der Aggregation enthält ein übergeordnetes, umgebendes Objekt ein untergeordnetes. Vorteil: Es lassen sich bestehende Klassen, die nicht von einem Interface abstammen, einbinden. Das untergeordnete Objekt implementiert Schnittstellen, die (nur) für dieses übergeordnete Objekt verfügbar, sprich aufrufbar, sind.

Hier ist also der Unterschied zwischen Delegation und Aggregation zu sehen. Bei der Aggregation sind nebst der Schnittstelle mindestens zwei Objekte im Spiel, sodass Objekt A an ein Objekt B delegiert. Die CLX enthält beispielsweise Klassen, welche die Aggregation unterstützen. Das Objekt gibt also vor (es delegiert eben), selbst Interfaces zu implementieren, die in der Tat von anderen Objekten stammen.

```
IMyInterface = interface //Beispiel 2
  procedure P1;
  procedure P2;
end;
TMyImplClass = class
  procedure P1;
  procedure P2;
end;
TMyClass = class(TInterfacedObject, IMyInterface)
  FMyImplClass: TMyImplClass;
  property MyImplClass: TMyImplClass
               read FMyImplClass implements IMyInterface;
```

Fazit: Schnittstellen bieten gegenüber abstrakten Klassen mehr verfügbare Merkmale an, sind aber auch restriktiver in der Handhabung. Ich erinnere daran, dass ein Interface

selbst kein Grundverhalten implementieren darf. Wenn ein Grundverhalten nicht nötig ist, steht dem Einsatz von Zwischengesichtern eigentlich nichts mehr im Wege.

Eine andere Technologie für verteilte Anwendungen ist CORBA. 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, sodass Parameterwerte und Rückgabewerte korrekt 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 Typinformationen enthalten).

Das komplette Beispiel läuft übrigens auch unter Kylix, wie Abb. 1.17 als Starter zeigt.[xiii] Wobei DLL+ dann zu SO+ (für Shared Objects Plus) mutiert. Warum sich aber die SO-Größe verdreifacht und welche anderen wundersamen Libraries sonst noch unter Linux geladen werden, steht dann in einem anderen noch nicht verfügbaren Kapitel ;).

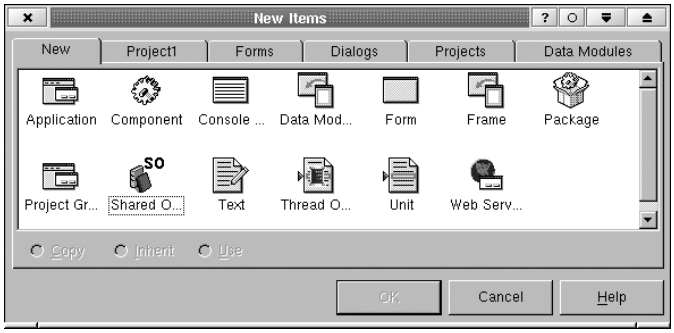

Abb. 1.17: Start einer SO als DLL+ in Kylix

1.2.9 Interface-Listen

Interfaces lassen sich auch in Listen zur Laufzeit verwalten, die `TInterfaceList` repräsentiert eine Liste mit Schnittstellen. Solche Listen sind von Vorteil, wenn die konkreten Objekte keine festen Namen besitzen oder erst zur Laufzeit instanzierbar sind. Mit der Methode `Add` lässt sich eine weitere Schnittstelle in die Liste aufnehmen. Mit den bekannten Items können Sie direkt auf eine Schnittstelle zugreifen, die in der Liste enthalten ist.

Sie merken, ich bereite den sanften Übergang zum nächsten großen Kapitel der Listen vor. Der Parameter-Index gibt die Position einer Schnittstelle in der Liste an.

Der Konstruktor erstellt eine Schnittstellenliste, die man mit den einzelnen Objekten füllt, welche die Schnittstellen unterstützen. Es ist dann möglich, mit einer Schnittstellenreferenz alle Objekte der Liste anzufragen, ob sie das Interface unterstützen, und entsprechend auszuführen.

Die Methode `QueryInterface`, die zur Laufzeit abfragt, ob das instanzierte Objekt die erforderliche Schnittstelle unterstützt, ist hier als Test gleichbedeutend wie der as-Operator und zeigt nochmals die Laufzeitabfrage. Wenn das Objekt die angeforderte

Schnittstelle unterstützt, wird es im Parameter `Obj`, in meinem Fall `aObjShout` zurückgegeben und `QueryInterface` liefert `S_OK`.

```
procedure TForm1.btnCollectClick(Sender: TObject);
var collection: TInterfaceList;
  i: byte;
  aObjspeak: ISpeak;
  aObjShout: IShout;
begin
  collection:= TinterfaceList.create;
  try
    with collection do begin
      add(TEnglish.create);
      add(TFrench.create);
      add(TTRex.create) ;
    end;
   for i:= 0 to collection.count - 1 do begin
      //TFrench, TEnglish
     aObjSpeak:= collection[i] as ISpeak;
     if aObjSpeak <> NIL then
       aObjSpeak.sayHello;
      //TFrench, TTRex
     collection[i].queryInterface(IShout, aObjShout);
     if aObjShout <> NIL then
     aObjShout.shout;
   end;
  finally
   collection.free;
  end;
end;
```

Auch in der CLX sind diese Listen gefragt, wie ein Blick auf den Mechanismus der Verwaltung von Scripten auf dem Webserver Framework zeigt, wo der Producer selbst die einzelnen Scripte und ihre Engine als Methoden vom Typ `IScriptProducer` verwaltet:

```
constructor TBaseScriptSite.Create(AProducer: IScriptProducer);
var
  I: Integer;
begin
  inherited Create(AProducer);
  FNamedList:= TStringList.Create;
  FIntfList:= TInterfaceList.Create;
  for I:= 0 to ScriptObjectFactories.Count - 1 do
    ScriptObjectFactories[I].AddGlobalObjects(Self);
end;
```

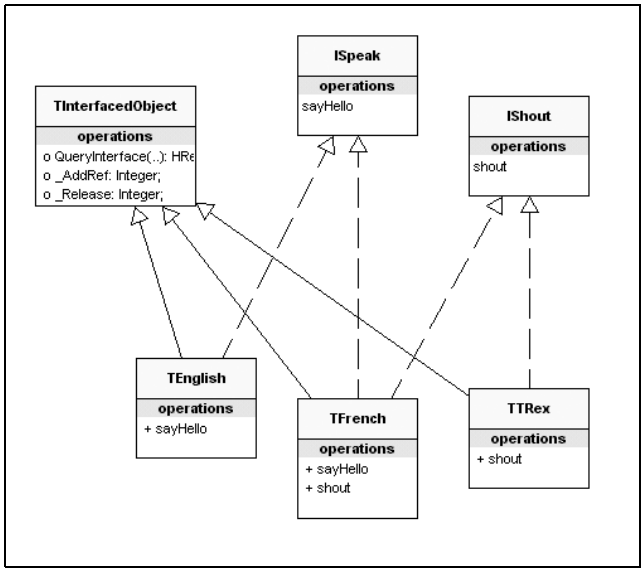

Abb. 1.18: Eine Liste voller Schnittstellen

1.3 Listen und Kollektionen

In diesem Schritt der Pattern-Feinheiten geht es um die eigentlichen Arbeitspferde innerhalb der Klassen, die Rede ist von Listen, welche in den Patterns rege zum Einsatz kommen. Obwohl Listen als Datenstruktur bestens bekannt sind, haftet ihnen doch der Hauch von Unbekanntem an.

Eine Liste, konkret die Klasse TList, speichert Zeiger in einem Array-Format. Wie Listen sind auch Streams oder Kollektionen in der BaseCLX zu finden und sind im ganzen Framework tüchtig in Gebrauch. Rund 190 Units benötigen die Dienste einer oder mehrerer Listen.

> *Die BaseCLX enthält eine Reihe von Units, welche die Grundstrukturen für einen Großteil der Komponentenbibliothek bereitstellen. Diese Units beinhalten globale Routinen, in der CLX-Laufzeitbibliothek vorzufinden, eine Reihe nützlicher Hilfsklassen, wie die Klassen zur Darstellung von Streams und eben Listen, sowie die Klassen TObject, TPersistent und TComponent.*

Diese Units fasst man unter der Bezeichnung BaseCLX zusammen. Die BaseCLX enthält keine der Komponenten, die in der Komponentenpalette sichtbar sind, werden aber von den Komponenten benutzt, die in der Komponentenpalette enthalten sind, und stehen auch zur Anwendungsentwicklung und zur Erstellung eigener Klassen zurVerfügung.

BaseCLX enthält viele Klassen, die Listen oder Kollektionen (auch Sammlungen genannt) von Elementen darstellen. Sie unterscheiden sich durch die Art ihrer Inhalte und ihres Funktionsumfangs. Zudem sind einige von ihnen persistent, andere nicht. Persisten-

te Listen sind in einer Formdatei speicherbar. Deshalb werden sie oft als published deklarierte Eigenschaft für Komponenten verwendet.

Persistente Listenelemente lassen sich zur Entwurfszeit hinzufügen. Sie sind mit dem Objekt speicherfähig und somit verfügbar, wenn man die Komponente, die sie verwendet, zur Laufzeit in den Speicher lädt. Es gibt zwei Typen von persistenten Listen:

- Stringlisten (TStringlist, THashedStringlist)
- Kollektionen (TCollection etc.)

Damit es nicht immer eine TList oder TStringList sein muss, gibt es weitere Arbeitspferde in der umfangreichen CLX Library, die bei strukturintensiven Anwendungen auf ihren Einsatz warten:

Listentyp	**Beschreibung**
TList	Die universelle Zeigerliste aus Vektoren
TThreadList	Eine thread-sichere Zeigerliste
TBucketList	Eine Zeigerliste, als Hash-Liste organisiert
TObjectBucketList	Liste von Objektinstanzen, als Hash-Liste organisiert
TObjectList	Eine speicherverwaltete Liste von Objektinstanzen
TComponentList	Eine speicherverwaltete Liste von Komponenten
TClassList	Eine Liste von Klassentypen
TInterfaceList	Eine Liste mit Schnittstellenzeigern
TQueue	Eine FIFO-Liste von Zeigern (First in, First out)
TStack	Eine LIFO-Liste von Zeigern (Last in, First out)
TObjectQueue	Eine FIFO-Liste von Objekten
TObjectStack	Eine LIFO-Liste von Objekten
TCollection	Basisklasse für spezielle Klassen mit typisierten Elementen
TStringList	Unsere bekannte Stringliste
THashedStringList	Eine Stringliste, als Hash-Liste organisiert

Tab. 1.5: Listentypen aus der CLX Library

Die meisten dieser Listen befinden sich in der Unit *Contnrs.pas* und deuten klar an, einen Namensraum für Containerklassen zu nutzen. Da die meisten zudem ein Subclassing von TList sind, gehen wir hier näher darauf ein. Da die TList ein dynamisches Array mit Zeigern speichert, sind auch die dazugehörigen Grundfunktionen eines flexiblen Arrays vorhanden.

1.3.1 Trans Europa Express

In der Praxis macht dieser Workaholic als TEE sicher 96 % aller zu Fuß aufgebauten verketteten Listen überflüssig.

Das von `TList` überwachte Array `FList` von `PPointerList` belegt seinen Speicher via GetMem und enthält seinerseits Zeiger, demzufolge die Zahl der Elemente auf SizeOf(Integer) div 16, d. h. rund 135 Millionen (134217727) Elemente begrenzt ist. `MaxInt` repräsentiert den höchsten Wert, der im Bereich des Datentyps Integer zulässig ist, rund 2.2 Milliarden (2147483647).

```
MaxListSize = Maxint div 16;
  TPointerList = array[0..MaxListSize - 1] of Pointer;
```

Die Allozierung der Daten erfolgt in 64-Byte-Blöcken auf dem Heap, welche die zugehörigen Methoden der Liste möglichst ökonomisch bearbeiten. Oft speichert man eine Liste von Objekten, die mit den Eigenschaften und Methoden in TList zu folgenden Manipulationen führen:

- Objekte zur Liste hinzufügen oder daraus entfernen
- Objekte in der Liste neu anordnen
- Objekte in der Liste finden und darauf zugreifen
- Objekte in der Liste sortieren
- Objekte in der Liste aufrufen

Die Sortiermethode implementiert den Quicksort, welcher an Geschwindigkeit kaum zu überbieten ist. Somit sollte eine meiner Lieblingsfehlermeldungen „Ein Eintrag wurde gefunden, möchten Sie sortieren?“ auch kein Problem mehr darstellen. Die Liste selbst besitzt einen Member `List` für den direkten Zugriff oder ein typisiertes Array-Property `Items` mit den Zugriffsmethoden get und put.

Für eine Liste mit Zeigern auf z. B. Records des Typs `TxyzRecord` soll also eine Klasse `TxyzList` bereitstehen, deren add- und get-Methoden einen Zeigertyp `PxyzRecord` bearbeiten kann.

Die allgemeine Eigenschaft `List` gibt ein Zeiger-Array an, aus dem die Items bestehen, von folgendem Typ:

```
  TPointerList = array[0..MaxListSize-1] of Pointer;
  PPointerList = ^TPointerList;

property List: PPointerList read FList;
```

Ein Problem ist allen Listen gemeinsam, nämlich dass ein Zerstören der Liste nicht automatisch die zugehörigen Elemente, meistens Objekte, mitbefreit.

> *Destroy gibt den Speicher, auf den die Elemente der Liste zeigen, nicht automatisch mit den Nutzdaten frei.*

Eine Liste arbeitet nicht wie eine Komponente, welche z. B. mit dem Zerstören eines Forms alle ihre zugehörigen Formkomponenten auch freigibt. Es gibt aber mit

`TObjectList` Abhilfe, wie weiter unten beschrieben ist. Ein Freigeben der Elemente muss also im Destruktor der Liste vor sich gehen:

```
For i:= 0 to pred(myList.count) do
  TObject(myList[i]).Free;
  MyList.Clear;
```

Mit `Clear` muss man noch die Array-Items selbst aus der Liste leeren, und die Felder `Count` sowie `Capacity` setzt die Methode dann auf null und der gesamte vom Array belegte Speicher wird schlussendlich freigegeben, aber eben nicht die enthaltenen Elemente.

Mit dem Iterieren durch den Loop vom Ende der Liste zum Anfang erreichen wir die Sicherheit, dass der beim Löschen der Elemente veränderte Indexzeiger zu keinem „list index out of bounds“ führt, da ja gilt:

„Mit Delete lässt sich das Element an der bezeichneten Position aus der Liste löschen. Die Numerierung von Indizes beginnt bei null für das erste Element. Das zweite Element hat entsprechend den Index 1 und so fort. Sobald ein Element entfernt ist, rücken alle nachfolgenden Elemente in der Indexposition um eins nach oben, und der Wert von Count wird um eins verringert.“

> *Wenn Sie die Referenz auf ein Element entfernen wollen, ohne den Eintrag aus der Liste zu löschen und die Eigenschaft Count zu ändern, setzen Sie die Eigenschaft Items für den Index am besten auf NIL.*

Wenn Sie wirklich intensiv Listen bearbeiten (respektive ihre Anwendung) und das maximale an Performance und Effizienz erreichen möchten, lässt sich mit einem direkten Zugriff auf das PointerArray die Zugriffsmethode von `TList.Get` und die zugehörige Validierung umgehen, vorausgesetzt der Index liegt im gültigen Bereich, der dann Ihrer Verantwortung unterliegt, d. h., es erfolgt keine Bereichsüberprüfung mehr:

```
TempPointer:= myList.List^[index];
FStream.readBuffer(myList.List^,Fcount * sizeOf(longint));
```

Eine explizite Typumwandlung auf ein Element eines Records der Liste wie

```
PXYRecord(myList.List^[idx])^.recValue:= 2341;
```

kann aber auch versierten Hackern eine kleine Denkpause verschaffen. Eine weitere Optimierung besteht mit dem Design von Hochgeschwindigkeitslisten vom Typ ICESpeed ;), d. h., wenn das als Listenelement zu speichernde Datum genau 4 Bytes veranschlagt, kann ich diese Daten z. B. als Messdaten oder Prozessdaten direkt in der Liste, respektive im Listen-Array speichern!

Niemand hält einen davon ab, das Listen-Array selbst als Datenspeicher zu benutzen. Die Methode `Add` trägt den Integer-Messwert oder Prozesswert wie ein Zeiger brav ein und gibt keinen weiteren Laut bezüglich der Zweckentfremdung von sich.

Hier ist aber dann durch den eingebauten Typwandler des Codegenerators (es soll eine Zahl und kein Pointer zurückkommen) wiederum der Direktzugriff mit einem Typecasting vonnöten:

```
MyLongDat:= LongInt(myList.List^[dat]);
```

1.3.2 Design von Listen

Seit Delphi 5 hat nun Borland mit einer neuen Klasse `TObjectList` die Freigabe der Objekte in einer Liste vereinfacht. Mit dem Setzen der Eigenschaft `OwnsObjects` auf true (Voreinstellung) verwaltet `TObjectList` den Speicher seiner Objekte selbst, das heißt, die Liste gibt den Speicher des Objektes frei, wenn sein Index neu zugewiesen wird, sofern es mit der Methode Delete, Remove oder Clear aus der Liste entfernt wird oder wenn die Instanz von `TObjectList` selbst aufgelöst wird.

Wenn man also Elemente aus der Liste löscht oder die Liste selbst freigibt, werden die Objekte oder Komponenten automatisch aus dem Speicher entfernt. Seltsamerweise geschieht das via Notify-Methode, da man die Klasse `TList` verändern musste, um den fast einzigen Nachfolger `TObjectList` auszubauen, ein aus OO-Design-Sicht fragwürdiges Superclassing-Vorgehen:

```
procedure TObjectList.Notify(Ptr: Pointer; Action:
                                  TListNotification);
begin
  if OwnsObjects then
    if Action = lnDeleted then
      TObject(Ptr).Free;
  inherited Notify(Ptr, Action);
end;
```

Auf der anderen Seite kann man sich als Entwickler der Klassenbibliothek die Freiheit in Borland nehmen, solange man keine Kompatibilitätsprobleme verursacht. „Stepwise Refinement im Zuge von Iterationen“, wie Beat Straehl meint.

Besser wäre eine Delegation mithilfe einer Wrapper-Klasse gewesen.[xiv] Man hätte dann allerdings den Nachteil, die Typensignatur (Interface) von `TList` noch einmal für `TObjectList` zu implementieren.[7]

Hier setze ich nun mit der Frage an, ob beim Einsatz einer Liste ein Subclassing (Vererbung) oder ein Delegieren mittels Aggregation oder Komposition von Vorteil ist. Vor der Implementierung der verschiedenen Listentypen galt es im R&D-Team von Borland, eine wichtige Entscheidung zur Datenstruktur zu treffen.

Für jeden Listentyp ließe sich eine eigene Klasse anlegen oder eine gemeinsame Klasse übernimmt die Grundfunktionen.

[7] There is no such Thing like a free Lunch.

Da die Speicherung polymorpher Strukturen, d. h. persistentes Speichern der Daten in einer Datei, von der CLX nicht immer unterstützt wird, muss das Team auch hier einen Kompromiss finden.

Obwohl sich die Listenklassen sowohl hinsichtlich ihrer Elemente als auch ihrer Abstammung unterscheiden, enthalten sie doch eine Reihe gemeinsamer Methoden zum Hinzufügen, Entfernen, Umsortieren und Abrufen der Listenelemente. Beim Erstellen einer eigenen Listenklasse oder eines Listenframeworks sind nun drei Möglichkeiten als Designgrundsätze in die weiteren Kopfarbeiten mit einzubeziehen, die auch sauber als Beispiele unterlegt sind:

- Einfache **Referenz** von TList in einer Klasse (Aggregation)
  Beispiel TClassFinder
- Eigene **Wrapper-**Klasse, welche TList umhüllt (Komposition)
  Beispiel TThreadList
- Eigenes **Subclassing** von TList (Einfache Vererbung)
  Beispiel TclassList

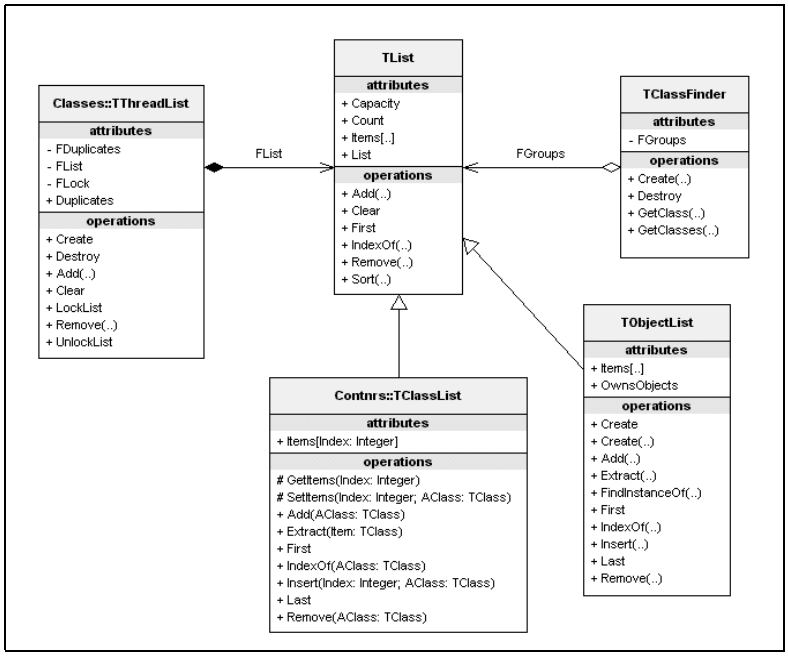

Abb. 1.19: Listendesign in der Übersicht

Aggregation ist eine modulare Möglichkeit zur Wiederverwendung von Quelltext durch Unterobjekte, welche die Funktionalität eines enthaltenen Objekts definieren, die Einzelheiten der Implementierung jedoch vor diesem Objekt verbergen. Wenn Sie also Ihre Liste wieder verwenden wollen, ist eine Aggregation vorzuziehen.

Steht hingegen ein eigenes Framework im Vordergrund, das mit den Unterklassen flexibel und skalierbar auf Änderungen reagieren muss, sollte man das Subclassing mit Vererbung vorziehen. Konkret: Beim Komponentenbau spielt die Vererbung die erste Geige, beim nachfolgenden Einsatz oder bei Integration der Komponente ist Aggregation oder Komposition gefragt, welche in den meisten Fällen der Vererbung vorzuziehen sind.

> *Eine Komposition hat dort ihren Nutzen, wo eine Applikation an verschiedenen Stellen auf eine zentrale Datenstruktur zugreifen muss.*

Eine separate Wrapper-Klasse als Komposition ist dann die beste Lösung, zumal sie auch in anderen Projekten zu Ehren kommt. Als Erstes gelange ich zur einfachen Referenz, welche die Listenklasse wie TList nur indirekt benötigt, sodass die Klasse selbst noch andere Aufgaben wahrnimmt. Die Klasse benötigt die Liste vor allem lokal, und auch die Zahl der Elemente hält sich im Rahmen. Hier der Entwurf mit direkter Referenz von TList in der Klasse `TClassFinder`:

```
TClassFinder = class
 private
   FGroups: TList;
 public
   constructor Create(AClass: TPersistentClass = nil;
                      AIncludeActiveGroups: Boolean = False);
   destructor Destroy; override;
   function GetClass(const AClassName: string):TPersistentClass;
   procedure GetClasses(Proc: TGetClass);
 end;
```

Das Wrappen einer Klasse bedeutet ein Umhüllen der Funktionen von der Listenklasse, die in der Wrapperklasse mit spezifischem Code angereichert die eigentlichen Methoden der aggregierten Klasse TList wiederum aufrufen oder eben delegieren, wie `TThreadList.Remove` als Nächstes verdeutlichen soll.

```
TThreadList = class
  private
    FList: TList;
    FLock: TRTLCriticalSection;
    FDuplicates: TDuplicates;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Add(Item: Pointer);
    procedure Clear;
    function LockList: TList;
    procedure Remove(Item: Pointer);
    procedure UnlockList;
    property Duplicates: TDuplicates read FDuplicates
```

```
                                        write FDuplicates;
  end;
```

Unter Windows ruft `LockList` die Windows-Funktion `EnterCriticalSection` zum Sperren der thread-sicheren Liste auf. Die Methode gibt das TList-Objekt zurück, in dem sich die Liste befindet. Bei Linux gibt LockList das TList-Objekt zurück, das die Liste enthält, nachdem Linux alle anderen fast gleichzeitigen Threads gesperrt hat.

```
procedure TThreadList.Remove(Item: Pointer);
begin
  LockList;
  try
    FList.Remove(Item);
  finally
    UnlockList;
  end;
end;
```

In der Klasse `TThreadList` wird die Liste vor einem Zugriff mit `LockList` gesperrt. Diese Methode gibt das Objekt `TList` mit den Listenelementen zurück, auf die man dann aus der Applikation wieder manipulierend zugreifen kann.

Als letzte Möglichkeit bietet sich das Subclassing an, welches eine maximale Flexibilisierung und Erweiterbarkeit in und während der Designzeit anbietet. Subclassing basiert auf einer Klassenstruktur im Gegensatz zum Delegieren (Aggregation), das auf einer Objektstruktur zur Laufzeit basiert.

Die meisten Methoden lassen sich durch schlichtes Vererben in der Unterklasse aufrufen, wie `TClassList.Remove()` zeigt. Es wäre aber völlig unkorrekt, direkt die Basisklasse in der CLX zu manipulieren (nur Borland darf), also ist Subclassing OO-politisch korrekt:

```
TClassList = class(TList)
 protected
   function GetItems(Index: Integer): TClass;
   procedure SetItems(Index: Integer; AClass: TClass);
 public
   function Add(AClass: TClass): Integer;
   function Extract(Item: TClass): TClass;
   function Remove(AClass: TClass): Integer;
   function IndexOf(AClass: TClass): Integer;
   function First: TClass;
   function Last: TClass;
   procedure Insert(Index: Integer; AClass: TClass);
   property Items[Index: Integer]: TClass read GetItems
                                   write SetItems; default;
  end;
```

```
function TClassList.Remove(AClass: TClass): Integer;
begin
  Result:= inherited Remove(AClass);
end;
```

Fachklassen mit Listen

Beim Modellieren mit Klassen (man kann auch von Business Patterns sprechen), die eine Kardinalität (Multiplicity) von 1..* oder 0..* besitzen, stellt sich in den Workshops vielfach die Frage, wie denn das Design mit Listen aussehen muss, wenn eine Klasse dann zur Laufzeit n-Objekte beinhaltet.

Die Aggregation kommt am häufigsten vor: Eine Teil-Klasse wie `TAccountList` gehört zu genau einem Aggregat wie `TCustomer`, und ein Aggregat kann aus einer oder vielen Teil-Objekten bestehen. Der zulässige Wertebereich reicht von 0 bis 32767, höhere Zahlen zeigen sich in einem Diagramm durch < *>, das in der UML vom ERD her ein <N> bedeutet.

Werfen wir einen Blick auf Abb. 1.20, so sehen wir bezüglich der Kardinalität, dass keine oder eine Klasse `TAccount` 1 oder N `TCustomer` Klassen besitzen können. Dies ist als Assoziation modelliert, da die Listen zur Laufzeit entstehen. Was bedeutet diese Beziehung nun konkret?

> *Ein oder mehrere Kunden können kein, ein oder mehrere Konten besitzen. Die 0..* auf der Seite von TAccount heißt, dass ein Kunde nicht Inhaber eines Kontos sein muss, aber Inhaber eines Kontos oder mehrerer Konten sein kann.*

Auf der anderen Seite bedeutet die 1..*, dass ein Konto aber **mindestens** einen Kunden besitzen muss, aber auch mehrere Kunden haben kann. Eigentlich ist das * zu viel des Guten, da es so keine Begrenzung für die Anzahl der Kontoinhaber gibt. Das hat Auswirkungen auf eine weitere Liste (Inhaberliste) in der Klasse `TAccount`, da ein und dasselbe Konto nun einer Gruppe von Kunden gehören kann, zu sehen an der Liste in der Klasse `TAccount`.

Technisch gesehen wird aber die TList aggregiert, da sie die Konteninhaber in der Liste aufnimmt und ich keine eigentliche Kundenliste implementieren will. Wichtig zu erkennen ist aber, dass ein Fachmodell nicht immer dem konkreten technischen Design in der Umsetzung mit Listen entsprechen muss. Das technische Modell entscheidet:

- Die Listenelemente sind als Records in der Datenbank
- Die Listenelemente sind als Objekte transient im Speicher
- Die Listenelemente sind als Objekte persistent (Blob) in der Datenbank

Wie soll auch das Fachmodell entscheiden, da die Art der Navigation und Speicherung von Beziehungsmengen zu diesem Zeitpunkt meist noch nicht definiert ist. Tools mit Codegeneration sollten aber so weit kommen, dass ein Symbol wie * automatisch veranlasst, diese Beziehungsmenge in einer Art Liste zu generieren.

Wenn ich nun sehen will, wer alles Inhaber eines bestimmten Kontos ist, lasse ich einen Count-Property laufen:

```
var acc: TAccount;
for i:= 0 to acc.accOwnerList.count -1 do
   memBank.lines.add(TCustomer(acc[i]).hisName);
```

Nachdem ich ein Kunden-Objekt erzeugt habe, ist dieser Kunde bei der Erstellung des Kontos zunächst noch alleiniger Inhaber, bis weitere Verwandte hinzukommen:

```
cust:= TCustomer.create('Fred', 777);
acc:= TAccount.create(cust, 2341167);
  acc.addCustomer(sohn);
  acc.addCustomer(schwiegersohn);
```

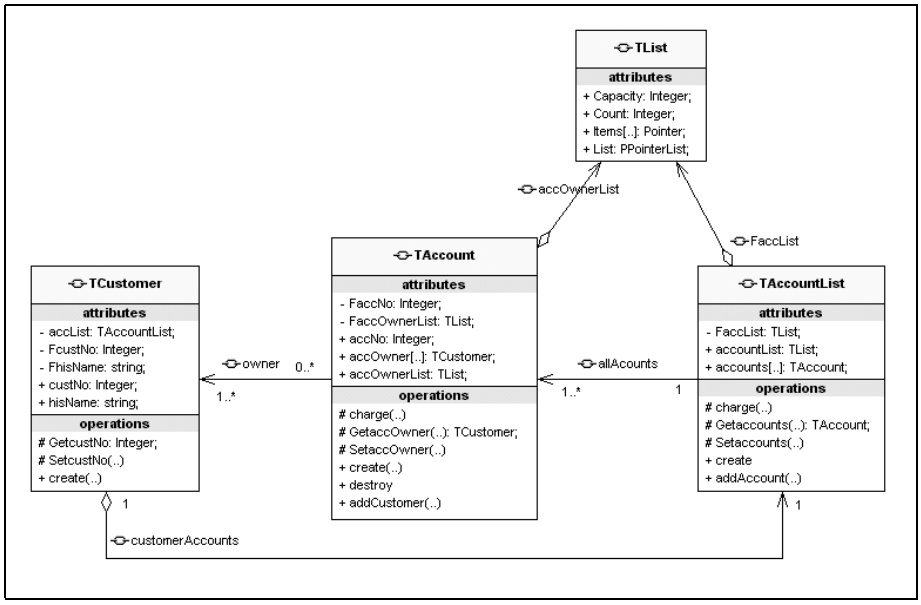

Abb. 1.20: Modellieren mit Listen in Klassen

1.3.3 Klassenreferenzen

Einige Patterns wie Abstract Factory, Broker oder der Builder enthalten auch Klassenreferenzen. In manchen Situationen führt man Operationen mit einer Klasse selbst und nicht mit ihren Instanzen (Objekten) durch. Dies geschieht fast automatisch, wenn Sie einen Konstruktor durch eine Klassenreferenz aufrufen. Denn im Moment des Konstruktoraufrufes ist man ja noch nicht im Besitze einer Instanz. Sie können dann auf eine bestimmte Klasse immer über ihren Namen zugreifen. Machmal müssen auch Variablen oder Parameter deklariert werden, die Klassen **als Werte** aufnehmen. Für diese Fälle benötigen Sie Klassenreferenztypen.

