2 Design Patterns

Nach der Vorstellung und Einführung in die Technik der Patterns und der intensiven Beschäftigung mit Prozessen, könnten Sie mir nun die Frage stellen, was denn von den strukturierten und objektorientierten Modellen langfristig erhalten bleibt. Nun, das stabilste Element der strukturierten Methoden ist zweifellos die Datenmodellierung, die auch in der Objektorientierung überlebt hat. Nicht mehr aus der Designphase wegzudenken sind das Klassendiagramm und das Sequenzdiagramm als technische Basiskonstrukte für den Softwarebau. Was nach wie vor methodisch fehlt, ist eine fundierte Theorie für Analyse und Design von Komponenten, die aber mit UML 2.0 mehr Gewicht erhält.

Ähnliche Probleme in Softwareprojekten müssten eigentlich zu ähnlichen Lösungen und damit zu ähnlichen Klassendiagrammen führen. Demzufolge ist es angebracht, die Diagramme als Lösungen zu dokumentieren.

Und genau das war die Motivation, bestimmte Klassen mit ihren Beziehungen als Lösungsmuster zu sammeln und **klassifiziert** zu veröffentlichen [DP95][i].

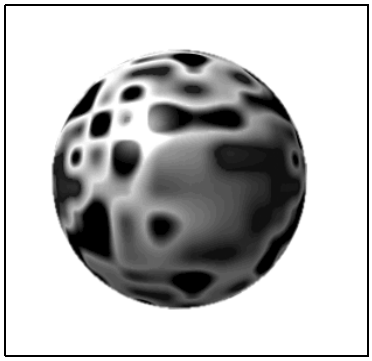


Abb. 2.1: Struktur und Funktion im Muster

Muster oder Patterns gehören zweifellos zu den nützlichsten und erfolgreichsten Konzepten des Software Engineerings. Muster basieren auf gemeinsamen Elementen, die eine gewisse Selbstähnlichkeit enthalten. So basieren das Composite und der Iterator auf dem Element Kinder (Childs), die wiederum in anderen Patterns, wie dem Fliegengewicht, vorzufinden sind.

Ein bekannter Ansatz ist auch Mandelbrots Arbeit über die fraktale Geometrie der Natur, in der die Bedeutung rekursiv definierter Formen und Kurven für viele Lehrbereiche deutlich wird (Mandelbrot, 1982). Wie Mandelbrot mit der Beschreibung von Wolken, Bäumen, Küstenlinien und anderen Formen nachweist, sind fast alle natürlichen Formen durch fraktale Algorithmen darstellbar. Die enge Beziehung zwischen Fraktalen und Formengrammatiken gab den Anlass zur Untersuchung, Muster und ihre Elemente auch in Entwurfsklassen auszuarbeiten.

Die in vielen Projekten enthaltenen Muster (Analyse-, Entwurfs- und Architekturmuster, engl. Pattern), schauen wir uns als Erstes anhand der offiziellen Design Patterns im Code und Diagramm an. Wieder einmal zeigt die Visualisierung, was sie kann. Dass sich Design Patterns mit UML visualisieren lassen, ist ja selbstverständlich.

2.1 Einleitung

„A Pattern is a proven Solution for a general Problem."

Laut Christopher Alexander, Professor für Architektur an der Universität Berkley, ist jedes Pattern eine dreiteilige Regel, welche die Beziehung zwischen einem vorhandenen Kontext, einem Problem und seiner Lösung ausdrückt.

> *Jedes Pattern beschreibt eigentlich die Struktur einer Lösung für ein Problem, das immer wieder vorkommt und somit beschreibend ist.*

Dabei bezeichnet man mit dem Ausdruck Pattern sowohl das Ergebnis, das durch die Anwendung der Regel entsteht, als auch die Regel selbst. Denn es ist effektiv so: Man muss **zuerst** das Problem erkennen, bevor man die Lösung einsetzen kann.

Patterns im Softwarebau lassen sich wiederum in drei Domänen[1] aufteilen (Process, Design, Architectural), die dem vorliegenden Buch die Struktur geben.

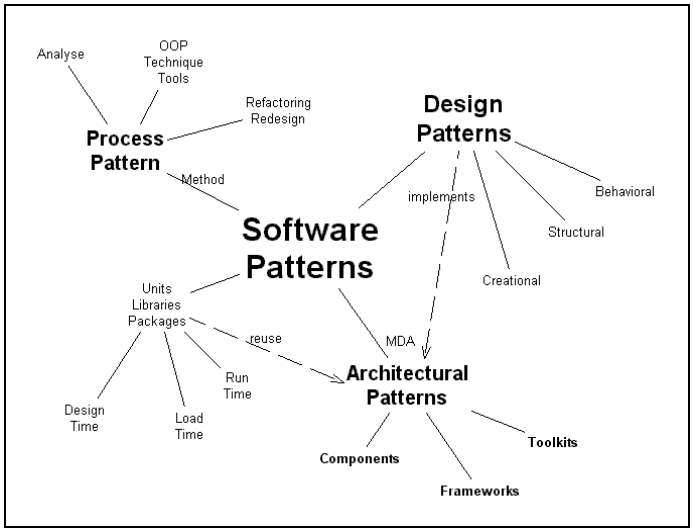


Abb. 2.2: Die Domänen der Patternwelt

[1] Die Idee stammt aus einer Schulung in Chur (Besuch der H. R. Giger Bar).

2.1.1 Wiederverwendung

Natürlich ist auch die Wiederverwendung eine immer wichtiger werdende Technik, indem Software-Elemente oder Komponenten wie Bausteine für die Wiederverwendung entworfen werden. Ein Grund für das „Reuse“ ist sicher der, dass sich der Entwicklungszyklus von Software immer mehr verkürzt und zudem beschleunigt.