Klassenreferenztypen

Klassenreferenztypen ermöglichen die direkte Durchführung von Operationen auf Klassen, was einem „static“ entsprechen kann. Dies unterscheidet sie von Klassentypen, bei denen man Operationen auf den Instanzen von Klassen durchführt. Angenommen Sie ermöglichen eine Benutzerauswahl, indem der Benutzer je nach Wahl eine Instanz einer bestimmten Klasse erzeugen kann:

```
case userChoice of
  button1: object:= Tclass1.create
  button2: object:= Tclass2.create
```

Nachteil ist, dass man zwar jedes Objekt erzeugen kann, aber die Klasse selbst nicht in einer Variablen speicherbar und somit schon zu Designzeit fest verdrahtet ist. Der Vorteil einer Klassenreferenz gegenüber einer festen Klasse liegt wieder einmal in der Polymorphie. Das Problem lässt sich nun elegant lösen, indem man die Hierarchie eine Stufe höher anlegt:

```
Type
     TmyClassRef = class of Aclass;
var  choiceClass: TmyClassRef;
     object: Aclass
     object:= choiceClass.create

case userChoice of
  button1: choiceClass:= object;
  button2: choiceClass:= object2;
  button3: choiceClass:= object3;
```

Die Wahl des Benutzers ist jetzt also als Klassenreferenz gespeichert und die zugewiesenen Klassen müssen nur noch von `Aclass` abstammen, sodass zur Laufzeit mehr Flexibilität unter den verschiedenen Klassen möglich ist. Klassenreferenztypen werden manchmal auch als Metaklassen oder Metaklassentypen bezeichnet und deklariert man auf diese Weise:

```
class reference type
type TClass = class of TObject;
var AnyObj: TClass;
```

Klassenreferenztypen sind in folgenden Zusammenhängen vorteilhaft:

- Zusammen mit einem virtuellen Konstruktor zum Erzeugen eines Objekts, dessen Typ zum Zeitpunkt der Compilierung noch nicht bekannt ist.
- Zusammen mit einer Klassenmethode zum Durchführen einer Operation auf einer Klasse, deren Typ zum Zeitpunkt der Compilierung noch nicht bekannt ist.
- Als Operand auf der rechten Seite eines is-Operators zum Durchführen einer dynamischen Typüberprüfung eines Typs, der zum Zeitpunkt der Compilierung noch nicht bekannt ist.

- Als Operand auf der rechten Seite eines as-Operators zum Durchführen einer überprüften Typumwandlung eines Typs, der zum Zeitpunkt der Compilierung noch nicht bekannt ist.

1.3.4 Jäger und Sammlungen

Bei den Kollektionen will ich eine weitere Pattern-Technik vorstellen. Sammlungen (Collections) sind von der Klasse `TCollection` abgeleitet und persistent. Persistente Listen mit ihren Daten lassen sich in einer Formdatei oder entsprechenden Datei speichern. Die Daten sind dann verfügbar, wenn die Komponente, die sie verwendet, die Objekte zur Laufzeit in den Speicher lädt. Die Kollektion kümmert sich um das Speichern und Laden der Elemente (Items), ohne eine Zeile Save/Load schreiben zu müssen.

Diese Listen mit Designtime-Support kennen wir alle, denn jedes Property im Object Inspector, welches eine editierbare Liste von Elementen zeigt, ist eine Ableitung von `TCollection`. Collections kann man von der Technik am ehesten mit einer `TObjectList` vergleichen, denn es ist eine Liste mit weiteren Klassen. Wobei ein Collection-Item wieder Subcollections enthalten kann, das bei Verschachtelung zu Unübersichtlichkeit führt.

Im Klassendiagramm in Abb. 1.21 zeigt die Technik der doppelten Aggregation, dass der qualifizierte Bezeichner `Index` auf die Elemente von `TCollectionItem` zugreift, diese Klasse wiederum einen Zugriff, via der Rolle <access>, auf `TCollection` haben kann und somit eine weitere Verschachtelung herbeiführt. `TCollectionItemClass` ist nur eine Klassenreferenz (wie oben besprochen), welche die Konstruktoren zur Laufzeit wechseln kann:

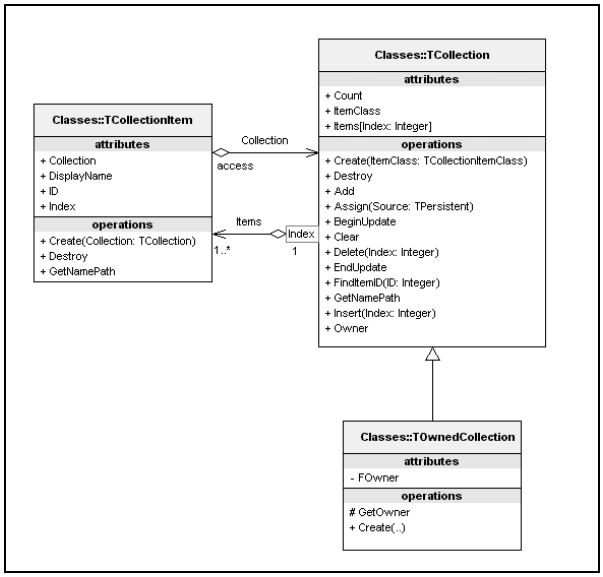

Abb. 1.21: Das Paket einer Kollektion

```
TCollectionItemClass = class of TCollectionItem;
TCollectionNotification = (cnAdded, cnExtracting, cnDeleting);
```

Es gibt zwei Typen von persistenten Listen: Stringlisten und eben Kollektionen.

Jeder Nachfolger von `TCollection` ist auf die Verwaltung einer speziellen Elementklasse spezialisiert, die von `TCollectionItem` abgeleitet ist. Kollektionen unterstützen viele Gemeinsamkeiten der bekannten Listenklassen.

Alle Sammlungen sind als published deklarierte Eigenschaften geeignet, und viele sind nicht unabhängig von dem Objekt funktionsfähig, das sie zur Implementierung einer ihrer Eigenschaften benutzt. Zur Entwurfszeit steht für die Eigenschaft, deren Wert eine Sammlung ist, ein Editor zur Verfügung, mit dem man Elemente hinzufügen, löschen oder neu anordnen kann. Der Editor stellt eine allgemeine Benutzerschnittstelle zum Bearbeiten von Sammlungen bereit.

Jede TCollection-Komponente enthält eine Gruppe von TCollectionItem-Nachkommen. `TCollection` verwaltet einen Index der Kollektionselemente im Array `Items`. Die Eigenschaft `Count` enthält die Zahl der Elemente in der Kollektion. Mit den Methoden `Add` und `Delete` können Sie der Kollektion Elemente hinzufügen und Elemente aus ihr löschen.

Von `TCollection` abgeleitete Objekte können Objekte enthalten, die Nachkommen von `TCollectionItem` sind. Ein TDBGridColumns-Objekt enthält beispielsweise TColumn-Objekte. Beide Klassen werden von `TDBGrid` bei der Anzeige von Gitterspalten verwendet.

Der Bau einer Kollektion kann nach folgendem Strickmuster ablaufen:

Als Erstes eine Ableitung von `TOwnedCollection` konstruieren, indem die neue Klasse die Methode `GetOwner` und auch den Konstruktor überschreibt. Oder man setzt direkt `TOwnedCollection` ein.

Dann sollte eine Ableitung von `TCollectionItem` ins Spiel kommen, welche die eigenen Daten und Properties aufnehmen kann. Kollektionen sind vor allem im Komponentenbau gefragt, einfache Kollektionen mit einer einfachen Ebene von Items gibt es aber des Öfteren.

> *Sie müssen der Kollektion in der Regel nicht mitteilen, welche Art von Daten sie aufnimmt – jeder Eintrag ist ein Zeiger.*

Ein konkretes Beispiel für ein Collection Framework finden Sie in der Unit *ObjBrKr* aus der ObjektBroker-Technologie, wo sich die einzelnen Server konfigurieren lassen.

```
constructor TCollection.Create(ItemClass: TCollectionItemClass);
begin
  FItemClass:= ItemClass;
  FItems:= TList.Create;
  NotifyDesigner(Self, Self, opInsert);
end;

TServerCollection = class(TOwnedCollection)
 private
```

```
   function GetItem(Index: Integer): TServerItem;
   procedure SetItem(Index: Integer; Value: TServerItem);
 public
   constructor Create(AOwner: TComponent);
   function GetBalancedName: string;
   function GetNextName: string;
   function FindServer(const ComputerName: string): TServerItem;
   property Items[Index: Integer]: TServerItem read GetItem
                                   write SetItem; default;
end;
```

`Notify` wird automatisch aufgerufen, wenn sich die Elemente in der Kollektion ändern. `Item` ist das Element, das gerade in die Kollektion eingefügt wurde bzw. aus ihr entfernt werden soll. Gemäß der Implementierung in `TCollection` ruft `Notify` die Methode `Added` auf, wenn `Action` den Wert `cnAdded` besitzt, und `Deleting`, wenn `Action` den Wert `cnDeleting` besitzt. Der Wert `cnExtracting` wird von `TCollection` ignoriert. `Notify` lässt sich in abgeleiteten Klassen überschreiben.

```
constructor TServerCollection.Create(AOwner: TComponent);
begin
//damit das Streaming funktioniert
  inherited Create(AOwner, TServerItem);
end;
```

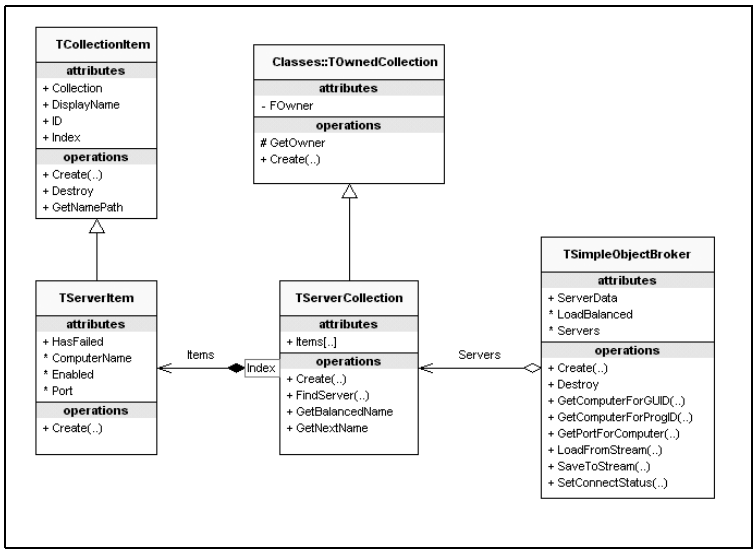

Abb. 1.22: Eine Kollektion im Einsatz

1.3.5 Schnelle Listen mit Hashing

Als letzten interessanten Tipp in der Listenverwaltung komme ich auf die dimensionierte Klasse `TBucketList` oder `THashedStringList` zu sprechen, die beide einen Hash-Mechanismus implementieren. Was soll aber ein Hash mit Pattern zu tun haben?

Beim Iterator, Interpreter oder innerhalb des Composite sind Hash-Listen eine mögliche Technik, den Direktzugriff auf eine Struktur zu optimieren.

Eine Hash-Tabelle beschleunigt die Suche nach Daten, da im Namen der Referenz bereits der Speicherort codiert ist. Verwenden Sie `TBucketList` als einfache Hash-Tabelle zum Speichern von Datenzeigern, die ihrerseits über Zeiger referenziert werden. Wie funktioniert nun eine Hash-Funktion: Man stelle sich eine Tabelle mit 80 Einträgen vor, speichern wollen wir den Begriff „CODESIGN for All“, der z. B. einen CD-ROM-Titel repräsentiert.

Eine Hash-Funktion erwartet nun einen so genannten Key (unseren CD-ROM-Titel als String) als Input mit dem Resultat, einen Zeiger auf den Ort der Daten zu erhalten. Schneller kann eine Suche kaum gehen, da im CD-ROM-Titel schon der Speicherort codiert enthalten ist. Hier macht jeder Hash intensiv Gebrauch vom folgenden MOD-Operator.

Angenommen wir codieren den gewählten Titel „CODESIGN for All“ durch die Quersumme der einzelnen ASCII-Werte und erhalten somit 1260. Mit dieser Nummer rechnen wir 1260 MOD 80 und erhalten, gemäß MOD-Operator, immer einen Wert zwischen 0 und 80, konkret in meinem Fall 60. Also hat der zugewiesene symbolische Speicherort eines Mediums die Adresse 60!

Die einfache Hash-Funktion im Beispiel ist schnell implementiert:

```
for I:= 1 to Length(edit1.Text) do
  hashOf:= hashOf + Ord(edit1.text[I]);
  edtresult.text:= intToStr(hashOf MOD 80);
```

Eine richtige Hash-Funktion jedoch ist komplizierter, da mit meiner simplen Quersumme die Gefahr von Kollisionen entsteht, will heißen, ein Wert kann durch die Hash-Funktion doppelt auftreten. Somit sollte der Wert, der durch den Key ermittelt wird, so weit als möglich eindeutig sein, da es nicht zwei gleiche Orte (Adressen) geben darf. Auch die echten Hash-Funktionen sind nicht ganz kollisionsfrei, nur besitzen sie einen Mechanismus zur Bereinigung des Konflikts, indem einfach die nächste freie Adresse zum Zuge kommt. Den Hash-Wert erhalte ich wie folgt:

```
Hash:= HashOf(Key) mod Cardinal(Length(Buckets));
```

Diesen Wert erzeugt die Funktion `HashOf()`, er gibt beim Suchen durch die Key-Eingabe (irgendein Titel) mit dem MOD-Operator wieder den Ort zurück, respektive den Zeiger, Block oder Index, je nach physischer Datenstruktur. HashOf entspricht in meinem Beispiel der Quersumme, also der Codierung eines Namens oder Titels. Die zentralen Hash-Funktionen sind in der Unit *IniFiles.pas* zu finden.

```
function TStringHash.HashOf(const Key: string): Cardinal;
var I: Integer;
begin
  Result := 0;
  for I := 1 to Length(Key) do
    Result:= ((Result shl 2) or (Result shr (SizeOf(Result)
                            * 8 - 2))) xor Ord(Key[I]);
end;
```

Bei der Stringliste mit Hash-Funktion wird intern ein `TMemIniFile` verwendet, um die Strings in einer INI-Datei zu verwalten. Das Objekt kann man jedoch wie jede andere Stringliste nutzen. Insbesondere bei Listen mit einer großen Anzahl von breiten Strings kann ein Suchsystem die Leistung durch die Verwendung der Klasse `THashedStringList` anstelle von `TStringList` enorm optimieren.

```
function TStringHash.Find(const Key: string): PPHashItem;
var Hash: Integer;
begin
  Hash:= HashOf(Key) mod Cardinal(Length(Buckets));
  Result:= @Buckets[Hash];
  while Result^ <> nil do begin
    if Result^.Key = Key then
      Exit
    else
      Result:= @Result^.Next;
  end;
end;
```

Das Suchen (und Finden) in der Hash-Liste funktioniert mit der gleichen Funktion `HashOf()`. Ich gebe wieder den Key ein und erhalte eine Referenz auf den Ort zurück, der vorgängig mit der Hash-Funktion ermittelt und belegt wurde.

`TBucketList` umfasst Methoden zum Hinzufügen, Entfernen und Suchen von Listenelementen. Bucket heißt so viel wie Eimer oder Kübel. Außerdem steht eine Methode zur Verfügung, die für jedes Listenelement eine Callback-Funktion ausführt. Und diese Callback funktioniert über eine bekannte ForEach-Methode.

Wer seine Sporen noch unter Borland Pascal oder neuerdings mit .NET in Delphi verdient, kennt den „ForEach-Iterator“, der eine Art Schleife in einer Kollektion durchführt. Ein Iterator lässt sich genauer definieren:

> *Jede ForEach-Methode hat einen Zeiger auf eine Prozedur als Parameter, die in der Reihenfolge, in der die Objekte oder Elemente gespeichert sind, aufgerufen wird.*

Sie bestimmen also, was konkret mit den Elementen geschehen soll, deshalb definieren Sie eine Prozedur, die man dem Iterator als Prozedurparameter übergibt. Das ist dann von riesigem Vorteil, wenn Standardmethoden wie Suchen oder Sortieren nicht ausreichen.

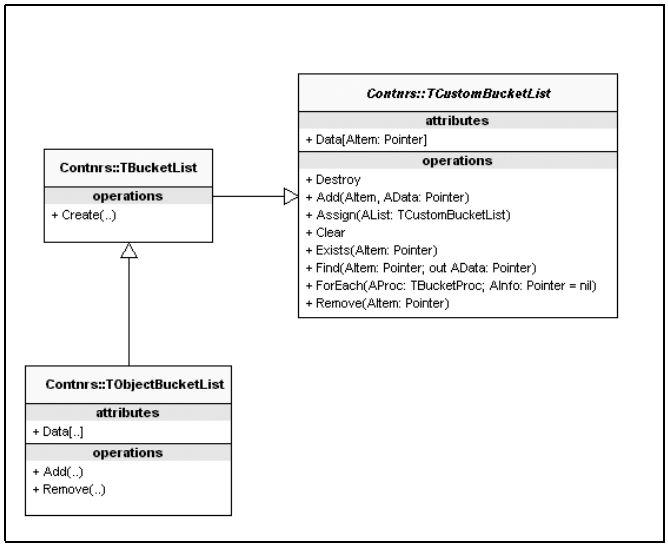

Abb. 1.23: Eine Hash-Liste mit Iterator

Angenommen Sie wollen jedes Element in der Liste im gleichen Zug durch einen Iterator drucken lassen, d. h., der Iterator durchforstet die Liste und ruft jedesmal ihre Printroutine auf, deshalb auch Callback genannt:

```
MyCollection.ForEach(@callPrint(items[I]));
```

Mit `ForEach` kann man nun die in `AProc` angegebene Prozedur für jedes Element in der Liste ausführen lassen. Die Methode durchläuft die gesamte Liste in einer Schleife und übergibt jedes Element zusammen mit seinen Daten an die Callback-Prozedur.

```
function ForEach(AProc: TBucketProc; AInfo:Pointer=NIL):Boolean;
```

`AProc` ist die Callback-Prozedur, die ausgeführt werden soll. `AInfo` hat keine vordefinierte Bedeutung, so sei er NIL. Es wird einfach an den Parameter `AInfo` der Callback-Prozedur übergeben.

`ForEach` gibt `true` zurück, wenn die Callback-Prozedur für jedes Element der Liste aufgerufen wurde. Der Rückgabewert `false` bedeutet, dass die Callback-Prozedur für ein Element in `AContinue` den Wert false zurückgegeben hat und die Bearbeitung der restlichen Elemente abgebrochen wurde.

Beim Design Pattern Composite streife ich das Gebiet nochmals. Typisch ist die Callback-Prozedur, die in der Regel als eigene Unterprozedur konstruiert ist, damit der übergebene Zeiger lokal bleibt. Es lassen sich auch eigene Iteratoren bauen, die nicht in einer Kollektion enthalten sein müssen, wie abschließender Blick in die Quellen zeigt:

```
procedure TDockTree.ForEachAt(Zone: TDockZone; Proc:
                                      TForEachZoneProc);
  procedure DoForEach(Zone: TDockZone);
```

```
  begin
    Proc(Zone);
    if Zone.FChildZones <> nil then
                    DoForEach(Zone.FChildZones);
  end;

begin
  if Zone = nil then Zone:= FTopZone;
  DoForEach(Zone);
end;
```

1.3.6 Konstruktor und Destruktor

Schlussendlich streifen wir noch die Erzeuger und Zerstörer, da Design Patterns eigene Erzeugungsmuster aufweisen. Am korrektesten ist wohl, dass ein Objekt eine Instanz einer Klasse ist. Bei n-Instanzen, die auch ohne Namen in einer Liste daherkommen, kann man von Exemplaren einer Klasse sprechen.

Die Membervariablen wie `btnClose: TButton` sind nur deklariert, aber nicht alloziert. Erst im Zuge des Konstruktors erhalten auch die Member ihren **Speicherplatz** im System des Arbeitsspeichers zugewiesen.

Konstruktoren

Konstruktoren werden benutzt, um neue Objekte zu erzeugen und zu initialisieren. Dazu muss man den Konstruktor über den Bezeichner der Klasse (statt der Instanz) aktivieren. Wenn der Aufruf einen Konstruktor über den Bezeichner der Klasse (Klassenreferenz) aktiviert, geschieht Folgendes:

1. Speicherplatz für das neue Objekt wird auf dem Heap zugewiesen.
2. Der zugewiesene Speicherbereich wird initialisiert. Alle ordinalen Werte werden auf null, alle Zeiger und Objektinstanzen auf NIL und alle String-Felder auf leer gesetzt. Der Compiler initialisiert die VMT und die Runtime Library (RL).
3. Die benutzerdefinierten Aktionen des Konstruktors werden ausgeführt.
4. Eine Referenz auf den neu allozierten und initialisierten Speicherbereich wird vom Konstruktor zurückgegeben.

Folgendermaßen können Sie eine Kontoklasse „Wertschriften“ erzeugen:

```
constructor TStockKonto.createKonto(cust_No: longInt);
  begin
    inherited createKonto(cust_No, STOCK_KTO);
    with datamodule1.dsOpen.dataSet do
      FDepotGebuehr:= fieldValues['FEE'];
  end;

stockKonto:= TStockKonto.createKonto(FCustNo);
```

Falls für eine Klasse bereits eine Instanz existiert, lässt sich der Konstruktor auch über eine Objektreferenz, d. h. den Namen der Instanz, aufrufen. In diesem Fall werden nur die benutzerdefinierten Anweisungen des Konstruktors ausgeführt, aber es wird kein neues Objekt im RAM erzeugt.

Der erste Befehl eines Konstruktors ist häufig der Aufruf eines ererbten Konstruktors, um die ererbten Felder des Objektes zu initialisieren (`inherited`). Danach initialisiert der Konstruktor die zusätzlichen Felder der Klasse.

Konstruktoren müssen die Standardaufrufkonvention `register` verwenden. Obwohl die Deklaration keinen Rückgabewert enthält, gibt ein Konstruktor immer einen Verweis auf das erstellte Objekt zurück, wenn er sich mit einer Klassenreferenz aufrufen lässt. Eine Klasse kann auch mehrere Konstruktoren haben. Sie können mehrere Routinen mit identischen Namen in demselben Gültigkeitsbereich deklarieren. Dieses Verfahren wird Überladen genannt. Überladene Routinen müssen mit der Direktive `overload` deklariert werden und unterschiedliche Parameterlisten (Signatur) haben. Beispiel:

Mit `Create` erstellen Sie ein TXMLDocument-Objekt zur Laufzeit. Komponenten, die Sie in der Entwurfsumgebung in Formulare oder Datenmodule einfügen, werden automatisch instanziert, es stehen aber zwei Konstruktoren zur Auswahl:

```
constructor Create(const AFileName: DOMString);
                              reintroduce; overload;
constructor Create(AOwner: TComponent);
                              overload; override;
```

Im Normalfall ist jedoch nur einer vorhanden. Konstruktoren heißen normalerweise eben `Create`.

Destruktoren

Destruktoren verwendet man, um Objekte aus dem Speicher zu entfernen. Wenn ein Destruktor aufgerufen wird, so lassen sich zuerst die benutzerdefinierten Aktionen ausführen, dann wird der dem Objekt zugewiesene Speicherbereich freigegeben. Die benutzerdefinierten Aktionen eines Destruktors sind normalerweise das Löschen von eingebetteten Objekten und das Freigeben von Ressourcen, die vom Objekt belegt wurden. Delphi kennt einen „Garbage Collector“ bei Klassen, die einem `Owner` angehören.

Bei den Schnittstellen ist eine andere Art Destruktor am Werk. In den meisten Implementierungen inkrementiert `AddRef` den Referenzzähler für die Schnittstelle und gibt den neuen Zählerstand zurück. Wenn der Aufrufer die Schnittstelle nicht mehr benötigt, wird die Methode `Release` aufgerufen, die den Referenzzähler dekrementiert. Sobald der Zähler den Wert Null erreicht, wird das Objekt automatisch freigegeben.

Tritt in einem mit einer Klassenreferenz aufgerufenen Konstruktors eine `Exception` auf, wird das unvollständige Objekt automatisch durch einen Aufruf des Destruktors `Destroy` freigegeben.

```
destructor TStockKonto.destroy;
  begin
    dataList.Free;
```

```
    Transakt_Obj.Free;
    ctr.stockKonto:= NIL;
  end;
```

Sichtbarkeit von Klassen

Jedes Objekt eines Klassentyps besitzt seine eigene Kopie der im Klassentyp deklarierten Felder, alle Objekte verwenden jedoch dieselben Methoden gemeinsam. Eigene Konstruktoren werden nicht vererbt, außer dem Standardkonstruktor von `TObject`. Konkret heißt das, eine Klasse, die nur mit statischen Methoden ausgestattet ist, muss man nicht instanzieren, um ihre Methoden zu nutzen.

> *Das gilt vor allem für Klassenmethoden, welche keine Instanz benötigen und direkt über den Klassennamen aufrufbar sind.*

Wenn man einen Bezeichner in einer Klassentyp-Deklaration vereinbart, erstreckt sich der Gültigkeitsbereich vom Ort der Deklaration bis zum Ende der Klassentyp-Definition sowie über alle Nachkommen des Klassentyps und die Blöcke sämtlicher Methodendeklarationen des Klassentyps hinweg.

Eigenschaften (Properties) sind ein weiteres zentrales Element von Objekten, mit denen Sie bereits vertraut sind. Eigenschaften sehen zwar wie Felder aus, sind aber eine natürliche Erweiterung von diesen.

Sowohl Felder wie Eigenschaften lassen sich dazu benutzen, die Sichtbarkeit eines Objekts auszudrücken. Aber während Felder Elemente sind, die sich direkt lesen und ändern lassen, verschaffen Eigenschaften (Properties) mehr Kontrolle über den Zugriff auf die privaten Felder.

Schutzebenen

Die Sichtbarkeit eines Bezeichners (in Klassen lassen sich Felder, Eigenschaften und Methoden deklarieren) wird durch das Sichtbarkeitsattribut des Komponentenabschnitts bestimmt, in dem der Bezeichner deklariert ist. Es gibt vier Sichtbarkeitsattribute:

`private, protected, puplic, published`.

Die Schutzebenen `private` und `public` sind allerdings nicht so strikt, wie man zuerst vermuten könnte: Sie gelten nur für den Zugriff aus einer anderen Unit (Modul) heraus. Werden verwandte Klassentypen in derselben Unit untergebracht, so können die Klassentypen gegenseitig auf ihre privaten Komponenten zugreifen, als wären sie wie in C++ als „friend“ deklariert.

Die Sichtbarkeit eines Klassenbezeichners, den man in einem `private`-Abschnitt deklariert, ist auf die Unit beschränkt, welche die Klassentypdeklaration enthält. Ein Zugriff ist über die Instanz in derselben Unit möglich. Das heißt, private-Klassenbezeichner verhalten sich in dem Modul mit der Klassentypdeklaration wie öffentliche Klassenbezeichner, außerhalb des Moduls (Unit) sind sie jedoch unbekannt und es ist kein Zugriff auf sie möglich.

Die Bezeichner von geschützten Klassen (protected) und deren Nachkommen sind nur sichtbar, wenn der Zugriff über einen Klassentyp erfolgt, der in der aktuellen Unit deklariert ist.

In allen anderen Fällen sind die Bezeichner geschützter Klassen oder Komponenten verborgen. Der Zugriff auf die geschützten Felder ist nur zur Implementierung von Methoden der Klasse und ihrer Nachkommen möglich. Deshalb werden Klassenkomponenten (Felder, Methoden, Ereignisse etc.) in der Regel mit `protected` deklariert, wenn sie ausschließlich bei der Implementierung von abgeleiteten Klassen verwendet werden sollen. Man bezeichnet dies auch als **Entwicklerschnittstelle**.

Komponentenbezeichner, die man in öffentlichen Abschnitten mit `public` deklariert, unterliegen hinsichtlich ihrer Sichtbarkeit keinen besonderen Einschränkungen. Die `Public`-Abschnitte kann man auch als **Laufzeitschnittstelle** oder Applikationsschnittstelle betrachten.

Die Sichtbarkeitsregeln für `published` (veröffentlichte) Felder entsprechen denen für öffentliche Felder. Der einzige Unterschied besteht darin, dass für die Felder und Eigenschaften von veröffentlichten Klassen die Typinformation zur Laufzeit erzeugt wird (veröffentlichte Felder lassen sich mit published deklarieren). Die Laufzeitinformationen ermöglichen einer Anwendung die dynamische Abfrage von Feldern und Eigenschaften eines Klassentyps, die andernfalls unbekannt wären.

Die CLX greift mithilfe der Typinformation zur **Laufzeit** auf die Werte der Eigenschaften einer Klasse oder Komponente zu, um Formulardateien zu speichern und zu laden. Außerdem verwendet die IDE diese Informationen, um die Liste der Komponenteneigenschaften festzulegen, die Delphi im Objekt-Inspektor anzeigt.

Es folgt die Übersicht des Zugriffs auf die Attribute aus einer anderen Unit gesehen:

	Zugriff auf		
Attribut	**Klassen in Unit**	**Vererbte Klasse**	**Klasse aus Unit**
Private	Ja	Nein	Nein
Protected	Ja	Ja	Nein
Public	Ja	Ja	Ja

Tab. 1.6: Sichtbarkeit im Überblick

1.4 UML 2.0 Exkurs

Die letzte freigegebene Version von UML betrifft den Release 1.4, den die OMG im Mai 2001 verabschiedete. Auf dieser Basis sind übrigens die Patterns in den Diagrammen notiert. Der Release 1.4 betraf vor allem Erweiterungen in der Geschäftsprozessmodellierung sowie Anpassungen an die OCL (Object Constraint Language), der ich ein eigenes Kapitel widme.

Seit dieser Zeit arbeitet man an der Version 2.0, die den modellgetriebenen Ansatz (MDA) in den Vordergrund stellen will. Ivar Jacobson[8] erwähnte in einem Seminar im Dezember '02 in Zürich den Begriff „Executable UML“, welcher angeblich einer der fünf Macrotrends im künftigen Softwarebau sein soll.

Präzise wie ein Gebet, dessen Code-Jünger bereits erste Proberunden im Orbit drehen, und unmissverständlich ist die Aussage von Ivar Jacobson:

Aus einer Bibliothek von Modellen lässt sich die Zielsprache wählen, die dann auf die entsprechende Plattform portiert und generiert wird.

So wird man wohl weniger programmieren und vermehrt einen Modellkatalog konsultieren müssen. Dies bedingt aber eine stärkere Formalisierung, welche als Gesamtziel des neuen Release hervorgeht. All diese Vorhaben in UML 2.0 laufen unter dem Begriff MDA (Model Driven Architecture) zusammen, welche die Modelle in ausführbaren Code transformieren will.

1.4.1 MDA

Dass ein Architekturstil nicht nur etwas mit dem Bau und dem Zusammenspiel von Komponenten zwischen den Schichten zu tun hat, sondern auch mit den Personen und Rollen in der Organisation, die solche Komponenten generieren, macht MDA als Technik wie auch als Prozess interessant.

Die bereits erwähnten Prozessmodelle als Prozess Pattern sind vor allem Use-Case-getrieben, auch XP setzt mit den User Stories die Anforderungen an den Anfang. Wenn aber die Anforderungen schon im Modellansatz bestehen, spricht wenig dagegen, diese modellgetriebene Architektur bei ähnlichen Anforderungen immer wieder einzusetzen. MDA gibt es bereits für existierende Software [xv], die sich mittels Reverse Engineering umwandeln lässt. Zumal MDA auf der höheren Abstraktion ja sprachunabhängig ist. Nun, was meint Model Driven Architecture im Alltag?

Jede Sprache mit der zugehörigen Entwicklungsumgebung ist von einer inneren Architektur von Subsystemen abhängig, z. B. dem Framework der GUI, der Struktur zwischen Interface und Implementierung oder den typischen Datenbankzugriff-Komponenten mit ihren Connection Strings.

> *Ein UML-Generator 2.0 sollte diese Architektur kennen, damit man sich in den Modellen nicht um diese technischen Details kümmern muss.*

Mein Paketdiagramm sollte also bereits wissen, ob ich z. B. dbExpress mit SOAP statt ADO mit COM einsetzen werde, wenn ich das Modell zum Generieren einsetzen will!

Um diese „Probleme“ bei der Generierung zu lösen, schlägt die OMG vor, ein „Platform Independent Model“ (PIM) durch einen MDA-Generator zu erzeugen. Ein PIM ist ein UML-Modell, in dem die technischen Details nicht ersichtlich sind. Somit befindet sich das Modell auf einer höheren Abstraktionsebene und ist unabhängig von technischen Fakten wie der bekannten Programmiersprache oder Datenbanktechnik.

[8] Er gilt als Vater der Use-Case-Modellierung.

Das PIM muss aber genug Details besitzen, um das generische Modell durch definierte Abbildungsregeln und Architektur-Muster (siehe Teil 3) mit einem PSM auf die Zielsprache abzubilden, d. h. innerhalb der Plattform mit zu generieren.

> *Ein PSM (Platform Specific Model) ist das spezifische und technisch abhängige Model, das die Architektur erzeugt.*

Wer sich jetzt fragt, wie die Abgrenzung zu Design Patterns zu setzen ist, der weiß aus eigener Erfahrung, dass ein Tool aus einem Pattern-Katalog direkt sprachspezifischen Code produziert. So etwas wie einen Design-Pattern-Generator für Geschäftsprozesse gibt es noch nicht.

> *Also positioniert sich die MDA einen Schritt vor dem Design und ermöglicht bereits in der Analyse, sprachunabhängige Klassendiagramme zu generieren.*

Kernidee der MDA ist also die schrittweise Verfeinerung von der Analyse zum Design, ausgehend von einer Modellierung der fachlichen Applikations-/Geschäftslogik, die völlig unabhängig von der technischen Implementierung und der umgebenden IT-Infrastruktur ist. So ein simples PIM-Modell könnte grob so aussehen:

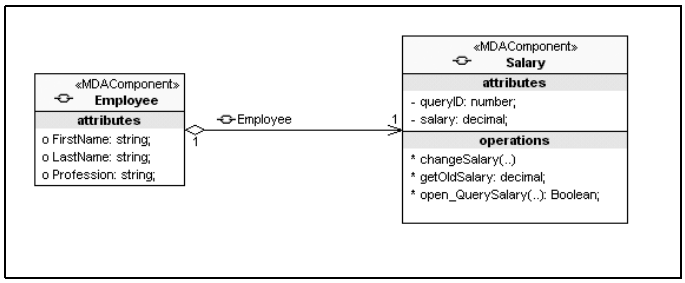

Abb. 1.24: Ein kleines MDA

Das kleine Modell soll das abstrakte Typensystem verdeutlichen, ähnlich einer IDL (Interface Definition Language), d. h., jeder mögliche Wert der `queryID` kann als Typ eine Zahl sein, z. B. integer, longint etc. Der neuralgische Punkt des MDA ist die exakte Definition einer Plattform.

Wenn nun das Tool aus dem PIM ein plattformspezifisches Modell (PSM) generiert, sollte Einigkeit über die zu implementierende Plattform herrschen. Ist nun eine Plattform CLX, .NET oder J2EE, dann sollten die entsprechenden Klassen und Typen zum Zuge kommen. Was aber ist, wenn die Plattform eine Makro-Sprache mit Objektmodell oder sogar eine Spezifikation wie CORBA ohne Applikationsserver hat?

> *Der PSM-UML-Generator muss also nicht nur die Zielsprache, sondern auch die Zielarchitektur kennen.*

Im Hinblick auf die Ausarbeitung solcher Details steht die MDA-Bewegung in Bezug auf die vorhandenen Tools noch ziemlich am Anfang.

Am 6. Januar 2003 wurde die überarbeitete Fassung der OMG übergeben. Am Ende dieses Jahres, da erfahrungsgemäß die Finalization Task Force neun Monate dauert, er-

wartet die Entwicklergemeinde eine Freigabe der Version 2, die sich in drei separate, aber zusammenhängende Gebiete aufteilt:

1. **UML Infrastructure**, welche eine Vereinfachung und Modularisierung des Metamodells vorsieht und keinen Einfluss auf den Benutzer hat. Dieses Gebiet ist für die Toolhersteller und Methodiker interessant, die an Profilen und möglichen Stereotypen Freude und Nutzen haben. Zudem wird das Metamodell von sprachabhängigen Konstrukten (wie C++ oder ADA) gesäubert und sprachneutral, d. h. in OCL, definiert.
2. **UML Superstructure**, mit den eigentlichen Erweiterungen bezüglich der Notation und Syntax und somit für den Benutzer sichtbar und verwertbar. Ziemlich aufwerten will die OMG die komponentenbasierte Entwicklung mit z. B. einer präziseren Definition einer Schnittstelle mit erweiterten Signalen oder Nachrichten. Einige statische und dynamische Elemente werden auf Echtzeitfähigkeit getrimmt.
3. **UML OCL**, die eine erhöhte Integration in das Metamodell und die Syntax vorsieht, sodass die Spezifikation durch ein Modell zum Code auch eine formale Sprache beinhaltet. Die OCL wird von einer Sprache der Bedingungen zu einer generellen Ausdruckssprache erweitert. Weiter folgt der verbesserte Modellaustausch durch die XMI (XML Metadata Interchange) z. B. mit Layout Informationen zwischen den einzelnen Tools.

Es ist klar, dass auf der einen Seite die Toolhersteller nun gefordert sind. Auf der anderen Seite wird die Vision des „Executable UML“ eine gravierende Änderung des Entwickleralltages zur Folge haben, sofern die Welt und schlussendlich auch unsere Kunden mit „standardisierter Individualsoftware“ und einheitlichen Geschäftsprozessen zufrieden sind. Etwas in dieser Art ist beim BizTalk Server von MS zu finden, der eine Generierung von Dokumenten-Workflow erlaubt.

Mehr noch, bezüglich der Wartung und Pflege einer Anwendung sind dann dieselben Fehler zu erwarten, da aus dem weltweiten Modellkatalog in den Codegeneratoren auch ähnliche Fehler entstehen können. Die wichtigsten Neuerungen (über 500 wurden im Sommer '02 der Revision Task Force überreicht) des Release 2 folgen nun in gestraffter Form auf einen Blick:

- Für das Metamodell entsteht ein Sprachkernel
- Zusätzliche Notationen bauen auf dem Kernel auf
- Support für den Einsatz von Komponenten untereinander
- Ausbau der Geschäftsprozessmodellierung (BPM)
- Interne Struktur für Bezeichner, Schnittstellen, Klassen, Typen und Rollen
- OCL-Integration als generelle Spezifikationssprache eines Modells
- State-Event-Generalisierungen und Echtzeiterweiterungen
- Erweiterungen für kompaktere und wiederverwendbare Sequenzdiagramme
- Architekturgetriebene Modellierung (MDA)
- Präziseres Abbilden der Notation zur Syntax

1.4.2 Codieren wie Architekten

Mit der Unterscheidung zwischen PIM und PSM definiert die MDA einen Prozess für die Entwicklung von Systemen in der Analyse-Phase. Durch Transformation lässt sich aus einem unabhängigen PIM ein spezifisches PSM erzeugen. Im PSM sind Informationen über die eingesetzte Softwareinfrastruktur enthalten, wie z. B. ein J2EE-Applikationsserver, ein dbExpress Provider oder ein .NET Framework.

Die Transformation kann ausgehend vom PIM über mehrere unterschiedlich abstrakte PSM hin bis zum Code erfolgen. In der MDA sind die notwendigen Schritte bei dieser Transformation definiert und durch den Einsatz von Tools halb oder vollständig automatisierbar.

Der Vorteil: Beim Austausch einer Middleware muss der Entwickler also nur die geeignete regelbasierte Transformation verwenden, um aus dem unveränderten PIM ein neues, passendes PSM zu generieren.

Zentral ist hier das Modellieren der Geschäftslogik und -anwendung ohne Details zu ihrer technischen Umsetzung. Darauf aufbauend werden technologiespezifische Modelle erstellt, die eine konkrete Umsetzung in einer gewählten Technologie, wie z. B. J2EE, CLX oder .NET beschreiben. Ähnlich wie z. B. in LabView oder MathLab in der Technischen Informatik ergibt eine grafische Simulation eines Regelkreises eine Steuerung für einen Klimaregler in einer Anlage.

> *Die Aufteilung in plattformunabhängige und plattformspezifische Modelle ermöglicht eine langfristige Anpassbarkeit an zukünftige Technologien, ähnlich dem Auswechseln des Datenbank-Provider.*

Auch das Wissen über die zugehörige Fachlogik soll einfließen. Für Domänen wie z. B. Finanz- oder Versicherungswesen, Telekommunikation und Medizinaltechnik sollen die typischen abstrakten Prozesse schon vordefiniert sein und für die Erstellung der PIM als so genannte „domain-specific core models“ bereitgestellt werden. Für den Austausch der Modellinformationen über Toolgrenzen hinweg wird XML Metadata Interchange (XMI) eingesetzt. UML und XMI sind ebenfalls Standards der OMG.

1.4.3 Der Turmbau zu UML

Der MDA-Ansatz birgt zusätzliches Potenzial für die Wiederverwendbarkeit von Modellen, da diese ja plattformunabhängig sind. Durch den Einsatz von standardisierter Transformation ist auch die Qualitätssicherung erhöht. Nach OMG (Object Management Group) sind dies die Hauptvorteile von MDA:

- Reduzierte Kosten bei der Entwicklung
- Erhöhte Qualitätssicherung durch Selbstähnlichkeit
- Schnellere Integration neuer Technologien

Auch zum Modellaustausch hat die OMG das entsprechende Konzept, XMI, schon als Standard verabschiedet. Der Name zeigt es bereits deutlich: XMI ist ein Mitglied der XML-Sprachfamilie. Die XMI-Norm definiert für das UML-Metamodell eine XML Document Type Definition (DTD) oder ein Schema. Sie stellt die Grammatik der Sprache zur Verfügung.

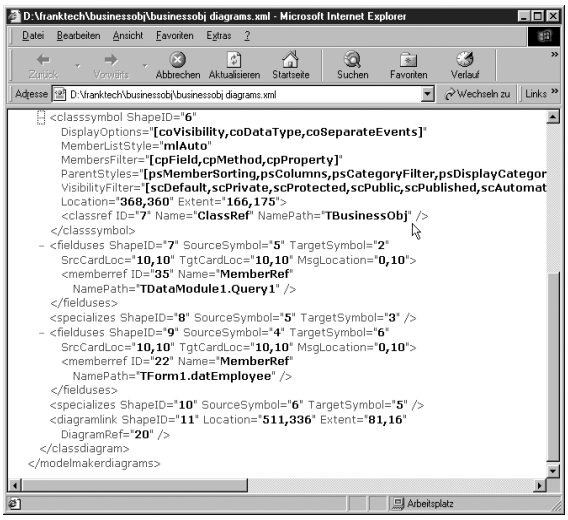

Abb. 1.25: Export der MDA-Modelldaten als XMI

Es ist festzustellen, dass die meisten UML Tools sich betreffend der verfügbaren Diagramme und Modellinformationen einander nähern. Auch wenn z. B. mit Stereotypen die Notation erweiterbar ist, die Elemente normieren sich. Diese Tatsache führte zum Bedürfnis, diese Modelldaten auch unter den CASETools austauschen zu können.

Einfach gesagt, besteht eine XMI-Datei aus einem Header- und einem Content-Bereich. Der Header bestimmt das Metamodell und der Content demzufolge das Modell mit den zugehörigen Elementen. Das Metamodell hat als oberste Klasse das Element. Diese Elemente sind dann pro Diagramm verfügbar.

In einer heterogenen Entwicklungslandschaft, auch im Hinblick auf die kommende UML 2.0, lassen sich künftig Ergebnisse zwischen verschiedensten Tools austauschen. Manchmal steht dahinter eine weiter reichende strategische Entscheidung, nämlich sich als Unternehmen nicht von den Herstellern der eingesetzten Tools abhängig zu machen und aus einem beliebigen Tool „Architektur“ zu erzeugen.

Auch in der modernen Architektur ist der Detaillierungsgrad anscheinend verloren gegangen: Die auf dem Brett entstandene Zeichnung und das architektonische Resultat eines Hauses haben sich als unnötige Ornamente und Schnörkel erwiesen. Doch die innere Komplexität der Gebäude ist gewachsen. Außenwände so genannter intelligenter Gebäude (Intelligent Buildings) verbergen hinter der glatten Fassade ein äußerst kompliziertes Innenleben.

Damit wird klar, dass nebst der Architektur auch die Funktionalität einen hohen Detaillierungsgrad aufweisen kann.

> *Der Trend im Softwarebau wird sein, die Architektur zu standardisieren (J2EE, CLX, .NET), aber die Funktionalität zu erhöhen.*

Standardisieren kann auch vereinfachen bedeuten. Denn in den seltensten Fällen wird künftig jeder Teil eines Modells ein anderer Sprachentyp sein, das entspräche der Arbeit

mit Einzelelementen. Ziel ist die mehrmalige Wiederverwendung von Variationen eines Standardtyps, um so ein generelles Ordnungsprinzip zu erreichen. Techniken wie ein „Common Type System“ oder die CLX Runtime zielen jetzt schon darauf ab.

1.4.4 Pattern-Generierung

Weitere Techniken sind im Gespräch, die alle eine gewisse Gemeinsamkeit mit MDA aufweisen, die sich „generative Programmierung mit adaptiven Mustern“ nennt. Diese Technik lohnt sich vor allem dort, wo man lange in der gleichen Fachrichtung entwickelt, also gute Kenntnisse der „Business-Domain“ hat. Die Umgebung von Fachanwendungen impliziert meistens eine ähnliche Struktur, sodass weitere Anwendungen wie eine Instanz aus einem Metamodell entstehen könnten.

Bei der Entwicklung eines CAD Tools oder einer Software aus der Regelungstechnik sehe ich kaum eine Chance, von einer besseren Wiederverwendbarkeit als bisher zu profitieren. Auch der Begriff „Intentional Programming“[xvi] gehört zu einer Thematik, welche vorderhand ein interessanter Microsoft-Traum ist, aber durchaus Potenzial im Zusammenhang mit dem Applikations-Framework .NET besitzt.

Muster im Großen wie im Kleinen, von Architektur-Mustern über Entwurfs-Muster bis hin zu Code-Mustern spielen bei der objektorientierten Entwicklung eine Rolle, wenn es darum geht, Effizienz und Qualität gleichermaßen entscheidend zu steigern.

Patterns sind auch Denk- und Entwurfshilfen zur Erfassung eines Problems.

Ohne ein Tool allerdings, das die Verwendung solcher Muster aktiv unterstützt, ist nicht viel gewonnen. Zu vieles beschränkt sich dann auf das fehleranfällige, manuelle Abschreiben und Kopieren von Vorlagen.

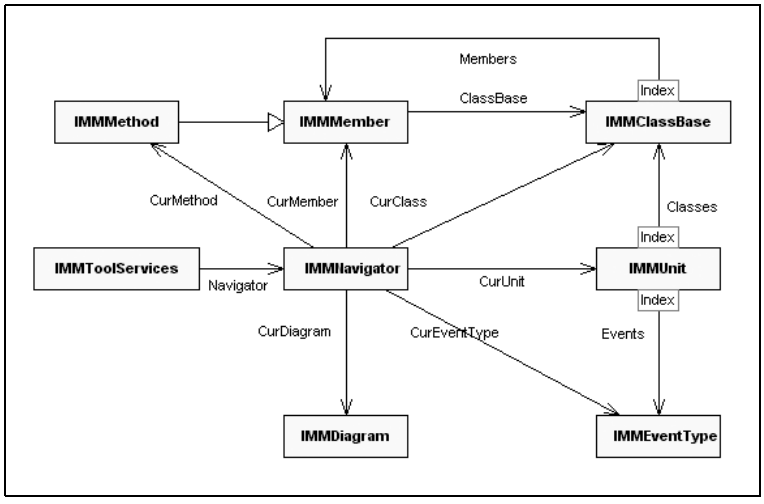

Abb. 1.26: Pattern Generator Framework

Das exemplarische Tool hat gemäß Abb. 1.26 einen Navigator, der anhand von Pattern Diagrammen oder eines Katalogs die nötigen Member und Methoden zusammensucht und anschließend Klassen, Units oder Typen generiert. So kann ein PBE (Pattern by Example) Framework wirken.

Die musterbasierte Anwendungsentwicklung soll diese Schwierigkeit umschiffen. Mit PBE ist eine auf Beispielen basierende Entwicklungstechnik realisierbar. Bewährte Lösungen lassen sich schrittweise in automatisch wiederverwendbare Muster überführen.

> *In PBE kann man Musterdefinitionen erstellen, durch einfaches Markieren parametrisieren und auf „Knopfdruck" für die gewünschte Zielsprache expandieren.*

Bei einer konventionellen Vorgehensweise sind Design Patterns, die als theoretische Konzepte existieren, immer wieder neu im jeweiligen Kontext zu implementieren; mit PBE werden sie jeweils nur einmal als Musterdefinition implementiert und sind dann für verschiedene Sprachen und Kontexte erweiterbar.

Kommen wir zur Template-Metaprogrammierung, im Folgenden auch „statisch" genannt, da der Compiler sie im Gegensatz zu dynamischen Programmen nicht zur Laufzeit, sondern während der Übersetzung eines Programms ausführt. Während „normale" Programme Daten verarbeiten, verarbeiten Metaprogramme andere Programme. Das erste Metaprogramm geht wohl auf Erwin Unruh zurück, der einen C++-Compiler dazu überredete, Primzahlen als Warnungen auszugeben. Ein einfaches Template zum Schluss zeigt einen ersten Ansatz, den wohl jedes Tool schon besitzt:

```
unit ArrayProp_List;
//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

  TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!ItemCount!>: Integer;
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!ItemCount!>: Integer read Get<!ItemCount!>;
    property <!Items!>[Index: Integer]: <!TObject!>
                                        read Get<!Items!>;
  end;
```

Hier wird eine Klasse als indexierte Listenklasse mit Makros zur Designzeit spezifiziert.

1.4.5 OCL-Regelwerk

In der Literatur sind die Einstiege in OCL meist mit dem Hinweis auf die UML 2.0 zu finden, wie der Originaltext der OMG bestätigt:

„On June 3, 2002 a revised proposal for OCL 2.0 has been publised. Information on this submission can be found on the OCL 2.0 Submission webpages. The final version of the submission, will be undergoing a final review together with the UML 2.0 Infrastructure submissions. We expect that OCL 2.0 will approved during the fall of 2002."

Am 3. Juni 2002 wurde also erstmals die revidierte Fassung von OCL 2.0 publiziert. UML und OCL haben sich als Sprachen zur Modellierung, zur Spezifikation und zum Design von objektorientierten Systemen etabliert. Wobei die meisten OCL gar nicht kennen, es also doch noch nicht etabliert ist. Die Modellierung eines Systems kann man als eine Serie von Redesign-Schritten betrachten, sodass sich Spezifikationen erweitern oder ändern lassen.

Bei der Änderung eines Modells hingegen möchte man direkt mit Modellparametern arbeiten, sodass die Spezifikation, z. B. von Geschäftsregeln, sich in Form eines Dokumentes aus dem Modell aktualisieren lässt. OCL ist ein Weg dazu, Bedingungen oder Regeln schon im Modell präzise als Code oder Text zu formulieren. Als Ergebnis erhält man meist mehrere Diagramme mit unterschiedlichen Abstraktionsniveaus. OCL hat zwei Hauptfunktionen:

1. OCL ist zusätzlicher Text oder Stereotypen-Code im UML-Diagramm.
2. OCL ist eine Sprache, die zum Spezifizieren, Testen oder logischen Abfragen dient.

Am Beispiel eines konkreten Sequenzdiagramms sehen wir einen ersten Vorteil von OCL. Zwischen den Aufrufen sind Zusicherungen formuliert, die das Objekt erfüllen muss, damit die Sequenz geordnet weitergeht.

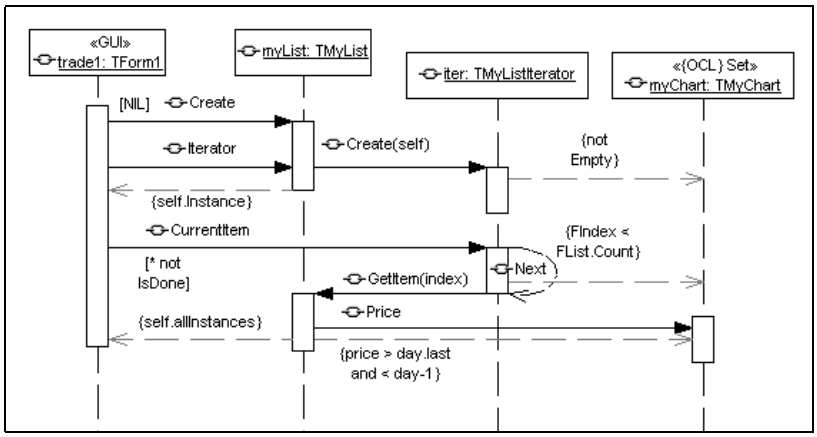

Abb. 1.27: Ein Szenario mit Zusicherungen

In Abb. 1.27 wird eine Liste gefüllt, die eine erste Zusicherung erfüllen muss, die Liste darf nicht leer sein `{not Empty}`, weil ansonsten die zweite Zusicherung keine gültige

Instanz zurückliefert `{self.Instance}`. Während des Lesens der Liste mit einem Iterator wird ständig geprüft, ob der Index die Kapazität der Liste nicht überschreitet.

Die Zusicherung `{FIndex < FList.Count}` hat auch klare Auswirkungen auf den Code, sofern man aus einem Sequenzdiagramm in ferner Zukunft Code generieren kann. Es gibt aber einen interessanten Zusammenhang mit der Funktion `Assert()`, die fast jede Sprache kennt.

> *Eine Zusicherung ist ein boolescher Ausdruck, der niemals falsch werden darf. Eine Assert-Funktion erfüllt genau diese Aufgabe, sie liefert auch true oder false zurück. Bei OCL wie bei Assertions geht es um die Korrektheit des Codes.*

Hier kommt häufig die Frage, welcher Unterschied denn zwischen einer Zusicherung und einer Bedingung in UML besteht:

- Zusicherung in geschweifter Klammer {} gibt immer true oder false zurück und ist identisch mit der Assert-Funktion
- Bedingung in eckiger Klammer [] kann auch eine allgemeine Anweisung sein, wie [nachbestellen] oder [Mindestmenge] und muss nicht logisch messbar sein

Änderungen in einem Diagramm oder Modell ziehen dann oft Änderungen in anderen, dazu verwandten Diagrammen und Verbindungen nach sich. Um deshalb durch die verschiedenen Ansichten navigieren zu können und um ein Redesign zu erleichtern, ist die Unterstützung eines Tools erforderlich.

Und diese wiederum benötigen OCL, um die Regeln überprüfen zu können, z. B. ob eine Instanz als Singleton nach einer Modelländerung wirklich nur einmal existiert. Wie funktioniert nun die OCL, die bereits in der Bold-Architektur als objektorientierte Abfragesprache implementiert ist?

OCL konkret

Auf gut Deutsch kann man OCL als Objekt-Zusicherungssprache verstehen, d. h., man sichert oder regelt ein Modell mit Einschränkungen, indem man Garantien oder eben Zusicherungen (constraints) abgibt. Diese Zusicherungen betreffen einerseits den Nutzen einer Klasse: Es werden gültige Eingangswerte erwartet (precondition). Andererseits macht die Klasse selbst die Zusicherung, gültige Ausgangswerte zu liefern (postcondition).

Jede Zusicherung muss mit einer Klasse des Modells verknüpft sein. So muss z. B. das Alter in einem Modell Kinoreservationssystem in der Klasse `TKinoBelegung` größer gleich 12 Jahre betragen oder in OCL ausgedrückt:

```
age >= 12 or parent.age > 12
```

Des Weiteren lassen sich Vor- und Nachbedingungen an eine Operation oder Methode zusichern, die ihre Wurzeln in OO-Modellierungssprachen wie Syntropy, Catalysis und BON haben und zudem vom großen Vorbild Eiffel von Bertrand Meyer abstammen. OCL ist eine formale Sprache, die ursprünglich IBM entwickelte. Zusicherungen können in vier Varianten zur Anwendung gelangen:

- <invariant>
- <precondition>
- <postcondition>
- <guard>

Pre- und Postcondition lassen sich also auch mit Zusicherungen in OCL ausdrücken.

Der <guard> ist eine bedingte Zusicherung, die beim Feuern eines Zustandswechsels in Aktivitätsdiagrammen oder State Events zum Einsatz kommt.

Ein <invariant> ist eine Zusicherung bezüglich einer Klasse, die in der Regel für die Lebensdauer über alle Klassen, Instanzen und Schnittstellen der Klasse erfüllt sein muss, konkret muss sie <true> zurückgeben:

```
Context Enterprise inv:
salesman.knowledgeLevel >= 5
```

Eine Vorbedingung in einer Operation muss in dem Moment wahr sein, wo die Operation in die Ausführung kommt. Diese Vorbedingung ist ein allgemeines Konzept, mithilfe dessen bereits in der Analysephase Modelle nach dem „Design by Contract“ entstehen. Diese Entwurfstechnik prüft in der Regel die Zusicherungen nur während der Phase der Entwicklung und Fehlersuche. Als Beispiel kommt die Listenverwaltung in Abb. 1.27 nochmals zum Tragen:

```
context myList: invariant TMyList
pre: {length(values) >= 2}
post: {IsSorted}
```

Für jedes Objekt der Klasse `TMyList`, weil invariant, gilt vor dem Zugriff eine Mindestmenge in der Liste von zwei Einträgen, und nach dem Zugriff sollte die Liste sortiert sein. In diesem Zusammenhang eine meiner Lieblingsfehlermeldungen:

```
Tradelist:1 Eintrag wurde gefunden, möchten Sie sortieren?
```

Die zugehörigen Nachbedingungen lassen sich mit dem Schlüsselwort <return> weiter auszeichnen und charakterisieren den bedingten Zustand nach der Operation, d. h., Nachbedingungen bleiben nach dem Aufruf der Operation bestehen (Folgendes Beispiel in OCL will ich gleich erörtern):

```
context Salesperson::sell( item: TBook ): Real
pre: self.sellableItems-> includes( item )
post: not self.sellableItems-> includes( item ) and
                                  result = item.price
```

OCL bindet auch Kollektionen von Objekten und Attributen mit dem eigenen Kollektionsoperator -> und betrachtet all die Resultate links vom Operator -> immer als eine Kollektion; auch wenn die Abfrage nur ein Element findet, wird es als einziges Element in eine Kollektion umgewandelt.

Eine Abfrage lässt sich also auf eine Kollektion anwenden.

Die <Collections> werden dort gebraucht, wo die Multiplizität zwischen Klassen 1.. ist, d. h., eine Klasse erzeugt mehrere Instanzen, auch Exemplare genannt.*

Obiges Beispiel bedeutet nun in der post-Variante: Gib vom nicht verkauften Buch den zugehörigen Preis als Resultat zurück. Wenn kein Preis vorhanden ist, gilt das Element als falsch `(result = not true)`.

Der Mengenoperator <includes> liefert den Wert true, wenn das genannte Objekt, in meinem Fall `item`, in der angegebenen Menge enthalten ist. Wenn es enthalten ist, überprüft die Kollektion der nicht verkauften Bücher, ob das gesuchte `item` enthalten ist. Man übergibt sozusagen die mögliche Menge in die Abfragemenge, also von links nach rechts, um dann eine Resultatmenge zu erhalten. Die mögliche Menge der Kollektion sollte immer größer der Abfragemenge sein.

Zudem ist auch ein Alles-Mengenoperator möglich, <forAll> testet, ob eine Bedingung für alle Elemente der Menge gilt:

```
post: not self.sellableItems-> forAll(item.price > 40)
```

Die OCL ist eine reine Bedingungssprache, will heißen, jede Auswertung hat niemals eine Änderung der Daten oder des Modells zur Folge. Wenn eine Zusicherung nicht erfüllt ist, resultiert keine Korrektur oder Manipulation der Objekte durch die OCL. Das Ergebnis ist einzig und allein immer ein (boolescher) Wert.

Kollektionen lassen sich auch navigieren und verknüpfen, sodass gilt:

```
Salesteam.persons.consultants-> size
```

Das Statement ergibt die Anzahl aller Berater im Team, hingegen bedeutet

```
Salesteam.cars.size
```

einen Wert aller Größen von Geschäftswagen des Teams. Dieses Statement ist übrigens keine Zusicherung, sondern eine Auswertung, da die Bedingung fehlt. Wenn alle Teams gefragt sind, die weniger als 10 Berater bekleiden, haben wir wieder eine Zusicherung:

```
Salesteam.persons.consultants-> size < 10
```

Ein weiterer Navigationspfad für ein so genanntes Objektnetz in OCL muss als Beispiel folgende Zusicherung einhalten und lässt sich so beschreiben: „Sichere zu, dass alle Fahrer in einem Mietvertrag mindestens 18 sind.“ Hier das zugehörige Konstrukt als Lösung:

```
KfzMietvertrag self.fahrer-> forAll(f|f.alter >= 18)
```

Instanzen sind mit einem Alias verkürzbar, wie `f` zeigt. Oder eine letzte Invariante als doppelte Kollektion, sodass gilt: Alle Kunden der Klasse `Salesperson` müssen mengenmäßig weniger als 100 betragen und für die Klasse `Customer` alle Kunden mehr als 40 Jahre alt sein. Die abgeleitete Assoziation läuft demzufolge über zwei Klassen und die zugehörige Assoziation heißt `clients`:

```
context Salesperson inv:
clients->size() <= 100 and clients->
                          forAll(c: Customer | c.age >= 40)
```

Man nennt dies eine Kollektion von Instanzen in einer Assoziation (klingt nach Haarspalterei), auch bekannt unter dem UML-Stereotyp <link> oder Link-Klasse. Kollektionen können auch Unterklassen enthalten, die bestimmte Mengentypen verkörpern:

- Set ist eine Menge ohne doppelte Werte.
- Bag ist eine Menge mit doppelten Werten.
- Sequence ist ein geordneter Bag.

Auch Design Patterns lassen sich mit OCL genauer mit Zusicherungen spezifizieren, z. B. muss ein Composite eine Liste enthalten oder ein Strategie-Muster muss sich selbst als Parameter übergeben (self). Dies würde die einheitlichere Umsetzung von Patterns erheblich standardisieren.

Keine Regel ohne Ausnahme

Die OCL wie die UML haben eine große Schwäche: Meistens wird immer der „Happy-Day-Case“ behandelt, will heißen, ein Regelwerk für das Nicht-Einhalten von Bedingungen fehlt gänzlich, man geht immer vom Schönwetterfall aus. Wenn eine OCL-Bedingung nicht erfüllt ist, resultiert daraus keine Konsequenz. Kann ja auch nicht, da die OCL eine Spezifikationssprache ist und in erster Linie zum Verfeinern der Objektmodelle dient.

Wie aber sehen die Ausnahmen und deren Behandlung aus? Auch mit der Ausnahmebehandlung folgt Delphi wie C# bekannten Konzepten anderer OO-Sprachen. Exceptions sind Objekte, die Informationen über Fehlerzustände enthalten, derentwegen ein Programm eine Operation nicht wie geplant ausführen konnte, zum Beispiel aus Speichermangel, wegen einer Schutzverletzung oder einer Division durch null. Trat bisher eine derartige Bedingung auf, brach das Programm mit einem Laufzeitfehler ab.

Um dies zu vermeiden, benötigte man immer tiefer geschachtelte IF-Statements mit alternativen **Ausführungspfaden**. Exceptions erlauben stattdessen eine gebündelte Fehlerbehandlung im Laufzeitsystem.

Exceptions

Eine Exception ist ganz allgemein eine Fehlerbedingung oder ein anderes Ereignis, das den normalen Ablauf einer Anwendung unterbricht. Wenn eine Exception ausgelöst wird, geht die Programmsteuerung von der gerade ausgeführten Anweisung auf eine Exception-Behandlungsroutine über.

Object Pascal unterstützt die Ausnahmebehandlung durch die Bereitstellung einer Struktur, wo der Compiler die normale Programmlogik von der Behandlung der Ausnahmefälle **trennt**. Dies führt zu besser wartbaren und erheblich robusteren Anwendungen.

Object Pascal stellt Exceptions durch Objekte dar. Dies hat eine Reihe von Vorteilen, zu denen als wichtigste zählen:

- Exceptions lassen sich durch Vererbung hierarchisch gruppieren.
- Neue Exceptions sind verwendbar, ohne bestehenden Code zu beeinflussen.
- Ein Exception-Objekt kann Informationen (wie z. B. eine Fehlermeldung oder einen Fehler-Code) vom Ort der Auslösung bis zum Ort der Behandlung transportieren.

In der Unit `SysUtils` ist die Erzeugung und Behandlung von Exceptions für die Laufzeitbibliothek implementiert. Wenn eine Anwendung die Unit `SysUtils` verwendet, wandelt die Unit alle erkannten Laufzeitfehler automatisch in Exceptions um.

Dies hat zur Folge, dass Fehlerbedingungen, wie nicht ausreichender Platz auf dem Heap – die sonst zum Abbruch der Anwendung führen würden –, abgefangen und gezielt behandelt werden können.

Stark in Szene setzt sich auch folgendes Beispiel: Mit dem Ereignis `OnException` und dem Umlenken der Laufzeitfehler in eine Datei lässt sich eine ganz brauchbare Protokollierung der Ausnahmen während der Entwicklung und dem Testen einsetzen. [xvii]

Die Exception-Klasse

Eigene Exceptions zu erzeugen, ist natürlich möglich. Dazu reicht es schon aus, eigene Referenzen von bestehenden Exceptions zu erzeugen:

```
Type MyException = Class(EMathError);
```

Solche Klassen sind auch in „guten“ Modellen enthalten. Dabei bildet die Klasse Exception die Mutter aller in Delphi existierenden Exceptions. Suchen Sie einmal im Browser nach Exception. Sie sehen dann die gesamte Exception-Hierarchie.

Nun haben Sie zwar auch einen Nachfolger von Exception gebildet, aber `MyException` ist gleichzeitig auch als eigene Klasse definiert. Wir können also `MyException` ganz im Sinne der Theorie der objektorientierten Programmierung erweitern. Sinnvolles Beispiel dafür ist die in Delphi bestehende Exception-Klasse `EInOutError`:

```
Type
  EInOutError = Class(Exception);
    ErrorCode: Integer;
  end;
```

Neben allen Elementen der Klasse Exception definiert `EInOutError` auch noch das Feld `ErrorCode`, das die Fehlernummer der fehlgeschlagenen Ein-/Ausgabeoperation enthalten kann.

Run Time Type Information

Dieses aus der C++ Welt ererbte Gut bringt eine enorme Flexibilisierung mit sich. Delphi gibt den Programmen zur Laufzeit Zugriff auf Informationen über den Objekttyp. Insbesondere kann man den neuen Operator `is` benutzen, um zu bestimmen, ob ein gegebenes Objekt von einem gegebenen Typ oder einer seiner Nachkommen ist.

Man kennt von C++ die Möglichkeit des dynamischen Typecastings von zwei Klassen, das auch mit Object Pascal möglich ist:

```
type
  TClass1 = class
  end;
  TClass2 = class(TClass1)
  end;
var a: TClass1; b: TClass2;
  b:= TClass2.Create;
  a:= b as TClass1; //Vorfahr:= Nachfahr, geht umgekehrt nicht
```

Der Operator is

Der Operator `is` ermöglicht eine dynamische Typüberprüfung. Mit dem is-Operator können Sie überprüfen, ob der aktuelle (zur Laufzeit vorliegende) Typ einer Objektreferenz zu einer bestimmten Klasse gehört. Die Syntax des is-Operators lautet wie folgt:

```
ObjectRef is ClassRef
```

Dabei ist `ObjectRef` eine Objektreferenz und `ClassRef` eine Klassenreferenz. Der is-Operator liefert einen booleschen Wert. Das Ergebnis ist True, wenn `ObjectRef` eine Instanz der über `ClassRef` angegebenen Klasse oder einer davon abgeleiteten Klasse ist. Andernfalls ist das Ergebnis False. Enthält `ObjectRef` den Wert NIL, ist das Ergebnis immer False.

Der Compiler meldet den Fehler „Typen nicht miteinander vereinbar (Type mismatch)“, wenn bekannt ist, dass die deklarierten Typen von `ObjectRef` und `ClassRef` nicht verwandt sind. Dies ist der Fall, wenn der deklarierte Typ von `ObjectRef` weder Vorfahr noch Nachkomme ist.

Der is-Operator wird häufig im Zusammenhang mit einer if-Anweisung benutzt, um eine überwachte Typumwandlung auszuführen, wie z. B.:

```
if ActiveControl is TEdit then
            TEdit(ActiveControl).SelectAll;
```

Wenn die is-Abfrage den Wert True liefert, lässt sich `ActiveControl` sicher in die Klasse `TEdit` umwandeln. Wenn man den is-Operator mit anderen booleschen Ausdrücken mixt, muss man den is-Test in Klammernotation setzen:

```
if (Sender is TButton) and (TButton(Sender).Tag <> 0) then ...;
```

Der Operator as

Der Operator `as` ermöglicht überprüfte Typumwandlung (Typecasting). Die Syntax des as-Operators lautet wie folgt:

```
ObjectRef as ClassRef
```

Ergebnis ist eine Referenz auf dasjenige Objekt, auf das `ObjectRef` verweist, jedoch mit dem Typ, auf den `ClassRef` verweist. Das ist, wie wenn eine Frau auf mich zeigt und ich dann alle Männer, auf die ich zeige, sofort zu Frauen mache.

Zur Laufzeit muss `ObjectRef` den Wert NIL enthalten oder eine Instanz der über `ClassRef` angegebenen Klasse oder einer davon abgeleiteten Klasse sein. Ist keine dieser Bedingungen erfüllt, wird eine Exception ausgelöst. Ansonsten ähnliches Verhalten wie beim is-Operator.

Der as-Operator wird häufig im Zusammenhang mit einer with-Anweisung benutzt (oder einer Schnittstelle), wie Folgendes zeigt:

```
with Sender as TButton do begin
  Caption := '&OK';
  OnClick := OkClickHandler;
end;
```

Wird der as-Operator direkt in einer Variablenreferenz benutzt, muss die as-Typenumwandlung in Klammern eingeschlossen sein:

```
(Sender as TButton).Caption := '&UML_OK';
```

1.4.6 UML 2.0 Projekt

In dieser Kurzübersicht gehe ich auf die einzelnen Diagrammtypen anhand eines durchgängigen Projektes ein. Es soll die Typen darstellen und zugleich eine kompakte Auffrischung Ihres Wissens bezwecken.

Beim Projekt handelt es sich um den TRadeRoboter, der nach bestimmten Signalen des gleitenden Durchschnittes mehr oder weniger selbstständig an den Finanzmärkten agiert, sprich kauft und verkauft. Im Kapitel Design Patterns liefert er ständig Beispiel-Code zu den einzelnen Patterns. Auch in den Architektur-Patterns dient der TRadeRoboter nochmals als vertiefte Studie zum Master-Slave Pattern. Die Neuerungen der Notation bezüglich UML 2.0 sollen in diesem Streifzug nicht zu kurz kommen.

Use Case

> *"Use-case diagrams, which model the functional requirements of the system in terms of actors and actions."*

Im Sinne der Anforderungsanalyse hat sich beim Use Case kaum etwas geändert, er ist weltweit etabliert. Use Cases lassen sich nach wie vor in der Analyse einsetzen:

- Initialisierung der Anforderungsermittlung
- Anforderungen und Soll-Situation darstellen
- Geschäftsprozesse im groben Funktionsblock
- Allgemeine Diskussionsgrundlage
- Definieren der Benutzervertreter als Akteure

Die wichtigsten Notationen sind:

Kommunikationsbeziehungen in einem UC lassen sich nur zwischen Akteur und Anwendungsfall anlegen, nicht zwischen den Akteuren selbst. Ein Akteur kann auch eine Maschine oder ein Umsystem wie eine Basisstation eines GSM-Netzes sein.

Man benutzt die <extend>-Notation bei einem UC, der einem bestehenden UC ähnlich ist, ihn aber um eine Variante erweitert. Der bestehende UC ist nicht unbedingt vom anderen UC abhängig.

Man benutzt die <include>-Notation bei einem UC, wenn man sich in zwei oder vielen UC wiederholt und diese Wiederholung auslagern und gemeinsam nutzen möchte. Der bestehende UC ist vom UC, den er beinhaltet, abhängig. Typisch ist auch, dass ein Akteur selten mit einem <include>-UC kommuniziert.

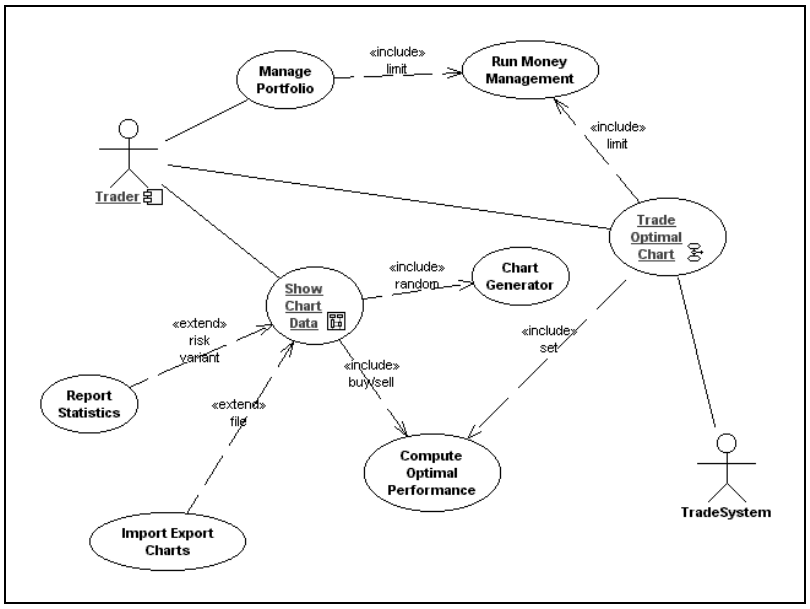

Abb. 1.28: Use Case des TRadeRoboters

Activity

> *Activity diagrams, a variant of state-transition diagrams, which are used to show state changes driven by internal processing (as opposed to external events).*

Hier ist eindeutig der Trend zu mehr eigener Prozessmodellierung auszumachen. Das Einsatzgebiet bleibt de facto beim Alten:

- Geschäftsprozesse im Ablauf
- Ablaufszenarien und Alternativen
- Parallele Prozesse oder Threads
- Workflow innerhalb der Organisation

In den früheren Versionen von UML war das AD eine Art Zustandsdiagramm; mit dieser ambivalenten Haltung ist in der Version 2 Schluss. Das Diagramm hat eine eigene Basis im Metamodell erhalten und heißt neu schlicht und ergreifend Activity oder Aktivität.

Die einzelnen Schritte nennt die OMG nun <activity invocation>, was einem Verhaltensaufruf gleichkommt. Hier versucht man die Trennung zwischen der Aktivität und dem Aufruf klarzustellen, sodass sich entweder eine Methode als Prozessschritt (wie bisher) oder eine ganzes Untermodell (ein anderes Activity) in einer Prozesskette aufrufen lässt.

Einige Notationen sind auch derart erweitert, dass eine Annäherung zu Petrinetzen möglich ist. Gemeinsam sind die beim Zustandsübergang ausgelösten Aktivitäten, die wiederum Folgeaktivitäten durch Ereignisse auslösen können.

Prozessorientierte Diagramme haben den Vorteil, eine verständlichere Semantik zu kommunizieren, notwendigerweise auch aus dem Grunde, weil eben die meisten Fachleute nicht in Datenmodellen oder Klassen denken wollen.

Sequenz, Selektion, Iteration und Subroutine, d. h. die Atome einer Programmiersprache, sind in einem AD nach wie vor gültig und abbildbar. Auch ODER-Bedingungen lassen sich modellieren. In die Aktivität <Delete Position> in Abb. 1.29 führen zwei Transitionen, welche eine Oder-Bedingung darstellen. Entweder wird der Handel aus der Watchlist genommen und die Position gelöscht **oder** die nächste Quote erhält kein Signal und die Position wird auch gelöscht.

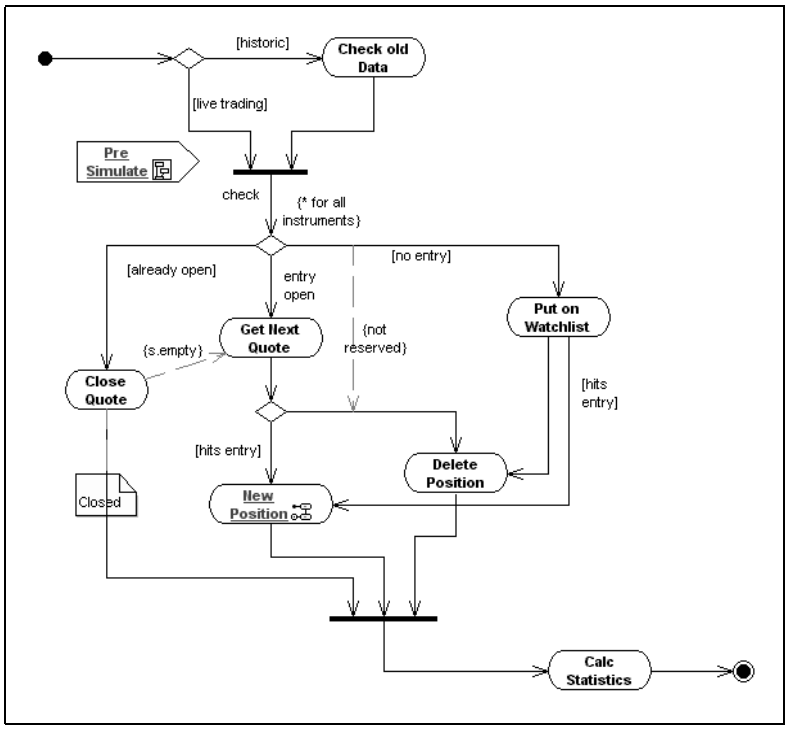

Abb. 1.29: Das Activity zeigt den UC <Trade Optimal Chart>

Class Diagram

> *"Class diagrams are one of the most widely used of the diagrams, and show the structure of the classes involved in the application, along with their inheritance and use relationships."*

Klassendiagramme haben sich stabilisiert und sind als Modell total etabliert. Der Einsatz ist sogar für das Finden der fachlichen Attribute in Masken zum GUI-Design erweitert worden, sodass ein CD universeller, aber auch genauer denn je ist:

- Strukturierung von Methoden und Eigenschaften
- OO-Entitäten als Fachklassen
- Definition von Interfaces und Komponentenklassen
- Finden der Attribute mit erstem Datenmodell oder GUI-Design
- Relationen der Klassen und so gut wie überall ...

Das Klassendiagramm wurde verfeinert und im Hinblick auf den vermehrten Einsatz von Komponenten erweitert. Alle bestehenden Notationen wurden unverändert übernommen. Klassen und Interfaces erhalten neu einen <part>, das ist ein Teil einer Klasse. Dieses Teil (im wahrsten Sinn des Wortes) ermöglicht in einer frühen Phase, den Typ einer Klasse zu bestimmen, der dann später in der Implementierung mit Operationen und Attributen ausgearbeitet wird.

> *Eine Assoziation ist die einfachste Beziehung zwischen zwei Klassen. Bei einer Assoziation ist deren Verhältnis am häufigsten 1..*.*

In einer Assoziation kann auch eine Klasse die Multiplizität 0 besitzen, weil es ausreicht, wenn nur die Instanzen einer der beteiligten Klassen die Instanzen der anderen Klasse kennen.

Als Multiplizität findet man auch das Wort Kardinalität, das so viel wie Auftretenshäufigkeit oder Beziehungsmenge heißt und im Englischen mit multiplicity übersetzt wird. Daher die zwei Wörter, die dasselbe bedeuten. Alle sechs Beziehungen folgen in der Übersicht, sozusagen im UML Sixpack:

1. **<generalisation>** – hierunter versteht man die Vererbung und die Polymorphie, die meist vom Allgemeinen als Basisklasse zum Besonderen läuft und mit einem abgeschlossenen Pfeil notiert wird.
2. **<realisation>** ist eine Beziehung zur Designzeit, die das Interface mit der zu realisierenden Klasse aufzeigt. Die realisierende Klasse hat eine gestrichelte Linie mit einem geschlossenen Pfeil zum Interface oder zur abstrakten Klasse. Alternativ ist auch ein Lollipop (offener Kreis) darstellbar.
3. **<association>** ist seine Beziehung zur Laufzeit mit loser Kopplung zwischen Objekten (Instanzenbeziehung), z. B. ein DLL-Aufruf oder Dialoge, temp. Strukturen, in Delphi und C++ pointer, wird mit einem offen Pfeil dargestellt.
4. **<aggregation>** – hier besitzt die Klasse ein Objekt (enge Kopplung) zur Designzeit (Klassenbeziehung), z. B. `TButton` in `TForm`, im Konstruktor, in Delphi, in C++ member, als offene Raute bei Klasse `TKonto` notiert, d. h. die Klasse `TKonto` aggregiert die Klasse `TTransaction`.

5. **<composition>** ist stärker als <aggregation> und meint die physik. Einbindung, z. B. `memo.lines.add`, meistens in Komponenten mit drei Klassen zu finden und als schwarz gefüllte Raute notiert.
6. **<dependency>** ist eine reine Typisierungsbeziehung, entweder man ist abhängig von einem Typ, z. B. einer Exception, oder man braucht eine Standardklasse wie `TStringList`, die im Fachmodell nichts zu suchen hat, aber trotzdem ins Modell einfließen soll. Wird mit einer gestrichelten Linie und offenem Pfeil dargestellt.

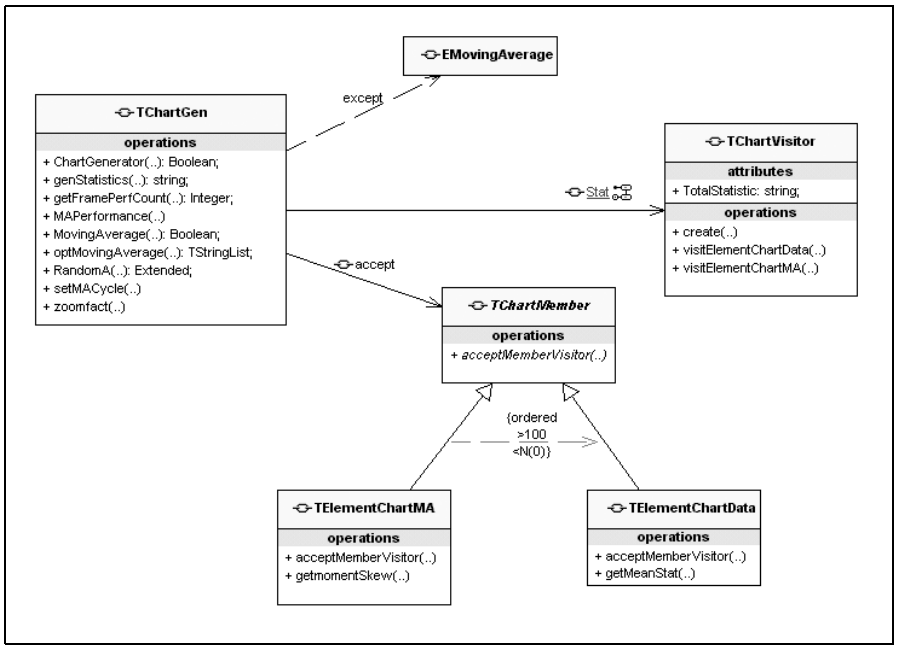

Abb. 1.30: Ein Klassendiagramm aus dem TRadeRoboter

State Event

> *"State-transition diagrams (also known as 'statechart diagrams') are used to show the states that an object can occupy, together with the actions which will cause transitions between those states."*

Zustandsdiagramme sind jetzt strikter von den Activities semantisch getrennt, sonst erhalten sie die gleiche Bedeutung als Zustandsmaschine und bei Klassen mit extensiven Zustandswerten:

- Dynamisches Verhalten innerhalb der Klassen
- Objektlebenszyklus der möglichen Zustände
- Validieren der möglichen bedingten Übergänge
- Zustandsübergänge durch Ereignisse verfolgen
- Zustandsautomaten mit deterministischer Case-Logik

Zustände haben meistens als Bezeichnung ein **Partizip** (gebucht, reserviert, gecheckt, closed, storniert etc.), d. h., das Ereignis oder die Aktion hat bereits stattgefunden und wird als Ergebniszustand gespeichert.

Im Zusammenhang mit den Ports im Component kann ein State Event in der Version 2 auch angeben, in welcher Reihenfolge man die Methoden eines Ports benutzen kann.

Wenn die Beschriftung einer Transition kein Ereignis enthält, sondern nur eine Bedingung <guard>, ist damit ein Zustandswechsel gemeint, der beim Beenden einer Aktivität fast automatisch eintritt, oder das Ereignis findet im Zustand statt. Warum fast? Weil eben die Bedingung erfüllt sein muss.

> *Beim Zustandswechsel ist es möglich, sowohl eine Aktion durchzuführen als auch ein weiteres Ereignis zu versenden, das so genannte Zustandsautomaten anderer Klassen akzeptieren. Hierfür lässt sich auch eine Zielklasse angeben.*

Die Syntax einer Transition kann zusammenfassend folgende optionale Elemente enthalten, wobei mindestens ein Ereignis oder eine Bedingung vorhanden sein muss:

Syntax: **Ereignis (parameter) [Bedingung] / Aktion**

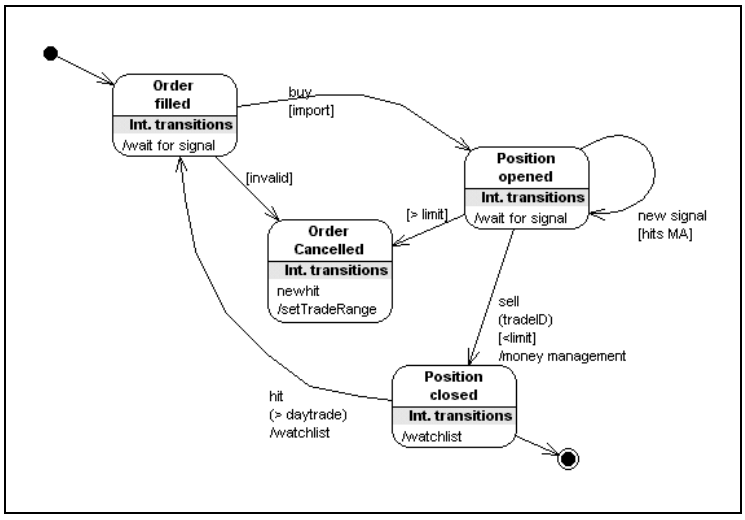

Abb. 1.31: Die Transitionen mit Bedingungen

Sequence

> *"Sequence diagrams show how objects interact through time, by displaying operations against a timeline."*

Das Sequenzdiagramm ist eine Art der Interaktionsdiagramme, die andere Art ist das Kollaborationsdiagramm. Beim SEQ hat mit UML 2.0 eine Präzisierung stattgefunden:

- Nachrichtenfluss der Objekte
- Zeitliche Aufrufstruktur/Aufrufkaskaden

- Dynamischer Ablauf zwischen den Objekten
- Feinstruktur mit Iteration, Selektion oder Subroutine

Durch den vermehrten Einsatz der Echtzeitmodellierung hat sich eine Erweiterung in der Sequence aufgedrängt, welche die zeitlichen Bedingungen festlegen kann.

Neu aufgenommen wurde das Timingdiagramm aus der Elektrotechnik und Echtzeit, zudem lassen sich mittels <parts>, <connectors> und <gates> interne Laufzeitstrukturen von Klassen oder Sequenzen abbilden. Die <parts>, also Teile, hatten wir schon bei den Klassen, ich komme zu den <gates>, also zu den Toren.

> *Auch hier ist die Neuerung, innerhalb des Diagramms auf andere Sequenzen, wie eine Subroutine, verweisen zu können. Es gleicht einem GOTO, ist aber noch besser, auch Iterationen und Selektion sind nun möglich. Sequenzen lassen sich dann zerlegen und so auch mehrfach referenzieren.*

Der File-Zugriff im Beispiel ist ein Kandidat für ein Auslagern als eigene Sequenz, die sich dann mit einem Hyperlink oder eben <gates> verknüpfen lässt. Der Abruf der einzelnen Berechnungen für die Charts ist als Loop dargestellt. Weiterhin ist auch ein Asterix * als Iterator möglich.

Damit diese Ein- und Ausstiegspunkte definiert sind, hat man die <gates>, also Tore, ins Leben gerufen. Zusätzlich hat man mit den erwähnten <gates> die Möglichkeit, Rücksprungbefehle abzufangen, sofern man den Namensraum eines Sequences verlässt, um Schnittstellen zu anderen Diagrammen, wie dem Component, definieren zu müssen.

Zwischen Sequenz- und Kollaborationsdiagramm besteht ein semantischer Zusammenhang, sodass ein Notationswechsel erfolgen kann oder muss. Die Notation und das Einsatzgebiet der beiden Diagramme differiert aber.

In der Regel gilt als Entscheid:

- Sequenzdiagramm bei wenig Klassen und vielen Nachrichten
- Sequenzdiagramm bei hoher zeitlicher Dynamik
- Kollaborationsdiagramm bei wenig Nachrichten und vielen Klassen

Grundsätzlich gelten beim Aufruf an die Instanzen folgende genauere Beziehungen der **Aufrufarten** in einem SEQ:

- **G**: Global/das aufrufende Objekt ist global
- **F**: Field/das aufrufende Objekt ist Attribut des Aufrufers
- **L**: Local/das aufrufende Objekt ist lokal in einer Methode
- **P**: Para/das aufrufende Objekt ist Parameter einer Methode
- **S**: Self/das aufrufende Objekt ist der Sender selbst

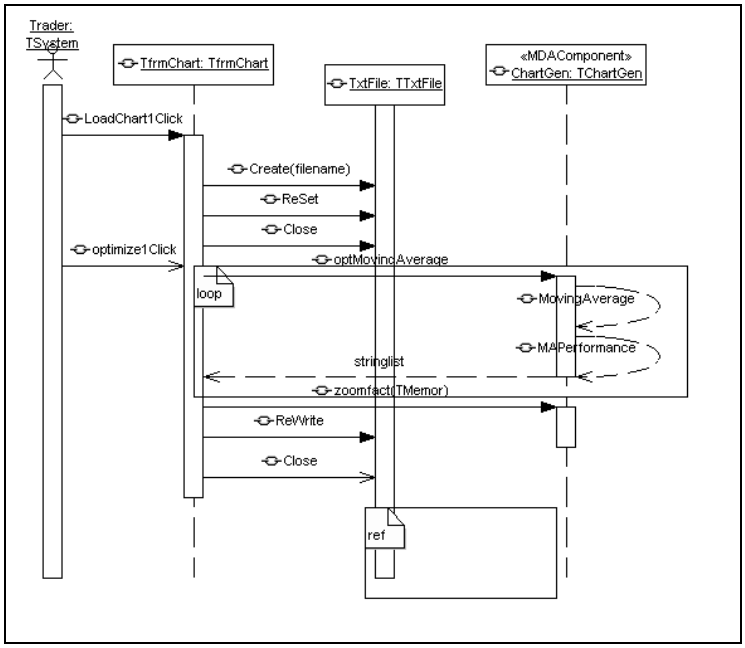

Abb. 1.32: Sequence neu mit <loop> und <reference>

Packages

> *"Packages diagrams are used to describe groups of a library at design time. Those will have internal structure of classes and their relations to outside dependencies."*

Die Paketdiagramme erhalten mit dem Konzept von Namensräumen, der Verwaltung und Versionierung von Code weiterhin Auftrieb:

- Auswirkung von Änderungen durch Abhängigkeiten
- Fachliche Gruppierung von Klassen zur Designzeit
- Vorbereitung zum Komponentenbau und Inventarisierung
- Libraries, Übersetzungseinheiten, Versionierung
- Modularisieren zum Aufbau einer Architektur

Packages sind genauer geworden, das Anzeigen der Klassen erhöht den Informationsgehalt, zudem lässt sich nun zwischen <access> und <import> als Abhängigkeitslinie unterscheiden; bei <access> greift eine Klasse auf die öffentlichen Elemente der anderen Klasse zu, bei <import> wird der ganze Geltungsbereich eines Paketes dem anderen Paket hinzugefügt oder eben importiert, z. B. bei einer IDL, DLL oder bei Type Libraries.

Mit dem Erweitern durch so genannte Profile in UML 2.0, welches eine Art Wörterbuch von Stereotypen darstellt, erhalten auch Packages eine Aufwertung.

Auf diese Weise erhalten Klassen oder Pakete innerhalb einer gemeinsamen Fachdomäne ihr eigenes Stereotypenpaket, z. B. das Profil Finanzmarkt.

Das Profil mit den zugehörigen Typen und sogar Operationen erlaubt, in einem Durchzug und in allen Diagrammen den Typ Integer auf Float zu wechseln!

Ein Profil sollte gemäß der Spezifikation von Version 2 auch toolübergreifend austauschbar und einsetzbar sein.

Packages sind nach der Spezifikation der UML ein Mechanismus zur Gruppierung von Design-Objekten. Diese Design-Objekte sind klar der Designzeit zuzuschreiben und sind eine Art **Bibliothek von Fachklassen**.

Jede Sprache hat so seine eigene Vorstellung, was ein Package ist. Allen gemeinsam ist die Gruppierung von Klassen. Das heißt auch, dass ein fachlich vollständiges Klassendiagramm ein Package ist. Was wiederum bedeutet, eine Unit mit ihren Klassen kann ein Package sein, im Sinne eines Entwurfszeit-Package.

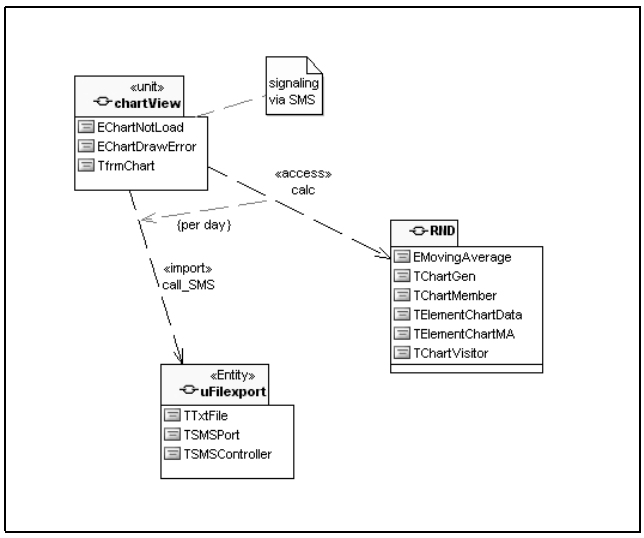

Abb. 1.33: Das Package mit Public-Beziehungen

Component

"Component diagrams are used to describe parts of a system at a high level, such as 'report generator'. These will have internal structure, and many tools will allow you to jump from the component to a detailed view of its structure."

Die Komponenten sind definitiv aufgewertet. Mit so genannten <ports> und <connectors> ist die Darstellung von internem Verhalten zu den exportierten Schnittstellen und anderen Komponenten möglich geworden.

- Software-Architektur (Implementierung)
- Technische Anforderungen des Frameworks

- Schichtenmodell (Multi-Tier etc.) zur Laufzeit
- Abhängigkeiten zwischen Prozessen des Systems
- Ausführbare Einheiten in Binärsicht

Vermehrt ermöglichen <ports> und <connectors>, zwischen einer angebotenen (Technik, Typ, Laufzeit und Signatur) und einer erforderlichen Schnittstelle das mögliche „Zusammenstecken“ prüfen zu lassen, um z. B. festzustellen, dass eine MSCOM nicht zu einem EJB oder einer CLX kompatibel ist.

Durch die Detailsicht mithilfe von <ports> sind neue, geschachtelte Komponenten oder komplizierte Architekturen wie ganze Container besser beschreibbar.

> *Im Gegensatz zu den Packages mit den fachlichen Gruppierungen ist das Interesse von Komponenten auf die technische Bereitschaft gerichtet.*

Ein Komponentendiagramm lässt sich verwenden, um die Zuordnung von Klassendiagrammen im Sinne von Packages und physischen Objekten zu Komponenten im physikalischen Systemdesign zu zeigen.

Abhängigkeiten bei Components sind meistens transitiv. In der Tat ist die Semantik jedoch von der Sprache abhängig. Das Java <import> ist nicht transitiv aufzufassen. Das <include> von C++ ist jedoch transitiv, beim <uses> von Object Pascal haben wir Wahlfreiheit. Sie sehen, eine transitive Abhängigkeit macht es schwierig, die Neuübersetzungen bei Änderungen einzuschränken.

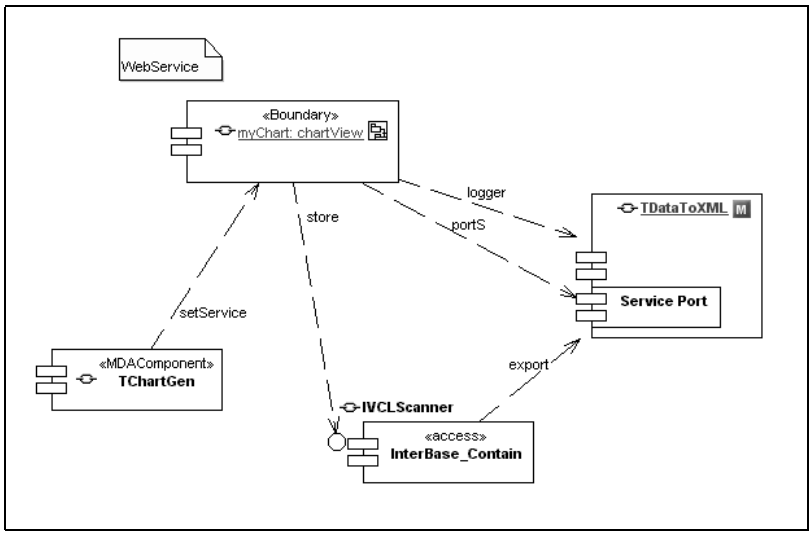

Abb. 1.34: Das Component mit Innenleben

Deployment

> *"Deployment diagrams are used to show how the model relates to hardware and, in a multi-processor design, how processes are allocated to processors."*

Das Deployment ist mit den Deployment Descriptors erweitert, sodass ein informeller Text in absehbarer Zeit auch ein Tool generieren kann und sozusagen als Manifest-Datei in den Knoten einpackt. Diese Angaben lassen sich dann auf der Zielplattform in eine Konfigurationsdatei umwandeln. Damit dient ein Deployment wie bisher dazu:

- Systemarchitektur im Netz/Topologie
- Zusammenspiel der Komponenten
- Hardwarearchitektur der Geräte und Peripherie
- Physische Aspekte der Verteilung

Der Entwurf mehrerer Teilprogramme oder Komponenten, die man auf unterschiedlichen Prozessoren oder Systemen ausführt, erfordert einen Designprozess, der sich völlig von der Modellierung von Klassen und Objekten unterscheidet. Um das Problem der Verteilung der **Prozesse** auf die einzelnen **Prozessoren** zu veranschaulichen, zeigt ein Verteilungsdiagramm (DEP) die realen Beziehungen zwischen Soft-, Hard- und Netzkomponenten an.

Ein DEP (ehemals Prozessdiagramm) enthält die Prozessarchitektur eines lauffähigen Systems. Dabei werden die Prozessoren eines Systems mit den auf ihnen laufenden Prozessen dargestellt oder auch die angeschlossenen externen Geräte und die Verbindungen zwischen den Hardwarekomponenten[9].

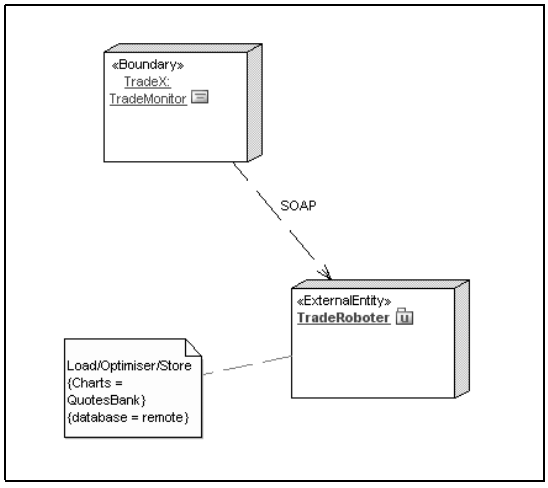

Abb. 1.35: Von Prozessen auf Prozessoren

[9] Der Begriff Komponente wird sehr vieldeutig gebraucht.

1.5 Patterns mit ModelMaker

Borland hat sich in den letzten Monaten großzügig auf Einkaufstour begeben, sodass nebst Bold auch ModelMaker in den höheren Versionen von Delphi 7 mit von der Partie ist. Ein deutliches Zeichen, wenn man die Modellierung vor die Codierung stellt.

Das Tool soll exemplarisch zeigen, wie mithilfe eines Tools der vermehrte Einsatz von Klassen und Patterns möglich ist und es damit die Güte fördert.

Vieles spricht für modellbasiertes Vorgehen: In der dunklen Nacht angetrieben mit Modellmacht ist es wie die Ruhe vor dem Sturm, denn die Ruhe vor dem Schirm lässt Lösungen klarer erscheinen und erhöht den Bau einer soliden Architektur. Welch ungeahnte Möglichkeiten ModelMaker bietet: Lesen Sie selbst.

1.5.1 Nahtlos

Zum Evaluieren von ModelMaker empfiehlt sich das Download und am besten starten Sie gleich mit dem Importieren eines ganzen Projektes. Hier zeigt sich sehr früh, wie konsequent man mit Kommentarzeichen (mal // und dann wieder {* oder nur {) und mit den Namenskonventionen umgeht. Das Tool parst beim Import jede Zeile und ist eine Art Prüfstelle, was den Codestil anbelangt.

Nicht dass MM Mühe bekundet, wenn die Einrückungen oder Zeilenumbrüche exotisch anmuten, jedoch bei Kommentar oder Compilerdirektiven wie auch bei Include-Dateien gibt es Fallen, die ich kurz andeuten möchte, indem Sie Folgendes beachten sollten:

- Kommentare sollten eine eigene Zeile erhalten
- Einheitlicher Stil von Kommentarzeichen
- HTML-, Script- oder SQL-Code immer trennen
- Keine Compilerdirektiven innerhalb einer Klassendeklaration wie:

```
  private
  {$IFDEF PATTERNS}
    myWS: IVCLScanner;
    myFac: T_RIOFactory;
  {$ENDIF}
```

- Form(dfm)-Dateien lassen sich nicht importieren
- Include-Dateien werden nicht automatisch importiert
- Deklarieren derVariablen in einer Zeile, nicht gemischt

```
Var  //bad
  I,
  J, K: Integer;
Var  //good
  I, J, K : Integer;
```

Schon der Einstieg zeigt eine wesentliche Eigenschaft von ModelMaker, zumindest wird diese Arbeitstechnik so empfohlen. Ein direktes Oberflächen-Prototyping oder die Formulargestaltung ist nach wie vor in Delphi oder .NET zu empfehlen. Sobald nicht visuel-

le Komponenten und nicht visuelle Units ins Projekt kommen, ist die Arbeit mit dem Modellieren und der späteren Codegenerierung aus ModelMaker heraus angesagt.

Die Form-Dateien mit *.pas importiert man nur aus Dokumentationsgründen, aus Designüberlegungen oder der Vollständigkeit halber, arbeitet aber nach wie vor mit den Forms in der Delphi IDE. Die restlichen Units entwirft und editiert man in MM und generiert (exportiert) den Code vollständig aus ModelMaker heraus nach Delphi.

Möglich ist auch ein Arbeiten mit dem Kommandozeilen-Compiler (dcc [Optionen] Dateiname [Optionen]), da man evtl. eine gewisse Zeit die IDE nicht benötigt und nur an der „Business-Logic“ herumwerkelt.

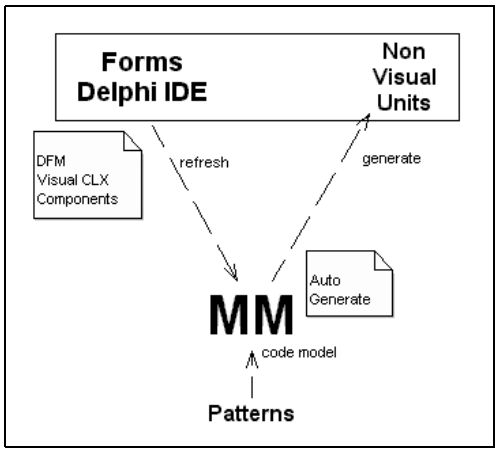

Abb. 1.36: Mit dem Editor/Generator in MM arbeiten

Demzufolge lässt sich ein ganzes Projekt direkt aus Delphi in die MM-Engine konvertieren (siehe Abb. 1.40). Alles Weitere, wie Diagramme Erzeugen, Code Generieren, Dokumentieren oder Refactoring Einleiten, geschieht mit MM. Es sei denn, man nimmt Änderungen an Forms, Formularen oder Masken vor, demzufolge ein Refresh wieder direkt aus Delphi zu ModelMaker erforderlich wird.

Diese Arbeitsstrategie gilt für viele CASETools mit Roundtrip Engineering!

Warum denn überhaupt Form-Dateien in ModelMaker importieren? Nochmals: Studien belegen, dass ein vollständiges Klassen- oder Paketdiagramm dem Entwickler und natürlich der Dokumentation mehr Übersicht und Stabilität verschaffen.

Diejenigen Dateien, welche man nicht aus ModelMaker generieren oder dokumentieren will, lassen sich dann immer noch gezielt deaktivieren. Jede Unit, die man nur in Delphi editieren will, erhält ein rotes Signal in ModelMaker.

Zudem ist ein vollständiges Modell meistens Garant dafür, das Release-Management besser in den Griff zu bekommen, da die MM-Engine jederzeit fähig ist, verschiedene Versionen einer Unit zu generieren und im Sinne eines Redesign die Versionen visuell zu testen. Wir sind nun mal visuelle Geschöpfe.

Als reiner OO-Entwickler wurde man in den letzten Jahren mit eingebauter Codegenerierung der Tools nicht gerade verwöhnt. Nicht zu unterschätzen ist deshalb die Möglichkeit, den Code aus den Modellen in Kombination mit dem leistungsfähigen Editor zu generieren und dann direkt den Kommandozeilen-Compiler zu aktivieren.

Versuche mit anderen Compilern wie FreePascal, VirtualPascal oder sogar mit Delphi-Script haben da durchaus ihre Berechtigung.

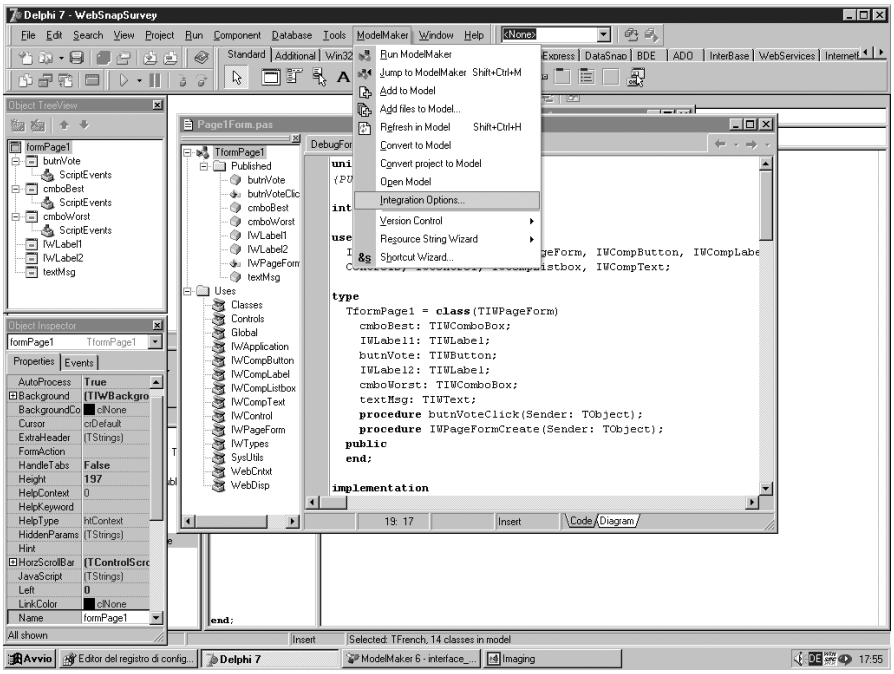

Abb. 1.37: ModelMaker automatisch integriert in Delphi 7

1.5.2 Modellbau

Im Weiteren sehen Sie ein Business-Objekt, das sozusagen als Referenz in ModelMaker dienen soll. Mit diesem Objekt will ich zeigen, dass sich mit UML auch bei kleinen Projekten ein durchgängiger Entwicklungszyklus mit Pattern-Generierung lohnt.

Die Anforderungen an unser Objekt sind bescheiden: Es soll uns ermöglichen, Lohnberechnungen und Änderungen durchzuführen und die Mutation der Daten als XML exportieren zu können. Vor allem sind wir an Lohnerhöhungen interessiert und dazu ist in der InterBase-Demo `IBLocal` mit den Tabellen `EMPLOYEE` und `SALARY_HISTORY` schon alles bestens vorbereitet. Man benötigt also nur den Code und kann direkt auf dem vorhandenen Datenmodell aufsetzen.

> *Die Analyse dient als Ausgangspunkt für die Definition von Anforderungen an eine künftige Systemumgebung beim Benutzer und als Basis für eine nachfolgen-*

de Klassenmodellierung sowie deren Umsetzung mithilfe der weiteren Diagramme in die Praxis.

Ich gehe exemplarisch von der Klasse `TDataToXML` aus und lasse sie dann aus dem Diagramm oder dem Editor generieren.

So erstelle ich Schritt für Schritt ein Klassendiagramm, lasse die Klassen zu Codefragmenten generieren und weise sie mithilfe des Paketdiagramms oder dem Unit-Generator direkt einer Unit zu. Denn die MM-Engine kann die Klasse erst generieren, wenn die physische Unit als File bekannt ist.

So etwas wie ein Klassen-File gibt es ja in keiner mir bekannten Sprache. Jeder Unit lässt sich übrigens in MM einen Alias zuweisen, das erleichtert die Suche und verhilft zu schnellerem Navigieren. Diese Aliase sind reine Zuweisungsnamen für einen Pfad, nicht wie in den Projektoptionen in Delphi, wo der Alias einen Namen neu zuordnet. Dies ist dann nützlich für die Abwärtskompatibilität, da man hier Alias-Namen für Units angeben kann, die in neueren Versionen einen anderen Namen haben oder zu einer Unit zusammengefasst sind.

Zurück zum Modellbau, ein Business-Objekt sollte folgende Kriterien erfüllen:

- Die Klasse erbt von einem Daten-Provider
- Die Abfragen sind autonom zur Klasse
- Logik und Validierung erfolgt in der Klasse
- Die Methoden sind Services
- Das Objekt ist unabhängig von visuellen Komponenten

1.5.3 Framework

Die Klasse `TDataToXML` hat ein Property `pBuffer`, das die Größe des Buffers definiert. Zudem besteht eine Abhängigkeit zur Klasse `TFStream`. Die entsprechenden getter und setter lassen sich von ModelMaker automatisch erzeugen. Somit wirkt sich eine Änderung von `pBuffer` sofort auf den getter `FpBuffer` und den setter `SetpBuffer` aus. Diesen Code verwaltet die Engine um bei jeglicher Änderung des Properties die geforderte Synchronisation zu garantieren.

```
Type
 TDataToXML = class (TObject)
 private
   FpBuffer: PChar;
   function getFieldStr(field: Tfield): string;
   procedure writeData(stm: TFStream; fld:TField; s:string);
   procedure writeRowEnd(stm: TFStream; isTitle: boolean);
   procedure writeRowStart(stm: TFStream; isTitle: boolean);
 protected
   procedure SetpBuffer(abuffer: PChar); virtual;
   procedure writeString(stm: TFStream; s:string); virtual;
 public
   constructor create;
```

```
   destructor destroy; override;
   procedure DatasettoXML(dSet: TDataset;
                          fileName: string); virtual;
   property pBuffer: PChar read FpBuffer write SetpBuffer;
  end;
```

Das gesamte Framework der Klassen besteht aus 5 Relationen, das im Klassendiagramm der Abb. 1.41 weiter unten im Klassenzauber ersichtlich ist:

Die Vererbung ist klar mit dem Business-Objekt von `TDataModule1` ersichtlich. Das Form wiederum hat eine Realisierung zur Schnittstelle `IVCLScanner`, die eigentliche Realisierungsklasse ist `TVCLScanner`. Die Aggregation <tblEmpl> erkennen wir als offene Raute, da hier eine Teil-Beziehung vorliegt. Jede Beziehung, außer der Assoziation, erkennt MM beim Einlesen durch den Wizard.

Als vierte Beziehung ist eine Assoziation im Diagramm ersichtlich. Die Assoziation <datasetToXML> ist eine Beziehung zur Laufzeit, d. h., die Klasse kennt die Instanz nicht, typischerweise sind es lokale Instanzen oder Referenzen, die aufgerufen und wieder zerstört werden, wie folgender Code zeigt:

```
with TDatatoXML.create do begin
try
  dataSetToXML(datEmployee.query1, 'salaryXport.xml');
finally
  free;
end
```

Eine Assoziation erkennt man als einfache Linie. Der Vollständigkeit halber gibt es noch als gestrichelte Linie die Abhängigkeit <dependency>, wobei die vor allem bei externen Typen einer Komponentenbibliothek ihren Einsatz findet, konkret die Abhängigkeit zu `TDataSet`, `TFileStream` etc. Wobei man genau genommen auf die entsprechende Klasse in der externen Unit verweisen sollte, dies würde aber jedes Klassendiagramm schnell unübersichtlich erscheinen lassen, deshalb die visuelle Reduktion auf so genannte Platzhalterklassen.

Die Bezeichnung Platzhalter, Abstrakte Klassen oder Interfaces sind in ModelMaker interne Unterscheidungskriterien für Typen, die aber beim Generieren keine Umwandlung ermöglichen, d. h., ich kann eine Abstrakte Klasse nicht in ein Interface wandeln, indem ich im Klasseneditor den Button ändere.

Active Modeling Engine

ModelMaker ist schnell und kompakt, diese Geschwindigkeit wird durch die parsergesteuerte „Modeling Engine“ erreicht. Diese Engine speichert und verwaltet alle Beziehungen zwischen den Klassen und deren Attribute. Das Umbenennen einer Klasse oder das Verändern eines Vorfahren aktiviert unmittelbar den Codegenerator, den man als Option in der Betriebsart automatisch oder manuell einstellen kann. Mit schnell meine ich auch die Ladezeit, die nur ein paar erstaunliche Sekunden beträgt.

Aufgaben wie das Hinzufügen eines Events, eines Propertys oder einer Zugriffsmethode werden als Skelette auf Mausklick hin angelegt. Diese hohe Übereinstimmung ermöglicht es, Änderungen in beide Richtungen zu aktualisieren und zusätzlich die Dokumentation zu gestalten.

> *Da Modelle proprietäre Informationen enthalten, die nicht im Quelltext enthalten sind, ist es wichtig, dass Sie die Modelldateien zusammen mit den Unit-Dateien in Ihre Speicher- und Versionskontrollprozesse aufnehmen.*

Die UML kennt mit Stereotypen und benutzerdefinierten Eigenschaften zwei Konzepte zur individuellen Erweiterung des Sprachumfangs. MM unterstützt beide Konzepte. Nützlich auch, weil Stereotypen und benutzerdefinierte Eigenschaften zusammen mit der Makrotechnik ermöglichen, aus den Modellen sehr differenziert und spezifisch Code zu generieren, z. B. das Schlüsselwort „deprecated“.

Jeder Anwender kann zusätzlich freie Referenzen zwischen beliebigen Modellelementen definieren. Einmal angelegt, pflegt das Tool diese Referenzen automatisch genauso wie alle Referenzen, welche die Engine standardmäßig vorsieht.

Im Folgenden ein Ausschnitt aus einer generierten oder importierten Unit-Datei, welche der Parser als aktive Klassen identifiziert und kompakt in der Unit aufführt:

```
  //strategy interface
MMWIN:CLASSINTERFACE TFinanzeCharge; ID=89;
  //Concrete Strategy
MMWIN:CLASSINTERFACE TRegularCharge; ID=90;
MMWIN:CLASSINTERFACE TPreferredCharge; ID=91;
MMWIN:CLASSINTERFACE TTrialCharge; ID=92;
  //Context Interface
MMWIN:CLASSINTERFACE TChargeContext; ID=93;
  //Concrete Context
MMWIN:CLASSINTERFACE TMonthlyCharges; ID=94;
```

Viele Merkmale einer Unit-Datei, wie Klassen-Implementierungen, sind durch die Verwendung separater Editoren eingerichtet. Diese Teile der Datei sind in der Vorlage durch Markierungszeilen gekennzeichnet, die mit MMWIN beginnen.

Diese Markierungszeilen soll man im Unit-Code-Editor nicht ändern (Sie dürfen diese jedoch innerhalb der Datei verschieben, solange sie intakt bleiben). Andere Zeilen, wie die uses-Anweisung, Typen oder Klassenmethoden der Unit, können Sie im Unit-Editor beliebig bearbeiten.

Wenn Sie ModelMaker zusammen mit der IDE benutzen, müssen Sie berücksichtigen, dass die IDE die Modelldateien nicht ändern kann. Alle Quelltextänderungen, die Sie mit den IDE-Editoren in Delphi durchführen, werden nicht automatisch in das Modell übernommen. Ihre Änderungen werden wieder verworfen, wenn ModelMaker den erzeugten Unit-Code das nächste Mal aktualisiert.

Die oben sichtbaren Klassen wie `TRegularCharge;ID=90;` sind deshalb Delphi unbekannt und die Engine erstellt diese bei jedem Import neu. Wenn Sie an einem bestehen-

den Modell Änderungen vornehmen müssen, sollten Sie ModelMaker statt Delphi verwenden, um auf diese Weise die Abstimmung von Modell und Code sicherzustellen.

Sollte das nicht möglich sein, dann dürfen Sie nicht vergessen, nach Beendigung Ihrer Änderungen die Unit wieder in das Modell zu importieren. Es besteht aber die Option, diese Updates gegenseitig in Form eines Round-Trip durch ModelMaker zu automatisieren, jedoch ist das eine Frage der Disziplin ;).

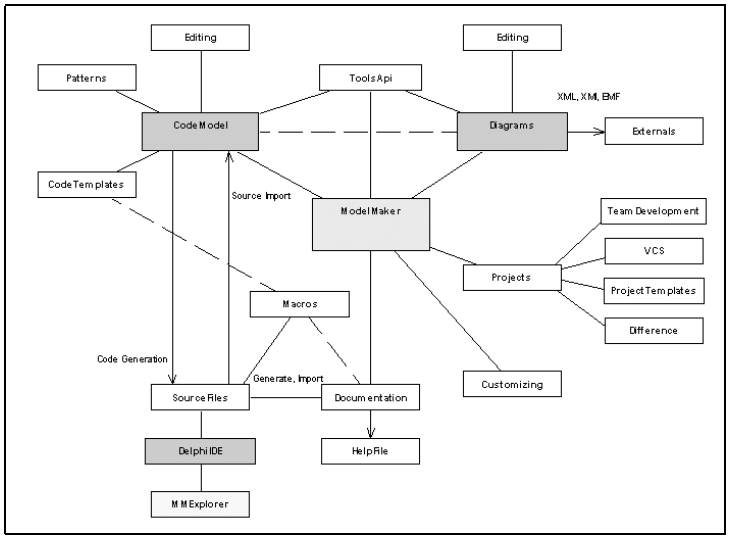

Abb. 1.38: Die ModelMaker-Funktionsblöcke auf einen Blick

Klassen und Diagramme sind aus der Sicht der Engine unterschiedliche Entitäten. Die Existenz einer Klasse bedeutet nicht gleichzeitig, dass ein Diagramm für diese Klasse existiert; Diagramme muss man explizit erstellen. Ebenso bewirkt das Löschen eines Diagramms nicht, dass damit eine Klasse oder eine Schnittstelle aus Ihrem Modell bzw. aus dem Code, der durch das Modell erzeugt wurde, gelöscht wird.

Als Generator bestehen nebst der Engine noch die Technik, Makros und Templates in bestehenden Code einzufügen, die sich sogar kombinieren lassen, wie folgendes Beispiel eines Templates zeigt.

> *Ein Template ist wie eine parametrisierbare Klasse oder Unit, die aber in MM keine Laufzeiteigenschaften hat, d. h., ein einmal aktiviertes Template ist eher als Konstruktionskopie und nicht als Link zu verstehen.*

Die spätere Modifikation des Templates aktualisiert die bereits generierten Codestrukturen nicht. Die Syntax <!FReferenceCnt!> steht für ein Makro:

```
TCodeTemplate = class (TObject)
  private
    <!FReferenceCnt!>: Integer;
  protected
```

```
    function <!Referenced!>: Boolean;
    procedure <!SetReferenced!>(IsReferenced: Boolean);
  public
    procedure <!AddReference!>;
    procedure <!RemoveReference!>;
end;
```

Die Grundlage dafür bilden die einzigartigen Funktionalitäten der Makros, die sich auch mit Laufzeiteigenschaften in Units einsetzen lassen. Dazu beschreibt man ein Unit Template als Muster. Im Anschluss daran kann man das Muster beliebig häufig mit geringem Aufwand aktivieren und immer wieder für neue Units einsetzen.

Die so generierten Templates können sowohl den Ausgangspunkt für ein neues Gesamtmodell bilden oder sind auch in ein bereits bestehendes Modell integrierbar (siehe Abb. 1.40). Unterstützend zur Codegenerierung gibt es den Critic Manager in Abb. 1.39, der eine Art Software-Metrik bereitstellt.

Zu prüfen ist z. B., ob ein Use Case mindestens einen Actor hat, ein Interface von mindestens einer Klasse unterstützt ist, eine Klasse mehrere Links oder kein Property hat, ein Sequenzdiagramm eine Transition ohne zugehörige Methode besitzt oder das Paketdiagramm keine Dokumentation innehat:

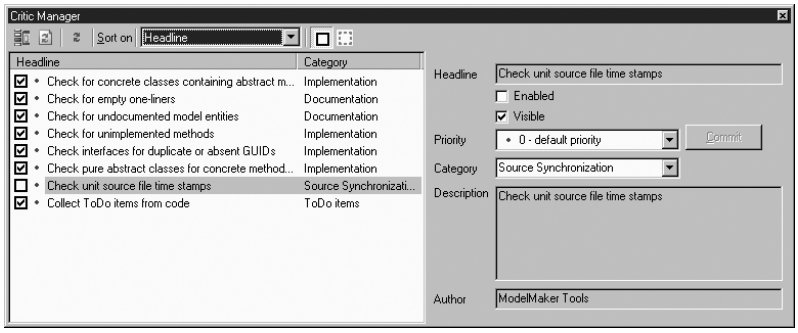

Abb. 1.39: MM mit dem Critic Manager

Generatoren

Häufig werden Sie ein Modell aus Units anfertigen müssen, die nicht mit ModelMaker erstellt wurden. Dazu gibt es in der Symbolleiste zwei Schaltflächen, die das Importieren von Quelltext in Ihr Modell ermöglichen. Nach dem Importieren übernimmt das Tool die Leitung. Auch den Codegenerator kann man in den Entwicklungsprozess so integrieren, um zu überwachen, steuern, entwerfen, dokumentieren und mit einem Refactoring den Code zu verbessern.

Da sich MM vollständig in Delphi integrieren lässt und trotzdem nicht zu kompliziert ist, ist der Lernaufwand gering. MM hat in der aktuellen Version 7.1 folgende neue Leistungsmerkmale:

- Multilevel Undo/Redo im Diagramm-Editor
- Qualifizierte Relationen zwischen den Symbolen
- UML Real Time Notation
- Robustnessdiagramme
- MMToolsApi mit neuen Schnittstellen
- Erweiterter Editor mit Syntax-Farbwahl

Ein wirklich gut durchdachtes GUI, das die nötigen Ansichten strukturiert und in mehreren Bereichen darstellt, hilft der Codegenerierung. Die Sichten sind in einem Kontext zueinander kombinierbar, d. h., zu den Files gehört die Unit-Sicht, zu den Klassen der Code-Editor und zum Diagramm der Diagramm-Editor mit den Symbolen. Die einzelnen Member der Klasse sind in Abb. 1.40 und 1.41 unten links ersichtlich, die dann jeweils durch einen aktiven Link zu den erwähnten Sichten ihre Member wechseln.

An Diagrammen sind seit der Version 6.2 nun alle Diagrammtypen vorhanden. Das Paketdiagramm glänzt, wie erwähnt, mit einer umfangreichen Analyse der Units, der mit dem zugehörigen Wizard alle Abhängigkeiten auf der Stufe Interface und Implementation grafisch aufbereitet. Nur diese eine Funktion macht das Tool schon beinahe unentbehrlich.

Die große Stärke liegt beim Reverse Engineering darin, ein Klassendiagramm oder Package zu erstellen. In Abb. 1.40 hat die Engine direkt von Delphi das Projekt ins Modell konvertiert. Einzelne Typen und Elemente besitzen umfangreiche Filter und gezielte Codegenerierung, die sich vom Editor aus nach Delphi steuern lassen.

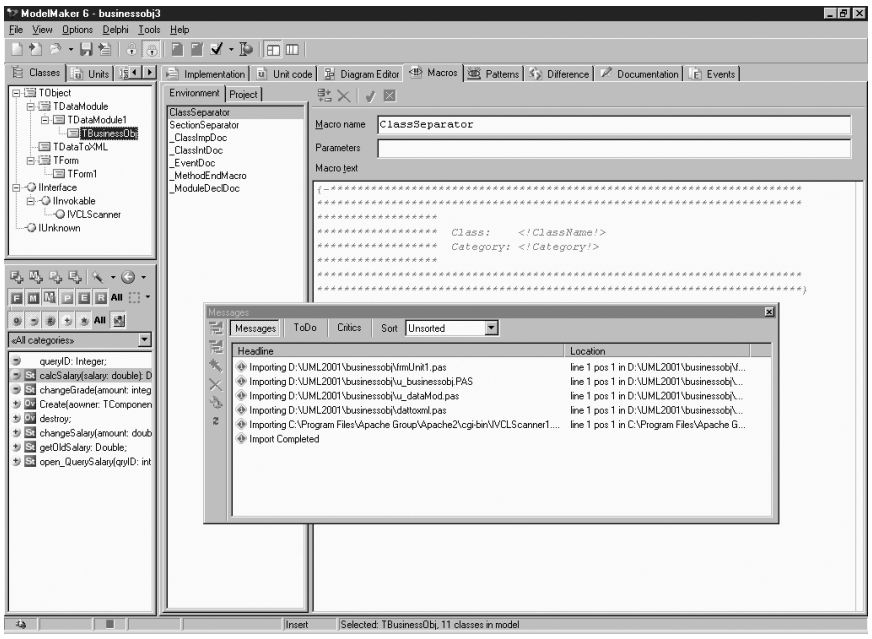

Abb. 1.40: Umfangreiches Reverse Engineering bei bestehenden Projekten

1.5.4 Klassenzauber

Weiter geht es nun mit dem Erstellen des Klassendiagramms. Hier spielt es keine Rolle, ob man die Klasse importiert oder direkt in MM modelliert. Der „Diagram Wizard“ in Abb. 1.42 ermöglicht es, die zugehörigen Relationen zu identifizieren und einen ersten Wurf darzustellen. Meistens bedarf das Diagramm noch der Nacharbeit, zumal der Parser weitere Stereotypen oder Link-Klassen nicht automatisch erkennen und benennen kann.

Ist eine Klasse mit eigenen Zustandsvariablen bestückt oder will man die möglichen Ereignisse mit ihren Übergängen genauer festhalten, dann sind State Events als weitere Diagrammart erste Wahl. Natürlich kann der Code auch aus dem Diagramm entstehen, somit hat unsere Klasse `TDataToXML` in Abb. 1.41 eine stete, synchrone Verbindung zum Code erhalten.

Die Klassenansicht in Abb. 1.40 links oben zeigt eine hierarchische Auflistung aller im Modell enthaltenen Klassen und Schnittstellen. Außerdem stellt sie die Vorfahren von Klassen und Schnittstellen des Modells dar. Vorfahren, die im aktuellen Modell enthalten sind, haben mit durchgezogenen Linien umrahmte Symbole. Symbole von Vorfahren, die nicht im aktuellen Modell enthalten sind, sind von gestrichelten Linien umgeben.

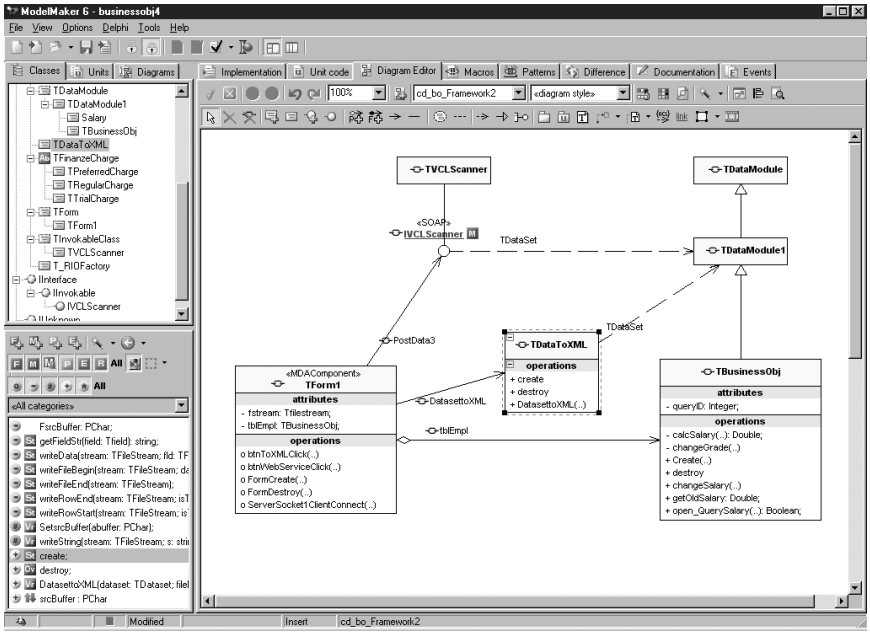

Abb. 1.41: Die 5 Relationen auf einen Blick

Ich bin jetzt im Zustand der abgeschlossenen Details, das nächste Diagramm nimmt schon Teile der Implementation vorweg und bewegt sich in Richtung Integration. Die Rede ist vom Sequenzdiagramm. Das Diagramm beschreibt das Zusammenwirken verschiedener Objekte für einen bestimmten Anwendungsfall. Es definiert somit die interne

Sicht eines Use Case, beschreibt also, wie sich der Use Case innerhalb der Anwendung realisieren lässt.

Diagramme werden häufig als Entwurfswerkzeug für Klassen verwendet. Sie können einem Diagramm Eigenschaften und Methoden hinzufügen, was zu Änderungen im Modell und folglich auch zu Änderungen am Quelltext führt. Nach der Entwurfsphase können Sie die Werkzeuge im Editorenbereich verwenden, um die Implementierungen für Ihre neue Klasse einzufügen.

Als nächsten Schritt komme ich zum Paketdiagramm, das ich am besten mit dem sagenhaft durchdachten Unit Analyzer vorstelle.

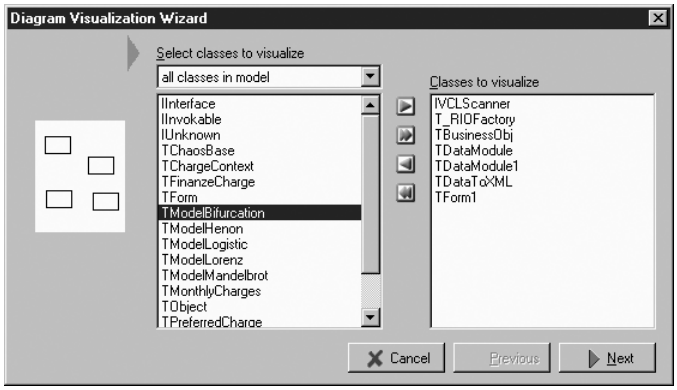

Abb. 1.42: Der Zauberlehrling in Aktion

1.5.5 Unit Analyzer

Manstelle sich vor, sich einen Überblick über die VCL oder die CLX verschaffen zu wollen. Oder sich die ganze Laufzeitbibliothek von Win32 oder die SOAP-Sourcen des WebService Package darstellen zu lassen. Hier kommt der „Unit Analyzer“ aus Abb. 1.42 ins Spiel, der nicht nur eine hervorragende Analyse mit zweifachen uses-Referenzen (`interface` und `implementation`) erstellt, sondern als Zugabe ein komplettes Paketdiagramm mit den Abhängigkeiten druckfertig aufbaut!

Jedes generierte Diagramm ist einzeln konfigurierbar bezüglich Suchpfad, Tiefenanalyse, und der Art der visuell gezeigten Klassen in den Paketen. Nachdem der Scanner die Abhängigkeiten analysiert hat, werkelt sogar ein kleiner interaktiver Experte, um die optimale Darstellung zu finden. Achten Sie aber darauf, reine Typisierungsunits wie `Variants` oder `Types`, die fast alle Units benötigen, zu löschen, da sonst ein grafisches Spinnennetz entsteht.

Die Units-Ansicht zeigt ein Baumdiagramm oder eine Liste mit allen im Projekt enthaltenen Units an. Darüber hinaus zeigt diese Ansicht alle Objekte, Schnittstellen und Ereignisse, die in den einzelnen sichtbaren Units enthalten sind.

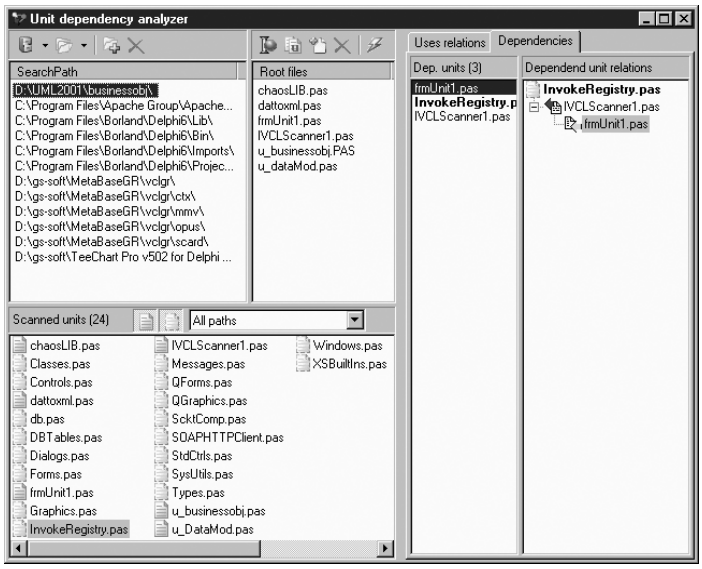

Abb. 1.43: Der Scanner hat gewirkt

1.5.6 Pattern Generator

Bei den eingebauten Design Patterns, die sogar musikalisch wirken können ;)[xviii], bildet MM nicht nur eine statische Struktur oder ein Code-Fragment ab, sondern es besteht ein aktiver Link von Design zum Code, der unmittelbar reflektiert wird. Gemeint ist die Unterstützung von Design Patterns im bestehenden und geplanten Code.[10]

Was meint nun laufender Code. Angenommen es besteht eine Instanz einer Klasse. Nach weiterer Analyse kommt man zur Einsicht, die Instanz darf nur einmal existieren. Nun ist MM in der Lage, den Aufruf mit einem Singleton Pattern zu erweitern, um somit die einzig mögliche Existenz dieser Klasse sicherzustellen.

Auch kompliziertere Patterns, wie der Decorator, lassen sich durch die MM-Engine im Sinne eines Redesign generieren [xix], sodass Strukturänderungen auch nach einem Review noch möglich sind. Die vorhandenen Patterns sind in einem eigenen Register untergebracht und setzen auch einen separaten Editor ein, der die generierten Patterns noch bearbeiten lässt.

Um das Arbeiten mit Patterns maschinell zu unterstützen, ist zweierlei nötig: ein Musterkatalog, also eine normierte, möglichst umfangreiche Sammlung von Mustern, und ein Mechanismus zur Übernahme von Mustern aus dem Katalog in den Kontext eines konkreten Modells. Beides bietet ModelMaker an.

Als folgendes Beispiel, siehe Abb. 1.44, dient ein Lock Pattern, welches folgenden Code so weit als möglich einwandfrei generiert hat:

[10] Patterns und Code sind mit Harmonien und Noten vergleichbar.

```
TTrialCharge = class (TFinanzeCharge)
  private
    FLockCnt: Integer;
  protected
    function Locked: Boolean;
    procedure SetLocking(Updating: Boolean);
  public
    function getCharge(const Balance: double):Double; override;
    procedure Lock;
    procedure UnLock;
end;

procedure TTrialCharge.Lock;
begin
  Inc(FLockCnt);
  if FLockCnt = 1 then SetLocking(False);
end;

function TTrialCharge.Locked: Boolean;
begin
  Result:= (FLockCnt <> 0);
end;
```

> *Klassische Design-Methoden konzentrieren sich hauptsächlich auf den funktionalen Bereich von Software. Für nicht funktionale Eigenschaften wie Wartbarkeit oder Erweiterbarkeit kann nur der Designer selbst durch seine Erfahrung sorgen.*

Genau hier setzen Patterns auch in ModelMaker an. Einerseits bieten sie funktionale Lösungen für wiederkehrende Design-Probleme und andererseits bringen sie nicht funktionale Eigenschaften in das Design ein.

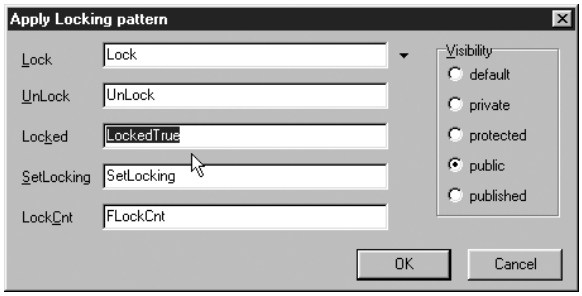

Abb. 1.44: MM beim Erzeugen des Lock Pattern

1.5.7 Refactoring

Da Modell und Code für MM eins sind, ergeben sich einige Refactoring-Techniken ganz nebenbei. Ein Beispiel ist das Umbenennen von Klassen und Methoden oder das Trennen eines Paketes. Nimmt man eine solche Änderung am Code vor oder modifiziert man das Modell entsprechend, wird die jeweils andere Seite automatisch aktualisiert.

Refactoring, d. h. die Umstrukturierung einer Anwendung, ohne seine Funktionalität zu verändern, ist ein wesentlicher Bestandteil der täglichen Arbeit. Durch meist kleine Schritte, wie Variablen umbenennen, Felder kapseln, Parameter extrahieren oder Vererbung durch Delegation ersetzen, wird der Code kontinuierlich klarer und lesbarer, d. h., ein Code sollte sich zunehmend selbst dokumentieren. Demzufolge sind auch die Möglichkeiten von Umstrukturierung in MM beachtlich.

Vielfach will man im Zuge der Paketbildung eine Klasse von einer Unit in die andere Unit verschieben. Oder einige zentrale Methoden müssen sich in einer neuen Klasse wiederfinden, d. h., man lagert sie in eine eigene Service-Klasse aus.

Es ist z. B. nötig, eine Klasse näher an eine bestehende andere Klasse anzubinden, d. h., wir verschieben eine Methode von KlasseA nach KlasseB und ändern den dazugehörigen Konstruktoraufruf. Da die Engine die internen Abhängigkeiten zwischen Interface und Implementation von Klassen kennt, ergibt sich schnell eine neue, „bessere“ Version. Was aber nicht automatisch vor sich geht, ist das Verschieben der uses-Referenzen und globalen Typen innerhalb der Units, da benötigt man eben noch den Kopf ohne Header ;).

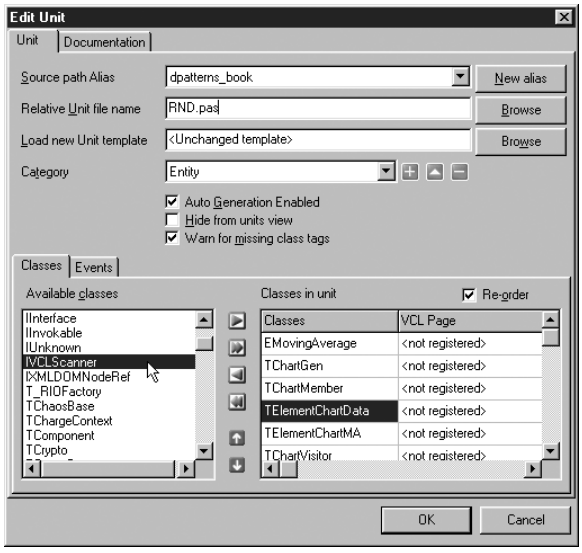

Abb. 1.45: Refactoring in ModelMaker

1.6 Refactoring

1.6.1 Redesign

Als Redesign betrachtet man allgemein ein größeres Umstellen von Code innerhalb eines Projektes. Es ist auf der Stufe von Packages anzusiedeln und hat teilweise Einfluss auf das Systemverhalten bis hin zu Änderungen in der Ergonomie.

> *Dies steht im Gegensatz zum Refactoring, das ein Vereinfachen und Stabilisieren des Code bewirken soll und keine Änderung auf das „beobachtbare" Verhalten der Applikation bewirkt.*

Hier gerät man zuweilen in Argumentationsnotstand, wenn man dem Kunden erklärt, drei Wochen Refactoring-Aufwand ohne sichtbare Ergebnisse reduzieren dafür im Gegenzug die künftigen Fehler- und Wartungskosten!

Änderungen und immer wieder neue Anforderungen während des Entwicklungsprozesses machen Software immer komplizierter. Gut, wenn man mit dem eingesetzten Tool die Architektur des Systems dann überarbeiten kann, ohne dass sich das nach außen sichtbare Verhalten der Anwendung ändert.

Dieses so genannte Refactoring unterstützen mittlerweile einige Tools. Ziel eines Refactoring sind robuste und stabile Systemarchitekturen mit klar gegliederten Einheiten von Paketen und Komponenten. Da Modell und Code für diese Tools eins sind, fallen einige Refactoring-Methoden ganz nebenbei ab.

Ein Beispiel ist das Umbenennen von Klassen und Methoden. Nimmt man eine solche Änderung am Code vor oder ändert man das Modell entsprechend, aktualisiert das Tool die jeweils andere Seite automatisch. Typische Redesign/Refactoring-Techniken, die auch in einem Review zum Tragen kommen, sind:

- Interface extrahieren und bilden
- Superklasse extrahieren
- Klasse und Unterklasse extrahieren
- Klassen in Pakete verschieben
- Pakete aufbrechen und entkoppeln

Für den Architekturentwurf bietet die UML mit Paketdiagrammen sehr weit reichende Unterstützung an: Die einfache Darstellung hilft Ihnen, IT-Systeme zu gliedern, Komplexität zu reduzieren und so robuste Systemarchitekturen zu bauen. In den Diagrammen definieren Sie grafisch die Abhängigkeiten und Hierarchiebeziehungen zwischen Paketen. Auf diese Weise kann man unter anderem vorgeben, welche Schnittstellen eines Paketes sich benutzen lassen und welche zu implementieren sind.

Am effizientesten ist ein Redesign, wenn die meisten Eigenschaften einer Klasse als Property angelegt sind. Dies ermöglicht ein gezieltes Ändern des Propertys selbst, sodass die abhängigen Methoden sich gekoppelt zwischen den Klassen verschieben lassen. Properties sind die augenfälligsten Teile von Komponenten und zugleich diejenigen, die Anwendern und Entwicklern, die mit den Komponenten arbeiten, den größten Nutzen bringen. Die Deklaration eines Propertys enthält drei Dinge:

- Den Namen des Propertys
- Den Typ des Propertys
- Methoden für das Lesen und/oder Schreiben des Propertywertes

Public Properties sind dann über wohl definierte Methoden veränderbar. Diese Methoden müssen bei einer Java-Bean-Komponente oder in Delphi die Form setXX() und getXX() haben. Hierbei handelt es sich für gewöhnlich um simple Attribute eines Objekts, wie zum Beispiel Farbe, Koordinaten oder Schriftart, bis hin zu komplizierten Instanzen innerhalb einer Klasse (Delegation).

1.6.2 Dekomposition

Das Hauptproblem bei jedem Design ist, eine Struktur zu finden, die flexibel (also innen kompliziert) und gleichzeitig stabil und robust sein soll. Der Einsatz vieler Patterns erhöht am Anfang eher die Kompliziertheit und suggeriert deshalb eine gewisse Anfälligkeit, die Übersicht zu verlieren. Dekomposition ist ein Abwägen, welche Klassen oder Packages man deshalb wieder entkoppelt und damit eine weitere Modularisierung der Struktur herbeiführt oder im ob man Zuge des Refactoring ein Interface extrahiert, d. h. auch wieder eine Dekomposition durchführt.

Wobei hier eindeutig ein Zielkonflikt besteht. Je mehr Klassen man erhält, desto übersichtlicher die einzelnen Methoden, aber die Gesamtübersicht wird bei erhöhter Anzahl von Klassen und Packages auch nicht besser. Die Praxis bestätigt aber: Besser mehr Klassen in den Paketen als zu viele Methoden in den jeweiligen Klassen.

Da mit der Zeit Patterns allgemein unter den Entwicklern bekannt sind, gewinnt man mit ihrer Anwendung die Übersicht wieder zurück. Dekomposition soll eine zu starke Interaktivität in den Modulen vermeiden und zu große, schwere Schnittstellen vereinfachen. Durch die Befolgung des Gesetzes von Demeter können Entwickler eine Dekomposition überprüfen, d. h., wenn Demeter nicht eingehalten wird, muss die Dekomposition nochmals überdacht werden.

> *Das Gesetz von Demeter besagt, dass ein Objekt O als Reaktion auf eine Nachricht m weitere Nachrichten nur an die folgenden Objekte senden sollte:*

1. an Objekt O selbst
   Bsp.: `self.letPrimeStatistics(vdata,mn,std);`
2. an Objekte, die als Parameter in der Nachricht m vorkommen
   Bsp.: `objCData.AcceptMemberVisitor(Visitor);`
3. an Objekte, die O als Reaktion auf m erstellt
   Bsp.: `visitor:= TChartVisitor.create(cData, madata);`
4. an Objekte, auf die O direkt mit einem Member zugreifen kann
   Bsp.: `perfList.add(period);`

Bau eines Business-Objekts

Wir bauen uns nun ein Business-Objekt, das sozusagen als Referenz zu einem Streifzug des Refactoring dienen soll. Einige Tipps zu Klassenbau und der visuellen Darstellung folgen. Mit diesem Business-Objekt will ich zeigen, wie sich mit Refactoring-Techniken auch bei kleinen Projekten ein durchgängiger Entwicklungszyklus lohnt.

Die Anforderungen an unser Objekt sind bescheiden: Es soll uns ermöglichen, Lohnberechnungen und Änderungen durchzuführen und die Mutation der Daten als XML exportieren zu können. Vor allem sind wir wieder an Lohnerhöhungen interessiert. Ich erzeuge also ein Refactoring am bekannten Lohnmodell, das beim Pattern Tool schon Pate stand.

Als Erstes nehmen wir die Geschäftsprozesse auf (Haupt- und Teilgeschäftsprozesse abgrenzen, Schnittstellen mit In- und Output und deren Protokolle zwischen Prozessen festlegen, Geschäftsprozesse in einem Ist-Modell abbilden).

Als Soll-Modell dient uns das zweite UML-Diagramm das Activity. Die Ergebnisse der Analyse dienen als Ausgangspunkt für die Definition von Anforderungen an eine künftige Systemumgebung beim Benutzer und als Basis für eine nachfolgende Klassenmodellierung sowie deren Umsetzung mithilfe der weiteren Diagramme in der Praxis der Implementierung.

So z. B. kennen Sie das Problem, das man eine Aktivität von einer anderen abhängig macht. Bei einer Abhängigkeit im Activity dient dazu der Synchronisationsbalken, der die einzelnen Bedingungen oder Trigger aufnehmen kann. Ein einfacher Synchronisationsbalken bedeutet, dass die ausgehenden Trigger erst in Aktion treten, wenn alle eingehenden vorhanden sind.

- Dies entspricht einer Und-Bedingung. In unserem Beispiel in Abb. 1.46 wird die Aktivität <get Employee> oder <check Contract> erst möglich, wenn die Bedingung [valid] der Aktivität <login> erfüllt ist.
- Die Aktivität <manage Salary> wiederum kann eintreten, wenn entweder der Angestellte auf der Lohnliste ist oder er einen Vertrag akzeptiert hat. Dies entspricht einer Oder-Bedingung, erkenntlich am direkten Einfließen der zwei Pfeile (Trigger) in die Aktivität <manage Salary>.
- Der abschließende Export in ein XML-File kann nur erfolgen, wenn die Mutationen gespeichert sind. Somit sollte jeder Trigger auch eine Bedingung enthalten, welche in UML als eckige Klammer, wie z. B. [saved], notiert wird.

Klassenbau

So, fürs Erste haben wir den Geschäftsprozess analysiert und die Anforderungen ungefähr abgesteckt. Im weiteren Verlauf gehen wir von der Aktivität <manage Salary> aus und spezifizieren daraus eine erste Klasse.

Es ist aber nicht so, dass jede Aktivität eine Klasse für sich im Hintergrund hat, es können auch n-Aktivitäten in eine Klasse münden, d. h., sie teilen sich die Klasse. Eine Heuristik besagt, dass eine Aktivität in der Regel bis zu drei Methoden einer Klasse ergibt. Hier sind eben schon erste Ansätze der „richtigen“ Klassengröße auszumachen. Bis zu diesem Zeitpunkt hat noch kein Refactoring stattgefunden, da ich noch in der Analysephase stecke.

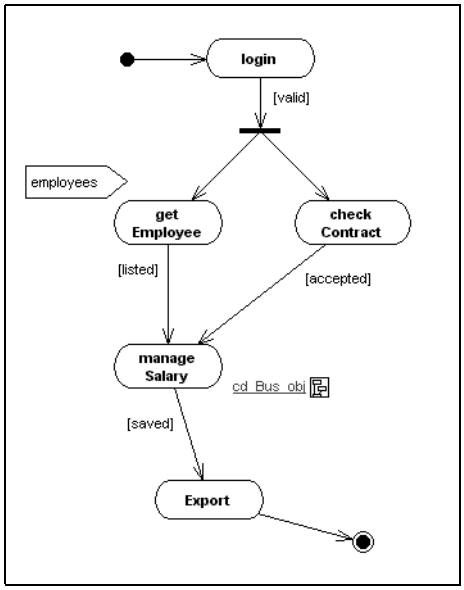

Abb. 1.46: Das Activity mit den Bedingungen

Ein Business-Objekt sollte dann folgende Kriterien erfüllen:

- Die Klasse erbt von einem Daten-Provider
- Die Abfragen sind Teil der Klasse oder haben eine externe Referenz
- Logik und Berechnung erfolgt in der Klasse
- Die Methoden erscheinen als public Services
- Das Objekt ist unabhängig von visuellen Komponenten

Ich komme zum Klassendiagramm. Lassen Sie uns das Beispiel in Object Pascal kurz diskutieren (erben, Abfragelogik, Autonomie). Die Klasse `TBusinessObj` hat zwei öffentliche Methoden, die das Öffnen der SQL-Menge und das Ändern der Lohnsumme erlauben. Im Weiteren ist die Berechnung der Lohnsumme, z. B. Angabe in Prozenten, eine „private“ Angelegenheit, welche mit der Funktion `calcSalary` erfüllt ist:

```
TBusinessObj = class (TDataModule1)
  private
    function calcSalary(salary: double): Double;
    procedure changeGrade(amount: integer);
  public
    constructor Create(aOwner :TComponent); override;
    destructor destroy; override;
    procedure changeSalary(amount: double);
    function getFullName: string;
    function getOldSalary: Double;
    function open_QueryAll: Boolean;
```

```
    function open_QuerySalary(qryID: integer): Boolean;
end;
```

Bei einer Änderung der Lohnsumme mit der Funktion `changeSalary`(), in unserem Falle das InterBase-Feld SALARY, wird im gleichen Atemzug noch eine Berechnung anhand einer Business-Rule durchgeführt. Die Query selbst wurde vorgängig mit der Prozedur `open_QuerySalary`() geöffnet und dem Business-Objekt vererbt, da das Objekt selbst von `TDataModule` erbt.

Solche Queries sind ein häufiger Grund für ein Refactoring, da eine erste Variante, das SQL-Statement intransparent im dfm-File zu beherbergen, wenig verständlich ist und keinen wartbaren Code darstellt; besser ist die zweite Variante mit dem transparenten Einlagern des Statements in den Code selbst oder in eine eigene Unit:

```
function TBusinessObj.open_QuerySalary(qryID: integer):Boolean;
begin
  result:= false;
  with query1 do begin
    try
      SQL.Clear;
      SQL.add('SELECT * from EMPLOYEE');
      SQL.add('WHERE EMP_NO =:PClient_acc');
      params[0].name:='PClient_acc';
      params[0].dataType:=ftInteger;
      params[0].AsInteger:= qryID;
      Open;
      result:= true;
    except on EDataBaseError
          do showmessage('Salary data not found');
    end;
  end;
end;
```

Das Objekt `TBusinessObj` ist auch für die Instanzierung des `TDataModule` verantwortlich, sodass im Erzeuger zur Laufzeit mit `inherited create(aOwner); database1.open;` die Datenbank geöffnet wird. Nach dem Öffnen des nun transparenten Query lässt sich jetzt die neue Lohnsumme mit `changeSalary`() schreiben, die wiederum `CalcSalary` aufruft:

```
procedure TBusinessObj.changeSalary(amount: double);
begin
  with query1 do begin
    try
      edit;
       FieldByName('SALARY').asFloat:= calcSalary(amount);
      post;
    except on E: Exception
```

```
        do showmessage('Salary out of range' + E.message);
    end;
  end
```

`CalcSalary` alleine ist nicht für die ganze Regel verantwortlich. Bei globalen Regeln, die jeden Client betreffen müssen, fährt man am besten mit einer zentralen Stored Procedure oder einer Check-Constraint.

Im Falle der IBLocal-InterBase-Datenbank wird die eingegebene Lohnsumme auf dem Server validiert, d. h., die Summe sollte sich in einem bestimmten Min./Max.-Bereich befinden, der sich anhand von Job-Grade und COUNTRY ermitteln lässt:

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT INTEG_30
  CHECK ( salary >= (SELECT min_salary FROM job WHERE
       job.job_grade = employee.job_grade AND
       job.job_country = employee.job_country) AND
  salary <= (SELECT max_salary FROM job WHERE
       job.job_grade = employee.job_grade AND
       job.job_country = employee.job_country))
```

Bis zum jetzigen Zeitpunkt hat noch kein Export der Daten stattgefunden. Diesen wollen wir jetzt als XML-Format realisieren und hier sind Streams jederzeit willkommen. Während des Klassendesign starten wir meistens mit den public-Methoden, die dann mithilfe von CRC-Karten auch die privaten Funktionen hervorbringen.

Dieses iterative Spiel führt dann zu den nötigen Methoden. Um Klassen wieder verwenden zu können, sollten sie möglichst wenig mit anderen Klassen kommunizieren, denn je weniger eine Klasse von ihrer Umgebung weiß, desto universeller ist sie einsetzbar (Autonomie).

Unsere Klasse `TDataToXML` hat nur ein Property `pBuffer`, das die Größe des Buffers aufnimmt. Zudem besteht eine Abhängigkeit zur Klasse `TFileStream`. Die entsprechenden getter und setter erzeugt ModelMaker automatisch. Andere Tools hatten beim Erzeugen dieser Klasse schon mehr Mühe, auf die richtige Konfiguration und das Instrumentalisieren kommt es eben an:

```
TDataToXML = class (TObject)
private
  FpBuffer: PChar;
  function getFieldStr(field: Tfield): string;
  procedure writeData(stm: TFileStream; fld:TField; s:string);
  procedure writeFileBegin(stm: TFileStream; dSet: TDataSet);
  procedure writeFileEnd(stm: TFileStream);
  procedure writeRowEnd(stm: TFileStream; isTitle: boolean);
  procedure writeRowStart(stm: TFileStream; isTitle: boolean);
protected
  procedure SetpBuffer(abuffer: PChar); virtual;
  procedure writeString(stm: TFileStream; s:string); virtual;
```

```
public
  constructor create;
  destructor destroy; override;
  procedure DatasettoXML(dSet: TDataset;
                         fileName: string); virtual;
  property pBuffer: PChar read FpBuffer write SetpBuffer;
end;
```

Das Business-Objekt mit dem zugehörigen Export-Objekt ist nun als Dekomposition vom Ganzen zum Einzelnen erstellt. Das Diagramm in Abb. 1.47 ist nun fähig, die Code-Rümpfe inklusive Parametrisierung zu erstellen. `TForm1` besitzt den Member `datEmployee` vom Typ `TBusinessObj`. Diese Beziehung wurde als <aggregation> modelliert, da `datEmployee` immer auch auf das Datenmodul angewiesen ist und zur Designtime bekannt ist.

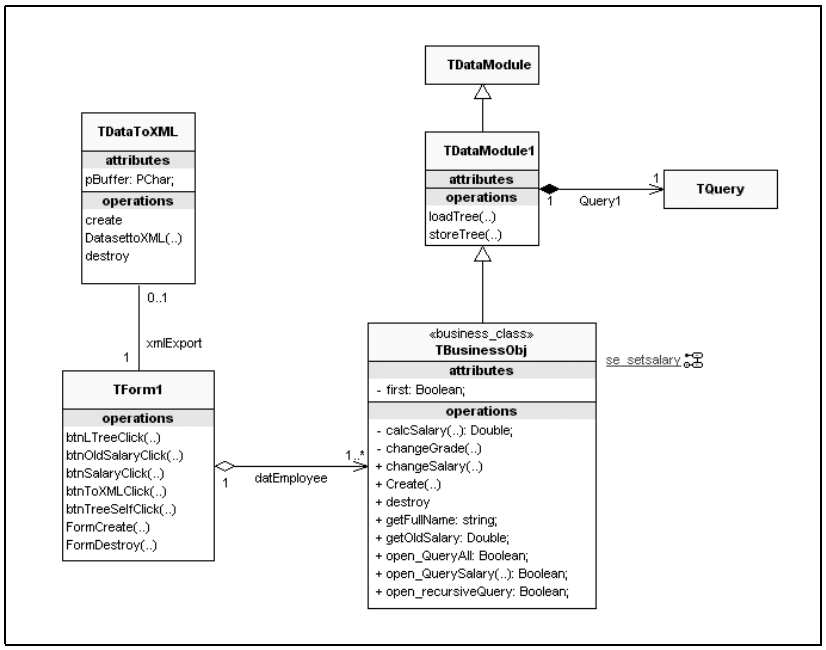

Abb. 1.47: Alle Relationen im CD auf einen Blick

Die <composition> mit dem Namen `Query1` ist stärker als <aggregation> und meint physikalische Einbindung, in unserem Fall `datEmployee.dataSource1.dataSet`, `datEmployee.query1.SQL` oder `cmxEmployee.items.add`, meistens in Komponenten oder Kollektionen zu finden und als schwarz gefüllte Raute notiert.

Eine Komposition bedeutet auch eine starke Abhängigkeit von einem anderen Objekt. Folgender Code demonstriert diese Abhängigkeit und füllt zur Laufzeit eine Combobox mit den SQL-Daten als Auswahlmöglichkeit in einer Combobox:

```
with datEmployee.dataSource1 do begin
  while not dataSet.EOF do begin
    cmxEmployee.items.add((dataSet.fieldValues['EMP_NO']));
    dataSet.next;
  end;
end;
```

Es sind also mittlerweile vier Relationen im Klassendiagramm enthalten: Die Vererbung ist klar mit dem Business-Objekt von `TDataModule1` ersichtlich. Das Datenmodul wiederum hat eine Composition zu `TQuery` als geschlossene Raute notiert. Die Aggregation erkennen wir als offene Raute, da die Instanz `datEmployee` zur Designtime als Attribut bekannt ist.

Als vierte Beziehung ist eine Assoziation im Diagramm ersichtlich. Die Assoziation <xmlExport> ist eine Beziehung zur Laufzeit, d. h., die Klasse kennt die Instanz nicht zur Designtime, typischerweise sind es lokale Instanzen, die aufgerufen und wieder zerstört werden, wie folgender Code zeigt:

```
with TDatatoXML.create do begin
try
  dataSetToXML(datEmployee.query1, 'salaryXport.xml');
finally
  free;
end
```

Eine Assoziation erkennt man als einfache Linie. Der Vollständigkeit halber gibt es noch als gestrichelte Linie die Abhängigkeit <dependency>, wobei die eher schwach (allgemein) definiert ist, und das Interface mit <realise>. Zusammenfassend lässt sich nach der Dekomposition folgende Tabelle aufstellen:

	Beziehung auf Stufe	**Bekannt zur**
Vererbung	Klasse	Designtime
Realisierung	Interface/Klasse	Design-/Runtime
Assoziation	Objekt	Runtime, Loadtime
Aggregation	Objekt	Designtime
Komposition	Mind. 3 Objekte	Designtime

Tab. 1.7: Alle Relationen zwischen Klassen

Weiter geht es mit einer genaueren Definition des Business-Objekts mithilfe eines Zustandsdiagramms. Beim State Event handelt es sich um eine detaillierte Sicht der Klasse. Jedes Zustandsdiagramm ist also einer Klasse zugeordnet und beschreibt deren Verhalten genauer.

Zwischen den Zuständen feuern Ereignisse als Nachrichten, oder kurzzeitige Aktionen werden ausgelöst. Ist eine Klasse mit eigenen Zustandsvariablen bestückt oder will man die möglichen und gültigen Ereignisse mit ihren Übergängen genauer festhalten, dann sind State Events die erste Wahl:

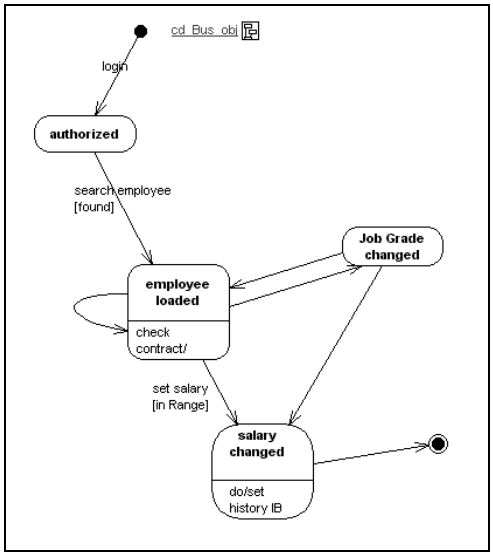

Abb. 1.48: Die erlaubten Zustandswechsel des Objekts

Interessant ist die Transition von <employee loaded> zu <salary changed>, die nur erfolgt, wenn der Betrag gemäß der Constraint im erlaubten Bereich liegt. Eine Ausnahmebehandlung wird nicht modelliert.

> *Einerseits ist zu erwarten, dass die Verwendung von State Events die Entstehung bestimmter Folgefehler reduziert; andererseits stellt sich die Frage, ob durch die spezifischen Mechanismen der Zustandsvariablen, wie dem Abfragen des Zustandes bei Auftreten eines Ereignisses, bestimmte Probleme erst entstehen.*

Das nächste Diagramm nimmt schon Teile der Implementation vorweg und bewegt sich in Richtung Integration, in der eine Dekomposition vermehrt möglich ist. Die Rede ist vom Sequenzdiagramm. Das Bild beschreibt das Zusammenwirken verschiedener Objekte für einen bestimmten Anwendungsfall. Es definiert somit die interne Sicht eines umgesetzten Use Cases, beschreibt also, wie sich der Use Case innerhalb der Anwendung realisieren lässt.

Der Nachrichtenaustausch zwischen den Instanzen ist hier in zeitlicher Reihenfolge ohne Warten[11] sichtbar, da die Pfeile horizontal sind. Im Diagramm wird der Use Case <manage Salary> als Szenario realisiert. Zum jetzigen Zeitpunkt entstehen die Parameter mit den zugehörigen Typen, d. h., ein Objekt muss jetzt mit anderen Objekten auf verträgli-

[11] Einen fatalistischen Stau von Momenten nennt man Warten.

che Art kommunizieren können. In diesem Stadium sind die Refactoring-Techniken am fruchtbarsten, da die Implementierung schon erste Funktions-Muster zeigt:

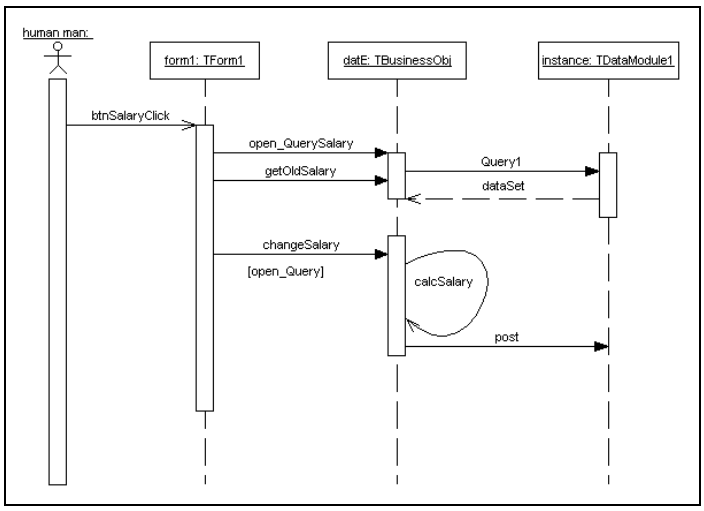

Abb. 1.49: Dynamischer Ablauf eines Szenarios

Umbau

Beim Package zeige ich nun eine Verschiebung als Refactoring einer Klasse zwischen den Paketen. Diese Technik kann die Abhängigkeiten zwischen den Paketen reduzieren, Änderungsraten einer Klasse in einem Paket A minimieren oder sind Teil zur Vorbereitung einer fast autonomen Komponente aus einem Paket.

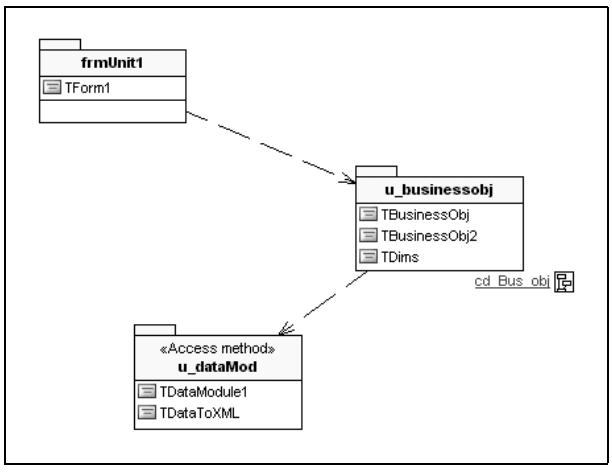

Abb. 1.50: Refactoring in Paketen

Die Einbindung der Units in Paket u_businessObj sieht vor und nach dem Verschieben der Klasse TDims gleich aus, hingegen entsteht eine neue Abhängigkeit von frmUnit1 zu Paket u_dataMod hin. Trotzdem ist dieses Paket nun für den Mehrfachnutzen optimaler vorbereitet.

Wiederverwendbarkeit kann in zwei Bereiche geteilt werden. Einerseits werden bei der Softwareentwicklung vorhandene Komponenten (VCL, CLX, EJB) wiederverwendet. Andererseits findet eine Entwicklung von Software (als Framework) statt, die auf Wiederverwendung in anderen Projekten ausgerichtet ist. Im letzteren Bereich ist die Wiederverwendbarkeit höher und der Aufwand für Modifikationen geringer!

```
Uses  //unit u_businessObj
  SysUtils, Windows, Classes, Controls,
              Dialogs, u_DataMod, DB;
```

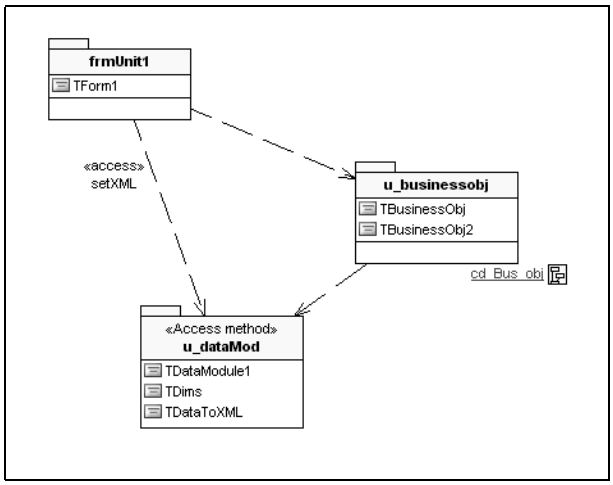

Abb. 1.51: Refactoring in Paketen, 2. Versuch

Durch das Refactoring hat auch eine Entkopplung von der GUI stattgefunden, da eine gefüllte Stringliste den TTreeview mit der Struktur zeichnet, d. h., jedes andere Rendering oder jedes andere Darstellen in einem Form ist nun möglich, sofern eine Routine den Inhalt der Stringliste parst und aufbereitet.

```
dimlist:= TStringlist.create;
datdepart:= TBusinessObj.Create(NIL);
try
  if datDepart.open_recursiveQuery then
  with TDims.create(NIL, DimList, datdepart) do begin
    FillTree(treeview1, NIL);
    Free;
  end;
```

```
  treeview1.FullExpand;
  btnLTree.visible:=true;
finally;
  dimlist.Free;
  datDepart.Free;
end;
```

Interessant ist wohl, sich am Schluss zu überlegen, wie denn die „richtigen“ Business-Objekte auf ihre Daten zugreifen. Richtige Business-Objekte haben oder können eine Persistenzfunktion gebrauchen, sodass auch der Zugriff und die Navigation auf eine relationale Datenbank objektorientiert, d. h. mit Methoden eines Objekts, erfolgen kann. Es ist mir also möglich, bspw. eine Personen-Tabelle wie ein Objekt zu behandeln und somit muss ich keine SQL-Statements absetzen:

```
 oPerson:=TPerson.Create;
try
   oPerson.Person_ID:=30;
   oPerson.DBRead(nil);
  if oPerson.Found then
    S_MBox(oPerson.LastName+' '+oPerson.FirstName);
finally
  oPerson.Free;
end;
```

Was aber, wenn wir es mit Relationen zwischen den Tabellen zu tun haben? Auch hier besticht ein objektorientierter Zugriff. Die Beispiele stammen übrigens aus MetaBase. Mit dem Framework MetaBase[xx] ist es möglich, ein Datenmodell komplett objektorientiert zugänglich zu machen. Der Meta-Layer erlaubt ein einfaches Portieren und MetaGen speichert die aus dem Datenmodell generierten Objekte in ein so genanntes Object Stream File. Später wird dieses File in der Entwicklungsschicht von Delphi oder von der Anwendung während der Laufzeit gelesen. Hier also ein relationaler Zugriff mit Methoden:

```
oPerson:=TPerson.Create;
oAddressList:=TAddressList.Create;
try
  oPerson.Person_ID:=30;
  oAddressList:=TAddressList.DBReadAllRelatedToObject(oPerson);
  for i:=0 to oAddressList.Count-1 do
    S_MBox(oPerson.LastName+' '+oAddressList.Adresses[i].Town);
finally
  oPerson.Free;
  oAddressList.Free;
end;
```

Die Klassenmethode `TAddressList.DBReadAllRelatedToObject(oPerson)` besitzt nun die Fähigkeit, eine Adressenliste anhand der übergebenen Person basierend auf dem

relationalen Datenmodell zu erstellen. Eine Klassenmethode kann über eine Klassenreferenz oder eine Objektreferenz aufgerufen werden. Hier wird sie über die Klassenreferenz aufgerufen. Bei einer Objektreferenz erhält `self` als Wert die Klasse des betreffenden Objekts. Somit lässt sich die Adressenliste an beliebige Instanzen von `TAddressList` zuweisen.

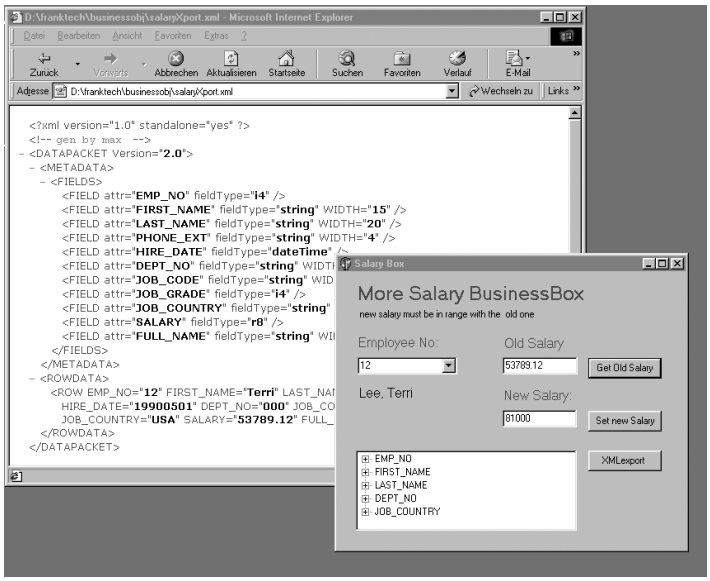

Abb. 1.52: Die Business-Box mit XML-Export

Nach diesem explorativen Businessbau als Refactoring-Geschichte wollen wir die erworbenen Kenntnisse innerhalb einer kurzen Schrittfolge zusammenfassen, sozusagen ein Strickmuster oder ein Vorgehen des Refactoring nach der Klassenfindung.

1.6.3 Schritte des Refactoring

Nachdem die Klassen durch die Techniken der Klassenfindung gefunden wurden (Activitydiagramme oder Design Patterns), bestimmen wir das nähere Verhalten und die Architektur mit den Paketen anhand folgender Schritte, die aber hochgradig iterativ ablaufen (eben auch Refactoring genannt):

Beim Design einer Paketstruktur kann folgendes Problem auftreten: Ein Paket A hängt von Paket B ab, wenn A das Interface von B benutzt. Wenn noch ein C ins Spiel kommt, gilt es zyklische Abhängigkeiten zu vermeiden.

> *Da eine Unit A in ihrer uses-Klausel die Unit B aufführen kann und Unit B wiederum eine weitere Unit C in ihrer uses-Klausel aufnehmen kann, ist A indirekt auch von C abhängig.*

Klassen in Module einteilen

Zirkuläre Referenzen treten auf, wenn Sie wechselseitig abhängige Units haben, d. h., wenn eine Unit A in ihrer uses-Klausel Unit B aufführt und wenn die uses-Klausel von Unit B ihrerseits Unit A referenziert. Solche zirkulären Referenzen sind dann zugelassen, wenn **höchstens eine** der zwei Referenzen im Interface-Teil erscheint.

Eine weitere Möglichkeit zum Aufbrechen eines Zyklus oder einer zu direkten Abhängigkeit ist die so genannte Umkehrung der Abhängigkeit:

Vorher: `A-> B-> C, C-> A`

Nachher: `A-> B-> C, A-> C`

> *Hier extrahiert oder verschiebt man ein Interface von A nach C.*

Logisch erscheint auch die Refactoring-Tatsache, dass abhängige Pakete nicht von solchen abhängen sollen, die noch änderungsanfälliger sind als das Paket selbst.

> *Es ist das Maß der Änderungsstabilität, welches versucht, Änderungsraten anhand dem Verhältnis abstrakter zu konkreter Klassen oder der Anpassungsmöglichkeiten zu bestimmen.*

Die Regel besagt nun, dass Abhängigkeiten in Richtung stabiler Pakete gehen sollten, was z. B. ein Kernel als stabiles Paket darstellt. Nun, wie erreichen wir eine geschickte Einteilung von Paketen, die auf Zuwachs oder die auf Änderung hin geplant werden? Hier hilft das Architektur Pattern MVC weiter, das ich in Teil 3 unter den Architekturen näher vorstellen will.

> *Document (Model), View, Controller (DVC) ist auch ein Prinzip zur Verteilung der Funktionalität auf Klassen, das als Vorbereitung zu mehrschichtigen (multitier) Klassen/Komponenten dient.*

Verarbeitungsdominante Klassen realisieren Kontrolle und Steuerung von 1 oder n-Use Case und werden auf <entity> oder <control> -Klassen verteilt.

Dialogdominante Klassen sollten wenig «Wissen» enthalten und sind in <boundary>-Klassen zu finden. Diese Notation findet man unter den Robustnessdiagrammen:

```
classes: boundary -> control -> entity
```

Es geht also darum, die Klassen in entsprechende Units zu verteilen. In unserem Business-Beispiel wurde folgende Aufteilung in vier Units (Module oder Packages) vorgenommen:

- `TDataModule1` und `TDataToXML` in der Unit `u_dataMod` (enthält den möglichen Zugriff auf InterBase)
- `TBusinessObj` und `TBusinessObj2` in der mittleren Unit `u_businessObj` (Verarbeitung und Business-Rules)
- `TForm1` in der Unit `frmUnit1` (die Darstellung der Dialoge und Formulare)
- `controllAction` in der Unit `busControll` (vermittelt zwischen dem View, den Ereignissen und der Logik)

Sichtbarkeiten und Zugriff definieren

Alle Teile von Klassen, einschließlich der Felder, Methoden und Eigenschaften befinden sich auf einer bestimmten Schutz- oder Sichtbarkeitsebene. Inwieweit der Zugriff auf die Attribute und Methoden erlaubt ist, können Sie allgemein an der Kennzeichnung (ab UML 1.1) erkennen:

- public: ein « + » vorangestellt
- protected: ein « # » vorangestellt
- private: ein « - » vorangestellt

Meist werden Sie Methoden in Klassen oder Komponenten als `public` oder `protected` deklarieren. Nur selten ist das Attribut private nötig, das bewirkt, dass nicht einmal abgeleitete Klassen Zugriff erhalten. Deklarieren Sie Elemente als `private`, wenn sie nur in der Klasse verfügbar sein sollen, in der sie auch definiert werden. Deklarieren Sie Elemente als protected, wenn sie nur in dieser Klasse und in ihren Nachkommen verfügbar sein sollen.

Bedenken Sie aber, dass ein Element, das an irgendeiner Position innerhalb einer Unit verfügbar ist, in der gesamten Datei zur Verfügung steht. Wenn Sie also zwei Klassen in derselben Unit definieren, können diese auf die als private deklarierten Methoden der jeweils anderen Klasse zugreifen (friend-Beziehung).

> *Zusätzlich lassen sich Properties-Methoden in der Klasse definieren. Somit lassen sich in den read- und write-Abschnitten einer Eigenschaftsdeklaration anstelle eines Feldes die Zugriffsmethoden festlegen.*

Angenommen Sie haben soeben eine Eigenschaft `KreditLimit` eingebaut. Die Anforderung wird nun dahingehend erhöht, dass jede Änderung der Kreditlimite eine Autorisierung mit einem Passwort erfordert. Mit einem normalen Feld oder einer mehrfach vorhanden Funktion wären Sie gezwungen, an allen Stellen im System eine zusätzliche Validierung einzufügen. Nicht so mit einer Kapselung, die einfach ist:

```
  protected
    function getKreditLimit: double;
    procedure setKreditLimit(newLimit: double);
  public
    property KreditLimit: double
      read getKreditLimit write setKreditLimit;
```

Hier ist wieder das eigentlich Feld `FKreditLimit` geschützt im privaten Bereich, jeder Zugriff muss also zwangsläufig über die Eigenschaft `KreditLimit` erfolgen, die bei Lesen oder Schreiben die entsprechende Methode, z. B. `setKreditLimit`, feuert:

```
procedure TKontoClass.setKreditLimit(newLimit: double);
begin
  if FKreditLimit <> newLimit then begin
    if getPassword(frmTrans.password) then begin
      FKreditLimit:= newLimit
```

```
    end else MessageDlg(frmTrans.Label2.
                    Caption,mtWarning,[mbok],0);
  end;
end;
```

Zunächst wird beim versuchten Schreiben eines neuen Kreditlimits ein Passwortdialog aufgerufen (der übrigens von einer DLL stammt), der bei erfolgreicher Autorisierung das private Feld `FKreditLimit` ändert. Ein direkter Zugriff auf das Feld ist somit nicht mehr möglich und die Methode ist künftig erweiterbar, z. B. mit einer Datumskontrolle oder einer Berechnung der Limite.

Beziehungen festlegen

Weiter lässt sich die Kommunikation zwischen den Klassen wie Vererbung (Polymorphie), Aggregation und Assoziation erstellen und <relationships> zu Klassen einbinden. Gedanklich sind diese Beziehungen meistens schon bei der Klassenfindung erarbeitet worden, werden aber in dieser Phase stets neu hinterfragt und schrittweise implementiert.

Denn während das statische Modell die Beziehungen abbildet, in denen Klassen zueinander stehen, beschreibt das spätere dynamische Modell, wie diese Beziehungen benutzt werden, d. h., welche Botschaften entlang diesen Beziehungen fließen.

Eine Methode umfasst nur in den seltensten Fällen die Ausführung einer einzigen Funktion. Wesentlich häufiger sind verkettete Aufrufe mehrerer Methoden und Kaskadierungen möglich, die aber nicht in derselben Klasse angesiedelt zu sein brauchen. Somit wird klar, dass diese Phase eben stark iterativ geprägt ist und einer wiederholten Selbstprüfung des Refactoring (zu zweit) nichts im Wege stehen sollte.

1.7 Unit Testing

Testen ist eine anspruchsvolle Aufgabe: Man kann auf verschiedenen Ebenen ansetzen und innerhalb dieser Ebenen diverse Vorgehensweisen wählen. Bei den Ebenen spricht man von Testlevels und bei den Vorgehensweisen von Teststrategien. Testlevels gehen vom Kleinen zum Großen. Gemeinhin unterscheidet man folgende Testlevels:

- **Einheitentests** (unit tests): Jede einzelne Komponente wird daraufhin getestet, ob sie die Detailspezifikationen korrekt implementiert. Mit regression testing wird dabei das Wiederholen von bereits bestandenen Tests nach Codeänderungen bezeichnet.
- **Integrationstests**: Zunehmend größere Gruppen von Softwarekomponenten werden zusammengefügt und es wird ihr Zusammenspiel getestet.
- **Systemtests**: Die gesamte Software wird zu einem Produkt integriert und es wird getestet, ob alle Spezifikationen erfüllt sind.
- **Akzeptanztests**: Es wird getestet, ob Kunden oder Benutzer die Software akzeptieren.

Mit dem Aufkommen von Extreme Programming (XP) hat sich der Fokus stark auf die Einheitentests verlagert. Einheitentests lassen sich heute mit den geeigneten Tools automatisieren, die ich mit DUnit und NUnit vertiefen möchte.

1.7.1 DUnit

Als letztes Kapitel des ersten Teils stelle ich das musterhafte Testen vor, dass im Zusammenhang mit dem Vorgehensmuster des Extreme Programing (XP) besser bekannt wurde und nun auch für Delphi, Kylix (DUnit) oder innerhalb .NET als NUnit einsatzbereit ist. Beide Units stelle ich vor.

In Teil 2 über Design Patterns wird auch immer die Rede von einem TRadeRoboter sein, der als fast durchgängige Anwendung für viele Patterns herhalten muss. Im Kapitel „UML 2.0 Projekt“ ist der TRadeRoboter als durchgängiges Modell schon zu erkennen. Im TRadeRoboter sind deshalb die modellierten Patterns als Beispielcode integriert, und die Anwendung selbst habe ich zusätzlich mit ModelMaker und DUnit teilweise getestet. So schließt sich der Kreis von UML, den Patterns und zugehörigen Techniken wieder. DUnit hat ja als Symbol Yin und Yang in harmonischer Ruhe verewigt.

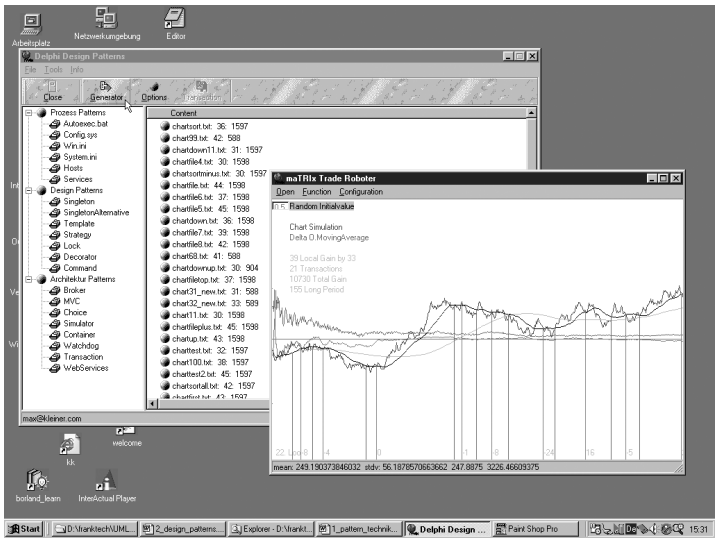

Abb. 1.53: TRadeRoboter als Begleiter durchs Buch

DUnit ist ein Framework, das ursprünglich aus der Java-Welt stammt und zugleich eine Technik und ein konkretes Werkzeug ist. Das faszinierende an DUnit ist die Tatsache, dass das Framework intensiv von Design Patterns Gebrauch macht, vor allem Decorator, Command und Composite, und gleichzeitig einen Testrahmen fürs Refactoring vorgibt. Denn die Qualitätssicherung von Code bei Änderungen ist sehr mühselig (Regressionstests sind dann nötig), sodass sich immer häufiger automatische Testmethoden in Form von Tools anbieten.

Die folgende Tabelle soll aufzeigen, wo sich innerhalb der Refactoring-Funktionen Fehler einschleichen können, die bei einer passenden, eingerichteten Testklasse ans Tageslicht kommen könnten.

> *Generell ist DUnit aber ein Einheitentest (Unit oder Patterns sind Einheiten), demzufolge sind Tests zwischen den Packages (Integration) nicht vorgesehen.*

Einheit	Refactoring-Funktion	Beschreibung
Package	Rename Package	Umbenennen eines Packages
Package	Move Package	Verschieben eines Packages
Class	Extract Superclass	Aus Methoden, Eigenschaften eine Oberklasse erzeugen und verwenden
Class	Introduce Parameter	Ersetzen eines Ausdrucks durch einen Methodenparameter
Class	Extract Method	Heraustrennen einer Codepassage
Interface	Extract Interface	Aus Methoden ein Interface erzeugen
Interface	Use Interface	Erzeuge Referenzen auf Klasse mit Referenz auf implementierte Schnittstelle
Component	Replace Inheritance with Delegation	Ersetze vererbte Methoden durch Delegation in innere Klasse
Class	Encapsulate Fields	Getter und Setter einbauen
Modell	Safe Delete	Löschen einer Klasse mit Referenzen

Tab. 1.8: Vom Refactoring zum Testen

Das Testen mit DUnit oder einem anderen Framework setzt auf drei Elementen auf:

- Das eigentliche Testobjekt, fixture genannt
- Der Testfall zum Objekt, action genannt
- Das Resultat nach dem Testfall, check genannt

Als einfaches Beispiel sei hier das Addieren von zwei Zahlen erwähnt. Das Resultat kann positiv, negativ oder ein unerwarteter Fehler sein. Der Check testet, ob das Resultat dem Originalwert plus 5 entspricht:

```
function TestAdd5ToNumber(n: integer):boolean;
var
  OrigVal: integer;
begin
  OrigVal := n;
  n := n + 5;
  //Check result of action
```

```
  Result := n = OrigVal + 5;
end;
```

Es stellt sich grundsätzlich die Frage, wieweit man beim Testen auch den bestehenden Code, den andere gebrauchen, infrage stellen muss. Man kann die bequeme Haltung einnehmen und immer den Aufrufer verantwortlich machen, wenn er ungültige Daten in einer Methode übergibt.

„Please call functions with valid data, don't bother me with data in bad shape", hörte ich einmal einen Entwickler aus Brighton, England rufen.

Oder man handelt nach dem Grundsatz des „Design by Contract", indem man den Verantwortlichen festlegt. Mit Nachbedingungen entlaste ich den Aufrufer und bin selbst verantwortlich, dass ich meine Methoden mit Assert hieb- und stichfest entwickle.

Mit einer Vorbedingung lässt sich zwar Verantwortung an den Aufrufer abschieben, eine Garantie hat man aber nicht. Angenommen Sie haben die Funktion einer Standardabweichung mit Durchschnitt entwickelt:

```
procedure MeanAndStdDev(const Data: array of Double;
                            var Mean, StdDev: extended);
var S: extended;
  n,i: Integer;
begin
  n:= High(Data)- Low(Data) + 1;
  Mean:= Sum(Data)/n;
  S:= 0;
  for i:= Low(Data) to High(Data) do
    S:= S + Sqr(Mean - Data[i]);
  StdDev:= Sqrt(S / (n - 1));
end;
```

Wer garantiert nun, dass die Anzahl der Elemente in `Data` nicht weniger als 2 sein darf, da sonst in der letzten Zeile eine Division durch null erfolgt?

Mit einem Assert würden Sie explizit die Verantwortung übernehmen, und bei ungültigen Daten käme eine `EAssertionFailed` geflogen:

```
Assert( length(values) >= 2);
```

Oder wie es im Original der Laufzeitbibliothek in der *math.pas* sauber nach „Design by Contract" vorzufinden ist, d. h., wenn der Vertrag gebrochen wird, verlassen wir mit `Exit` schleunigst die Prozedur:

```
 if N = 1 then begin
   Mean:= Data[0];
   StdDev:= Data[0];
   Exit;
 end;
```

Einrichten des Frameworks

Als Erstes gelangt man zum kostenlosen Framework mit allen Sourcen und exzellenter Dokumentation mit folgendem Link: *http://dunit.sourceforge.net*

Es wird mindestens Delphi 4 benötigt. Achten Sie darauf, beim Auspacken der Dateien die Verzeichnisnamen beizubehalten. Als Nächstes fügen Sie den Pfad der Sourcen (vor allem *TestFramework.pas* und *GUITestRunner.pas*) in den Delphi-Optionen hinzu. Einfach den kompletten Pfad \dunit\src hinzufügen. Einen ersten Eindruck gewinnt man, indem im erstellten Verzeichnis \dunit\tests zuerst die Library *dunittestlib.dpr* und dann *unittests.dpr* compiliert wird. Beim Starten der EXE wird das ganze Spektrum der eingebauten Testmöglichkeiten in Abb. 1.54 ersichtlich:

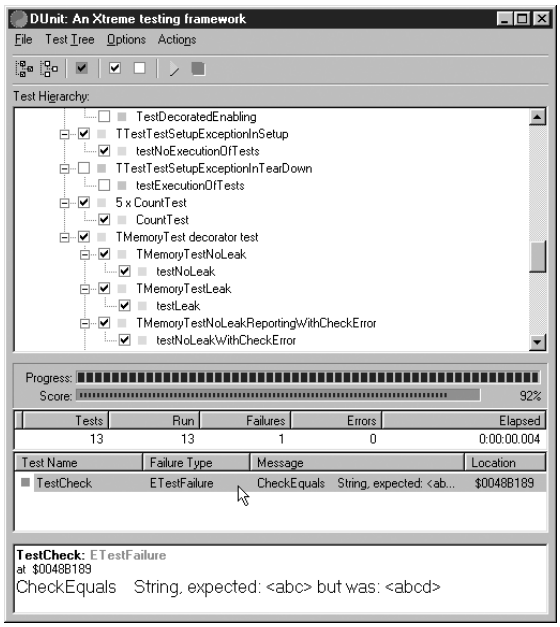

Abb. 1.54: Das GUI innerhalb von DUnit

Ich suche mir im mitgelieferten Code eine Klasse aus einer Unit, die ins Testframework eingebunden ist, die ich durch eine Suche nach `Check` gefunden habe.

Wenn ich nun in der Datei *UnitTestFramework* die Zeile eines Vergleichs derart ändere:

```
CheckEquals('abc', 'abcd', 'CheckEquals  String');
```

so provoziere ich die logische Fehlermeldung in Abb. 1.53 im Framework, das nebst der Adresse auch die vergangene Zeit und die Anzahl der Testläufe protokolliert. Wenn man einen eigenen Testfall einrichtet, erbt man die Klasse `TTestCase` als Subclassing, die wiederum zur Laufzeit unseren registrierten Test mit RTTI (RunTimeTypeInformation) auswertet. Demzufolge sind unsere Testmethoden als published zu deklarieren:

```
TFailuresTestCase = class(TTestCase)
  private
    procedure DoNothing;
  published
    procedure OneSuccess;
    procedure OneFailure;
    procedure SecondFailure;
    procedure OneError;
end;
```

Im initialisierten Teil der Unit lässt sich der Test beim Framework registrieren. Der initialization-Abschnitt enthält Anweisungen, die beim Programmstart in der angegebenen Reihenfolge ausgeführt werden. Arbeiten Sie beispielsweise mit definierten Datenstrukturen, können Sie diese im initialization-Abschnitt initialisieren.

```
Initialization
  RegisterTests('GUI Tests', [TGUITestRunnerTests.Suite]);
```

Damit das Framework weiß, welche registrierten Tests von Anfang an in der Startroutine geprüft werden, ist es zentral, die Projektdatei mit den Testklassen auf den Start des Frameworks umzulenken. Anstelle der eigentlichen `Applicaiton.Run` kommt die des Frameworks zum Flug. *TestFramework.pas* und *GUITestRunner.pas* sind in der uses-Anweisung der Projektdatei anzufügen. Mit den entsprechenden Compilerdirektiven lassen sich diese zusätzlichen Testcodes bei der Auslieferung ja wieder bereinigen:

```
begin
{$IFDEF DUNIT_CLX}
    TGUITestRunner.RunRegisteredTests;
{$ELSE}
    GUITestRunner.RunRegisteredTests;
{$ENDIF}
end.
```

Weil bedingte Direktiven jedoch den Quelltext schwer verständlich und komplizierter in der Wartung machen, sollten Sie wissen, wann es sinnvoll ist, diese zu verwenden. DUnit arbeitet intern auch intensiv mit den Methoden `assert` oder `assigned`. In Delphi und anderen Sprachen können Sie mit Assert testen, ob Bedingungen verletzt werden, die als zutreffend angenommen werden (siehe OCL).

Assert bietet eine Möglichkeit, eine unerwartete Bedingung zu simulieren und ein Programm anzuhalten, anstatt die Ausführung in unbekannter Konstellation fortzusetzen. Assert übernimmt als Parameter einen booleschen Ausdruck und einen optionalen Meldungstext. Schlägt der boolesche Test fehl, löst Assert eine `EAssertionFailed`-Exception aus. Auch die Kombination von beidem (Assert und Assigned) ist in DUnit anzutreffen:

```
procedure TTestResult.Run(test: ITest);
begin
```

```
  assert(assigned(test));
  if not ShouldStop then begin
    StartTest(test);
    try
      if RunTestSetUp(test) then
        RunTestRun(test);
      RunTestTearDown(test);
    finally
      EndTest(test);
    end;
  end;
end;
```

In der Regel werden Assertions nicht in Programmversionen verwendet, die zur Auslieferung vorgesehen sind. Hierzu dienen Exceptions, die während der Laufzeit den Code robuster machen sollen, indem sie auf Fehler reagieren und evtl. sogar korrigieren.

> *Assertions testen auf die Korrektheit des Codes, und Sie als Entwickler agieren beim Entdecken eines Bug eher, als die Exceptions reagieren!*

Deshalb existieren Compiler-Direktiven, mit denen die Generierung des zugehörigen Codes vor der Auslieferung deaktiviert werden kann:

```
$ASSERTIONS ON/OFF  (Lange Form)
```

Workshop in DUnit

Die Einarbeitungszeit eines solchen Frameworks beträgt einige Tage, bei wirklich zwingendem Gebrauch, sei es von der QS gefordert oder vom Kunden erwünscht, ist eine gute Woche für die Schulung zu planen, da vor allem das Wissen, wie man die Testfälle erstellt, zu Buche schlägt. Neben dem eigentlichen Programmentwurf besteht die Entwicklung aus einem sich ständig wiederholenden Zyklus von Quelltexteingabe und anschließender Fehlerprüfung bei DUnit.

Damit auch wirklich alle Aspekte des Programmverhaltens von diesem Test erfasst werden, müssen Sie planvoll vorgehen. Sobald Sie einen Programmierfehler gefunden haben, korrigieren Sie das Problem im Code, compilieren erneut und fahren mit dem Test und den Checks fort.

Eine besondere Situation ist auch das Testen bei Vererbung. Eigentlich sollte man die Methoden der Oberklasse nicht nochmals in den Unterklassen testen. Zu testen sind jedoch die überschriebenen Methoden!

Auch bei Eigenschaften, die in der Unterklasse mit neuen Werten gefüttert und dann von der Oberklasse gebraucht werden, ist Vorsicht geboten. Ich beginne nun Schritt für Schritt mit einem Testfall einer `StringList` mit DUnit. Listen sind häufige Kandidaten, da ein verändertes Datenformat, die Kapazität oder eine Typenkonversierung zu Seiteneffekten in den Listen führen kann.

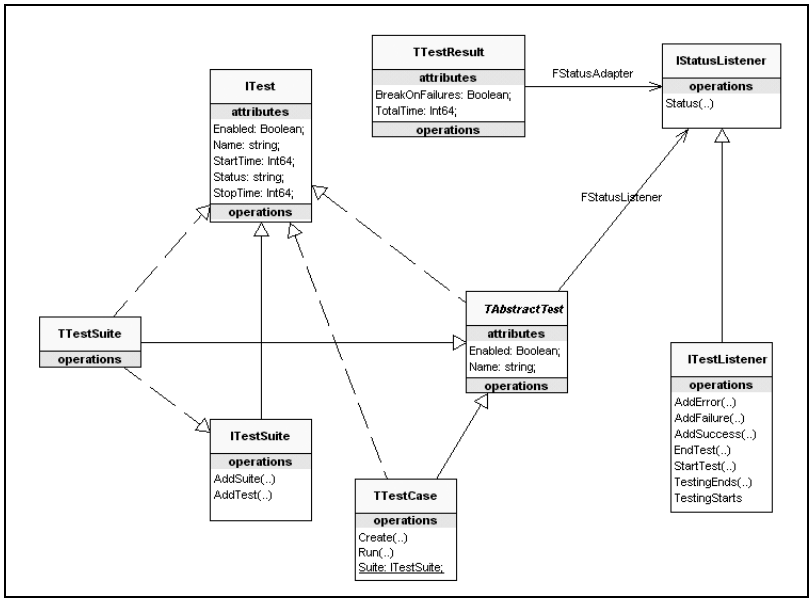

Abb. 1.55: Das Framework von DUnit

Testbau

Ich starte ein neues Projekt und füge in der zu testenden Unit `TestFramework` in der uses-Klausel hinzu. Ich deklariere eine neue Klasse, die von `TTestCase` abstammt.

```
Interface
uses
 TestFrameWork, Classes;
type
 TTestStringList = class(TTestCase)
 private
   Fsl: TStringList;
 protected
   procedure SetUp; override;
   procedure TearDown; override;
 published
   procedure TestPopulateStringList;
   procedure TestSortStringList;
 end;
```

Diese Testklasse hat sich nun höflichst im Initialisierungsteil der Unit zu registrieren:

```
RegisterTest('CODESIGN suite', TTestStringList.Suite);
```

Als letzter Schritt der Vorbereitung des „Frameworkshops“ ist das Einklinken in die Projektdatei zu bewerkstelligen, sodass beim Start der Anwendung getestet werden kann:

```
Application.Run; //normal
new
GUITestRunner.RunRegisteredTests;
```

Beim Starten des Programms sind die Testfälle im Baum des Frameworks ersichtlich, das die Aufgabe eines Regiezentrums übernimmt und bei uns auch mal geringschätzig mit TestDirector tituliert wurde. Ich komme zur Implementierung der beiden Testmethoden:

```
procedure TTestStringList.SetUp;
begin
  Fsl:= TStringList.Create;
end;

procedure TTestStringList.TearDown;
begin
  Fsl.Free;
end;
```

> *Die Bezeichnung `SetUp` und `TearDown` ist eine Konvention in der Testwelt, die besagt, in diesen Methoden die Vor- und Nachbedingungen einer Testklasse zu implementieren, d. h. Initialisierungen, Bedingungen bis hin zu abschließenden Aufräumarbeiten.*

Bei rein arithmetischen Tests fehlen in der Regel diese beiden Methoden.

Die eigentlichen Testfälle Population und Sortierung sollen aufzeigen, wie repetitive Fälle in das Framework eingebunden werden, und einen Eindruck vermitteln, dass sich das Testen von Mengen auch automatisieren lässt.

Klar, diese Fälle sind noch trivial und stellen in keiner Art und Weise die Funktionalität einer `TStringList` infrage. Wie häufig scheitert man jedoch am Trivialen, wenn es sich häuft. Check & Test soll hier die Devise sein, zumal sich die Technik auch hervorragend für einen Stress-Test eignet.

```
procedure TTestStringList.TestPopulateStringList;
var
 i: Integer;
begin
 Check(Fsl.Count = 0);
 for i:= 1 to 50 do // Iterate
   Fsl.Add('i');
 Check(Fsl.Count = 50);
end;

procedure TTestStringList.TestSortStringList;
```

```
begin
  Check(Fsl.Sorted = False); //Assertion
  Check(Fsl.Count = 0);
  Fsl.Add('Trade');
  Fsl.Add('Roboter');
  Fsl.Add('Love');
  Fsl.Sorted:= True;
  Check(Fsl[2] = 'Trade');
  Check(Fsl[1] = 'Roboter');
  Check(Fsl[0] = 'Love');
end;
```

Weitere Techniken wie das Verwalten von n-Testklassen in einer Testsuite, Starten aus der Konsole oder repetitive Tests werden in den Code-Beispielen von DUnit ausreichend beschrieben. Schließlich stellt sich die Frage, ob das Verwalten der Testklassen als eigene Unit oder direkt im produktiven Code anzustreben sei. Dazu sind in den Sourcen wiederum zwei Projekte, die den Unterschied aufzeigen, vorhanden.

Die Trennung in verschiedene Units hat zwei klare Vorteile: Das Release Management wie das Arbeiten in Teams lässt sich vereinfachen, bedingt aber durch Delegation einen höheren Anfangsaufwand, z. B. benötigt man auch zwei Projektdateien, die sich natürlich in einer Projektgruppe definieren lassen.

> *Sie können zusammengehörige Projekte zu Projektgruppen zusammenfassen. Projektgruppen ermöglichen es Ihnen, an miteinander verbundenen Projekten zu arbeiten und diese zu organisieren (z. B. Anwendungen und DLLs, die zusammen eine mehrschichtige Anwendung bilden).*

Die Projektgruppendatei (Dateinamenserweiterung BPG) enthält Make-Befehle, um die Projekte der Gruppe zu erzeugen. Immer wenn Sie einer Projektgruppe ein Projekt hinzufügen, wird ein Verweis auf dieses Projekt in die BPG-Datei eingefügt.

Das ganze Rahmenwerk hat sich über zwei Jahre entwickelt, die vorliegende Version ist recht ausgereift. Als Tribut der Arbeit und Wertschätzung an die Autoren von DUnit möchte ich es nicht missen, den Originaltext vorzustellen:

DUnit is an Xtreme testing framework for Borland Delphi programs. It was originally inspired on the JUnit framework written in Java by Kent Beck and Erich Gamma, but has evolved into a tool that uses much more of the Potenzial of Delphi to be much more useful to Delphi developers.

DUnit Real Test

Wozu dient das Testen, wenn man zwischen Klassen eine Beziehung hat? Dieser interessanten Fragestellung soll ein kurzes Beispiel folgen. Es handelt sich um zwei Klassen, `TCustomer` und `TOrder` genannt.

Es besteht eine doppelte Relation, da man in `TOrder` mit der Methode `getCustomer` auch wissen will, wer der Kunde des Auftrags ist.

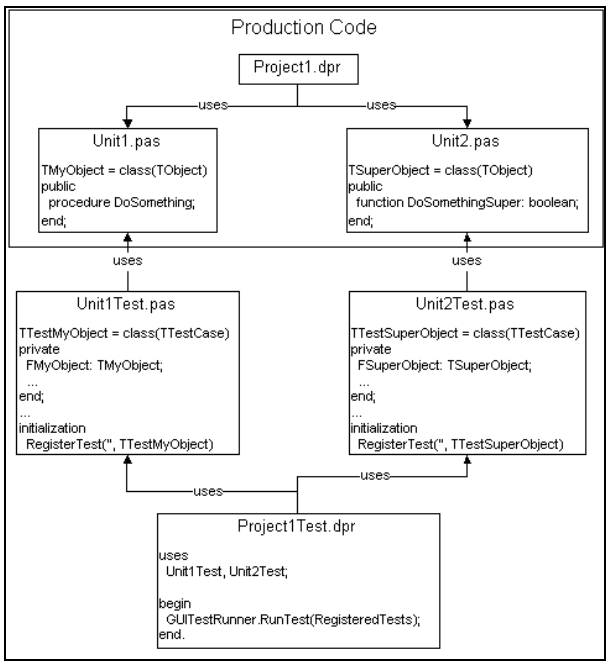

Abb. 1.56: Produktiver- und Testcode in DUnit

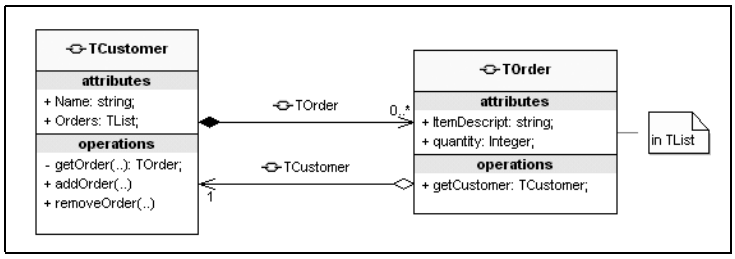

Abb. 1.57: Eine Beziehung zum Testen

In der Liste lassen sich nun die einzelnen Aufträge mit `addOrder` hinzufügen. Auf der anderen Seite will ich mit assert testen, ob die Funktion `getCustomer` den wirklichen Kunden kennt bzw. die Instanzierung und Navigation der Liste richtig funktioniert. Der Testcode erzeugt einen Kunden und einen Auftrag, der in die Liste aufgenommen wird. Somit ist eine Beziehung zwischen den beiden Klassen entstanden. Der triviale, aber stichhaltige Testcode sieht so aus: [xxi]

```
cust:= TCustomer.create();
ord:= TOrder.create();
cust.addOrder(ord);
```

```
assert(ord.getCustomer = cust)
assert(ord in cust.Orders())
```

In manchen Fällen muss eine Prozedur oder Funktion beim Auftreten eines solchen Tests beweisen, dass die Liste die Beziehung wirklich herstellt. Wenn die Initialisierung von `ord` aus irgendeinem Grund fehlschlägt, etwa weil der angegebene Suchpfad ungültig ist oder weil zu wenig Speicherplatz verfügbar ist, wird auch die Beziehung nicht klappen. Mit obigem Testcode sollten künftig auch im Klassenverbund Testfälle möglich sein. Testen ist nicht etwa erfolgreich, wenn keine Fehler entdeckt wurden, sondern dann, wenn so viele Fehler wie möglich gefunden werden!

1.7.2 NUnit

Das folgende und letzte Kapitel gibt einige Anhaltspunkte zum Testen von .NET-Projekten generell und zum Testen von Einheiten mit NUnit. Testen besteht im Allgemeinen aus den folgenden zwei Teilaufgaben:

1. **Verifikation** bzw. die Frage: „Machen wir den richtigen Job?“, d. h., entspricht die Applikation der Spezifikation?
2. **Validierung** bzw. die Frage: „Machen wir den Job richtig?“, d. h., läuft der Code einigermaßen fehlerfrei?

Testen unterscheidet sich vom Debugging darin, dass es beim Testen darum geht, Fehler zu finden. Beim Debugging dagegen geht es darum, für bekannte Fehler die Ursachen zu analysieren und diese möglichst zu beheben. DUnit und NUnit vertreten die gleiche Philosophie.

Teststrategien

Mit dem Aufkommen von .NET hat sich auch der Fokus stark auf die Einheitentests verlagert. Einheitentests lassen sich heute mit den geeigneten Tools ziemlich automatisieren und sie sind die Grundlage für alles andere, wie die folgenden zwei Zitate aus der MS-Online-Hilfe belegen: *„Das Testen von Einheiten hat sich bewährt, weil dadurch ein großer Prozentsatz von Fehlern ermittelt werden kann. ... Das Auffinden des bzw. der Fehler im integrierten Modul ist viel komplizierter, als zuerst die Einheiten zu isolieren, diese einzeln zu testen, sie anschließend zu integrieren und dann das integrierte Modul zu testen.“* [12]

Bei den Teststrategien unterscheidet man folgendeVorgehensweisen:

- **Blackbox-Tests** testen, ob eine bestimmte Komponente die Eingangswerte korrekt in das gewünschte Resultat umwandelt, wobei die konkrete Implementierung vernachlässigt wird.
- **Whitebox-Tests** testen die konkrete Implementierung Schritt für Schritt.

[12] Visual Studio .NET Online-Hilfe, Kapitel „Testen von Einheiten“.

- **Performance-Tests** testen, ob die Ausführungsgeschwindigkeit und der Ressourcenverbrauch akzeptabel sind. Bei Software für den Mehrbenutzerbetrieb wird mit Load-Tests oder Stress-Tests geprüft, wie sich das System unter großer Last (viele Benutzer/innen gleichzeitig) verhält.
- **Konfigurationstests** testen, wie sich eine Software unter verschiedenen Konfigurationen verhält (läuft eine Java-Applikation z. B. tatsächlich unter PC, Macintosh, Linux).

Einheitentest mit NUnit

Im Zusammenhang mit Patterns ist vor allem das Testen von Objekten oder Objektmethoden von Interesse. Wir bewegen uns also auf dem Level Einheitentests. Meistens werden diese Tests als automatisierte Blackbox-Tests durchgeführt.

Obwohl .NET noch nicht allzu lange existiert, gibt es bereits eine Unzahl von meist kostenlosen Tools. Eine kleine Webrecherche förderte unter anderem NUnit[xxii], NUnitASP (Ergänzung zu NUnit)[xxiii], csUnit[xxiv] oder #unit[xxv] (Teil der freien Entwicklungsumgebung #develop) zu Tage. Im Folgenden stütze ich mich auf NUnit, weil dieses Tool in einem MSDN-Artikel explizit behandelt wird.[13] Für den Download finden Sie NUnit unter *http://www.nunit.org/*.

Zentral für erfolgreiches Testen ist, dass die Tests und Testfälle bereits geplant werden, nachdem die Spezifikation vorliegt, und dass die Einheitentests parallel zur Implementierung erstellt und durchgeführt werden. Die Automatisierung ermöglicht, einen einmal definierten Testfall nach jeder Codeänderung ohne großen Aufwand wieder zu überprüfen, sodass sich Quelltexterstellung und anschließende Einheitentests ständig abwechseln.

Einrichten von NUnit

Nach dem Download muss NUnit zuerst mit einem Doppelklick installiert werden. Standardmäßig installiert es sich in das Verzeichnis *C:\Programme\Nunit V2.0*. Aufrufen lässt es sich in zwei Versionen, nämlich in einer Konsolenversion und in einer Version mit GUI.

Eine Hilfe enthält NUnit nicht (zumindest in der Version 2.0.6). Für den Einstieg empfiehlt es sich deshalb, die zwei im Verzeichnis \doc\ mitinstallierten Dokumente durchzugehen. *ReadMe.pdf* gibt Hinweise zur Installation, zu den Versionen und zum Aufsetzen einer Testklasse. *QuickStart.doc* ist ein Mini-Tutorial, das an einem konkreten Beispiel zeigt, wie man mit NUnit arbeitet. Wenn Sie ein wenig mehr als nur den nackten Testcode möchten, dann kann ich den bereits erwähnten MSDN-Artikel wärmstens empfehlen.

Um NUnit in einem VS-Projekt verwenden zu können, muss man die NUnit-DLL mit einem Verweis in das Projekt einbinden.

[13] Eric Gunnerson, März 2003: Unit Testing and Test-First Development, MSDN Online.

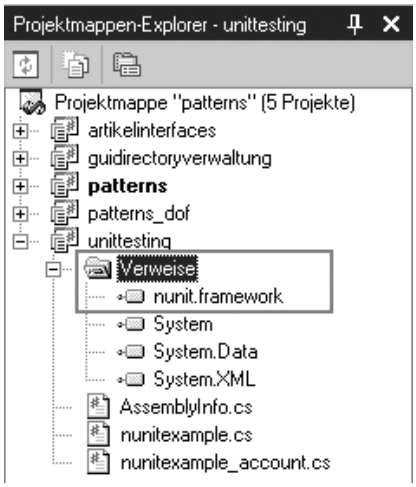

Abb. 1.58: Verweis auf NUnit in einem VS-Projekt

Workshop

Sofern die Installation erfolgreich verlaufen ist, soll als Nächstes das gleiche Codebeispiel wie für die vorgängige DUnit in Delphi durchgespielt werden, d. h., mit der in `System.Collections` vorkommenden Klasse `ArrayList` sollen ein paar Tests gemacht werden. Am besten erstellen Sie ein neues Projekt für eine Klassenbibliothek und fügen einen neuen Verweis ein auf die nunit.framework-DLL, die sich bei einer Standardinstallation im Verzeichnis *C:\Programme\Nunit V2.0\bin* befindet.

Als Nächstes erstellen wir den Rumpf einer Testklasse. In der Vorversion erstellte man Testklassen durch Vererbung, doch ab Version 2 von NUnit wird mit Attributen gearbeitet. Eine Testklasse weist mindestens die folgenden Attribute auf:

- [TestFixture] wird der Klasse vorangestellt, um sie als Testklasse zu kennzeichnen.
- [SetUp] steht vor der Methode, die den Test initialisiert, z. B. indem sie eine Instanz der zu testenden Klasse erzeugt. Der Methodenname ist übrigens beliebig.
- [Test] steht vor jeder Testmethode.
- [TearDown] kommt eventuell als weiterer Teil dazu, wenn nach dem Test gewisse Aufräumarbeiten notwendig sind.

Zusammengefasst ergibt dies folgenden Code:

```
using System;
using System.Collections;
using NUnit.Framework;
namespace unittesting {

  [TestFixture]
  public class TTestStringList {
    private ArrayList Fsl;
```

```
    [SetUp]
    public void SetUp() {
      Fsl = new ArrayList();
    }
    [TearDown]
    public void TearDown() {
      Fsl = null;
    }
    [Test]
    public void TestPopulateStringList() {
    }
  }
}
```

Damit haben wir zwar bereits ein funktionsfähiges Testprojekt, das sich nach der Erstellung in NUnit tatsächlich aufrufen lässt, aber getestet haben wir noch nichts. Das ändern wir, indem wir in der Methode `TestPopulateStringList` die ArrayList `Fsl` in einer Schleife mit 50 Strings abfüllen. Um zu testen, ob das klappt, stellt uns NUnit nun die Klasse `Assertion` mit mehreren statischen Methoden zur Verfügung.

Die wichtigste ist `Assertion.AssertEquals`, mit welcher erwünschte Werte oder Objekte mit resultierenden Werten oder Objekten verglichen werden können. Wir testen damit vor der Schleife, ob die Länge der ArrayList 0 ist, und nach der Schleife, ob tatsächlich 50 Elemente vorhanden sind. Das ergibt für den bereits erstellten Methodenrumpf folgende Erweiterung:

```
  [Test]
    public void TestPopulateStringList() {
      Assertion.AssertEquals(0, Fsl.Count);
      for (int i = 0; i < 50; i++) {
        Fsl.Add("i");
      }
      Assertion.AssertEquals(50, Fsl.Count);
    }
```

Lässt sich das Projekt fehlerfrei kompilieren, dann ist es Zeit, NUnit in der grafischen Oberfläche (NUnit-GUI) zu starten und den ersten Test durchzuführen. Die Oberfläche wartet nur mit dem absolut Notwendigen auf. Mit FILE – OPEN suchen wir die soeben erstellte Projekt-DLL und laden sie in die Testumgebung. NUnit erkennt sofort Testklasse und Testmethode und zeigt sie in Baumdarstellung in der linken Fensterhälfte an. Ein Klick auf die Schaltfläche RUN zeigt uns umgehend, ob wir den Test korrekt aufgesetzt haben.

Ein grüner Balken und grüne Markierungen bei den jeweiligen Tests zeigen, dass alles in Ordnung ist. NUnit lassen Sie nun am besten einfach offen neben der Entwicklungsumgebung laufen. Bei jeder Neuerstellung des Projekts wird NUnit automatisch aktualisiert, sodass nach jeder größeren Codeänderung der Test mit einem einfachen Klick auf RUN wieder durchgeführt werden kann.

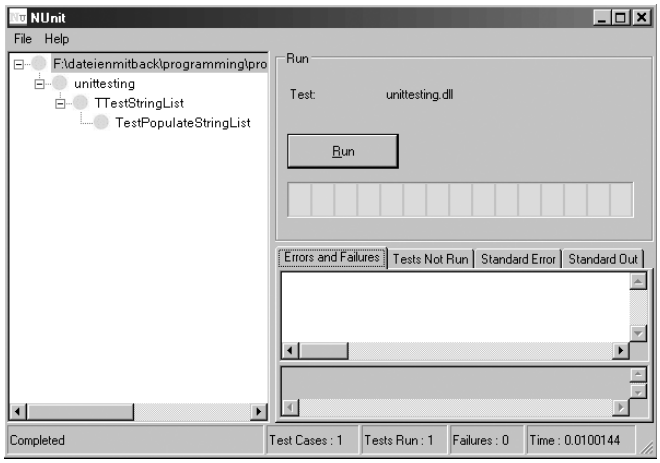

Abb. 1.59: Der erste Test mit NUnit

Nun ergänzen wir unseren Test noch mit einer zweiten Testmethode `TestSortStringList()`, welche die Sortierung von ArrayLists überprüft. Trade wird dabei absichtlich einmal groß und einmal klein geschrieben, um zu zeigen, was bei einem Fehler passiert (Delphi ist zum Glück nicht Case-sensitiv).

```
[Test]
  public void TestSortStringList() {
    Assertion.AssertEquals(0, Fsl.Count);
    Fsl.Add("trade");
    Fsl.Add("Roboter");
    Fsl.Add("Love");
    Fsl.Sort();

    Assertion.AssertEquals("Love", Fsl[0]);
    Assertion.AssertEquals("Roboter", Fsl[1]);
    Assertion.AssertEquals("Trade", Fsl[2]);
    Console.WriteLine("Count Elemente in Fsl: " + Fsl.Count);
  }
}
```

Sobald die Test-DLL neu erstellt worden ist, erscheint die neue Methode in NUnit und lässt sich mit RUN testen. Dieses Mal erscheinen wegen des Fehlers, wie man im zweiten Screenshot sieht, ein roter Balken und viele rote Punkte. Wenn Sie mit der Maus über die Fehlermeldung im Register Errors and Failures fahren, zeigt QuickInfo den vollständigen Fehlertext. Das Fenster ganz unten rechts gibt zusätzlich an, welche Zeile in der Testklasse den Fehler verursacht hat.

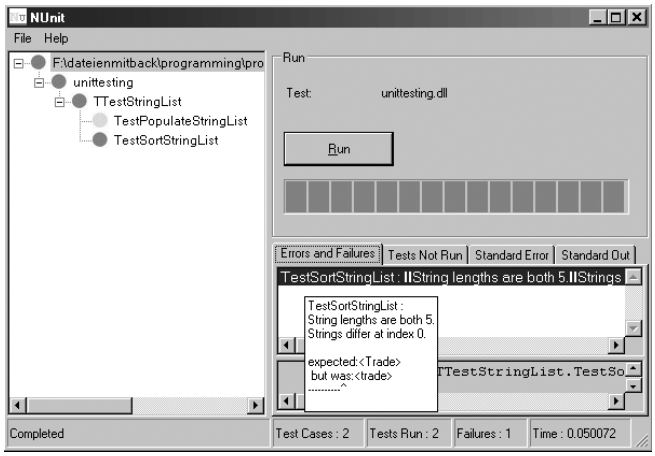

Abb. 1.60: Der erste gefundene Fehler

Wenn man nun Trade in der Testklasse an beiden Orten großschreibt, das Projekt wieder erstellt und den Test noch einmal laufen lässt, dann erscheint wieder ein hoffnungsvolles Grün. Alles weitere zur Bedienung von NUnit entnehmen Sie bitte der Dokumentation zu diesem Programm. Gerade beim Umbenennen von Methoden oder Klassen innerhalb des Refactoring kann auch solch ein trivialer Test seine Berechtigung erhalten.

Abb. 1.61: DUnit und NUnit friedlich vereint

Realistisches Testen

Zum Abschluss noch ein Beispiel mit einem realen Projekt, damit man sieht, wie Testklassen und produktiver Code zusammenspielen. Das Beispiel bezieht sich auf die uCrypto-Klasse, die im Kapitel über das Decorator Pattern genauer vorgestellt wird. Diese Klasse bietet mit `encrypt` eine Methode an, um Textstrings zu verschlüsseln. Es wird behauptet, dass diese Methode symmetrisch sei, d. h., dass die zweifache Verschlüsselung eines Strings wieder den Ausgangsstring erzeugt.

Genau dies wollen wir nun testen. Die Klasse `UCrypto` ist im Projekt „patterns“ eingebunden. Als Erstes erstelle ich in diesem Projekt einen Verweis auf das NUnit Frame-

work. Als Zweites erstelle ich im Projekt eine neue Testklasse `uCryptoTest` mit folgendem Code:

```
using System;
using NUnit.Framework;
namespace ch.kleiner.patternbuch.uCrypto {
  [TestFixture]
  public class uCryptoTest {
    TCrypto objTCrypto;
    [SetUp]
    public void SetUp() {
      objTCrypto = new TCrypto();
      objTCrypto.SetcryptKey(23);
    }
    [Test]
    public void testEncrypt_Symmetrisch() {
      string strTest = "c~d7~d7zvo7qexz7gvo";
      string strEncrypted = objTCrypto.encrypt(strTest);
      Assertion.AssertEquals(strTest,
                  objTCrypto.encrypt(strEncrypted));
    }
  }
}
```

Wenn ich nun *patterns.exe* in NUnit öffne und den Test durchführe, erhalte ich in Grün die Bestätigung, dass mein Verschlüsselungsverfahren tatsächlich symmetrisch ist!

Mit dieser Bestätigung aus dem bewegten Sicherheitsbereich möchte ich Sie zum Teil 2 des Buches einladen, da man mit Sicherheit sagen kann: „Wer sich nicht bewegt, spürt seine Fesseln nicht.“

---

i www.omg.org

ii Kleiner: Designermodell. In: Der Entwickler 1.2001

iii Kruchten: The Rational Unified Process, Addison Wesley, 2000

iv www.agilealliance.org

v Kleiner: UML mit Delphi, S&S Verlag, 2000

vi Kent Beck: Extreme Programming Explained, Addison Wesley, 1999

vii DUnit: http://sourceforge.net/projects/dunit/

viii Fowler: Refactoring – Improving Design of Existing Code. Addison Wesley, 1999

ix Immo Wache: DUnit Rahmenwerk. In: Der Entwickler 5.2001

x www.extremeprogramming.org/rules/pair.html

xi Rothen Silvia: Patterns. In .NET Magazin

xii Das Plus beim Export. In: Der Entwickler 4.2001

[xiii] http://www.delphi3000.com
[xiv] Bucknall Julian: The Tomes of Delphi, Algorithms and Data Structures, 2001
[xv] www.liantis.com, Tau Generation 2, ArcStyler, StP/UML
[xvi] www.csharp-info.de
[xvii] www.delphi3000.com – Exception or Event Logger
[xviii] Architecture & Morality by Orchestral Manoeuvres in The Dark, Dindisc 1981
[xix] Kleiner: Dekoratives Muster – Pattern Generierung. In: Der Entwickler 4/2002
[xx] MetaBASE Framework: www.gs-soft.com
[xxi] Australian Delphi User Group: www.adug.org.au/
[xxii] http://www.nunit.org
[xxiii] http://nunitasp.sourceforge.net
[xxiv] http://www.csunit.org
[xxv] http://www.icsharpcode.net/OpenSource/SD/Download/