Meine Partnerin, auch in der Entwicklung tätig, hat das in einem Artikel folgendermaßen beschrieben: „Immer wieder in meiner Entwicklerarbeit bin ich auf die Aufgabe gestoßen, irgendwelche Operationen an den Dateien und Verzeichnissen in einem Verzeichnis und in seinen Unterverzeichnissen vorzunehmen.

Einmal ging es darum, die Verzeichnisstruktur (ohne Dateien) von einem Verzeichnis in ein anderes zu kopieren. Ein anderes Mal wollte ich alle Bilddateien in einem Verzeichnisbaum finden und mit gewissen Informationen (Bildgröße, Farbtiefe, Format) in einer Datenbank archivieren.

Zur Dokumentation einer Website wiederum ist eine Routine nützlich, die alle vorhandenen Dateien mit ihren Formaten zusammenstellt. In Scriptsprachen gab es für diese Aufgaben vor allem das CPA-Verfahren, d. h. COPY – PASTE – ADAPT (vermutlich immer noch die meistverwendete Technik der Software-Entwicklung). Mit objektorientierten Sprachen gibt es stattdessen die Möglichkeit, einmal eine Klasse zu schreiben und sie dann für beliebige Anwendungen zu verwenden.“

Wenn dann in dieser Situation Nächte vergehen, um die grundlegenden Klassen erst noch zu entdecken, wird zu viel Zeit darauf verwendet und diese Zeit geht dann auf Kosten der Qualität, denn damit wird die Testphase verkürzt.

> *Nun gibt es mit Patterns einen Weg, nicht die fertigen Komponenten, aber zumindest die Lösungsidee in den Klassenbau einfließen zu lassen. Patterns bieten eine objektorientierte und erfolgreich eingesetzte Lösung für bestimmte Probleme, wobei die Sprache nicht festgelegt ist. Jede Sprache hat demzufolge ihre eigenen „Code Patterns“.*

Zum anderen gibt es bspw. die CLX, ein Komponentenmodell, das basierend auf einer Klassenhierarchie visuelle und nicht visuelle Bausteine anbietet. Nur sind die Komponenten zu konkret, um z. B. zu einer vernünftigen Architektur zu gelangen. In der VCL und der CLX selbst lassen sich wie in den meisten OO-Bibliotheken einige Patterns erkennen.

Neben den folgenden 24 Beispielen, in denen ich von den Design Patterns zu den konkreten Code Patterns komme, gibt es noch eine weitere Art von Patterns, teilweise auf konkreterer Ebene, wie z. B. die Frameworks oder Toolkits, die ich in Teil 3 beschreibe.

Zudem ist, wenn eine Lösung sich bewährt hat, ein Katalogisieren zum Auffinden der Patterns nötig. Denn es hilft wenig, neue Klassen zu entwerfen, die vielleicht schon seit geraumer Zeit vorhanden sind.[ii]

2.1.2 Nutzen

Patterns werden von erfahrenen Entwicklern schon lange verwendet. Zum Teil verwendet man Patterns auch, ohne sich dessen bewusst zu sein. Das Problem bis vor einigen Jahren war, dass jeder Entwickler seine Muster selbst mehr oder weniger kreativ aufbauen muss-

te, da keine geeignete Möglichkeit bekannt war, das Wissen in Form von Lösungsansätzen in definierter Form **weiterzugeben**.

Patterns bieten nun diese Möglichkeit. Ihre Verwendung zur Beschreibung von Problem-Lösungs-Beziehungen bringen laut [DP95] folgende vier großen Vorteile:

Ein gemeinsames Design-Vokabular

Patterns bieten durch ihren festgelegten Namen eine einfache Möglichkeit zur Kommunikation und Dokumentation. Kennen mehrere Entwickler bestimmte Patterns, dann reicht der Name aus, um über diverse Alternativen zu sprechen. Es ist eben sinnvoll, ein allgemein gültiges Glossar zu haben. Wir haben einmal einen Fall erlebt, wo der Begriff „Adapter“ im Zusammenhang mit inkompatiblen Schnittstellen den Review schnell beendet hat.

Dokumentations- und Lernhilfe

Bei großen OO-Projekten werden oft unbewußt Design Patterns eingesetzt. Beschreibt man dieses System mit Design Patterns und den entsprechenden Diagrammen, lassen sich im Nachhinein mit den gefundenen Patterns die vorhandenen Funktionen schneller und verständlicher erklären (auch ohne Reverse).

> *Da Design Patterns für allgemeine, stets wiederkehrende Probleme Lösungen anbieten, liegt es nahe, sie in der Lehre im Fach Softwarebau oder -Engineering einzusetzen.*

Auch in der Chartanalyse, einer Technik zur Kursprognose von Wertpapieren anhand historischer Charts, versucht man doch aus den Mustern der Vergangenheit etwas „Künftiges“ zu lernen. Muster haben immer Strukturähnlichkeit.

Erweiterung zu bestehenden Techniken

Existierende Design-Techniken beschreiben Probleme, die während des Designs auftreten können, und bieten dazu passende Lösungen an. Sie vernachlässigen jedoch den Bereich der Erfahrung während der Entwicklung, d. h., wie man auf eine Lösung kommt, wird eher schematisch als empirisch beantwortet. Design Patterns bringen in der Design-Phase folgende Vorteile:

- Design Patterns beinhalten Erfahrung von Experten und Praktikern.
- Design Patterns beschreiben, warum eine bestimmte Alternative, d. h. ein Pattern, verwendet wird und welche Konsequenzen sich daraus ergeben. Daher lassen sich Entscheidungen später leichter nachvollziehen.
- Design Patterns bringen flexible und wieder verwendbare Strukturen aus der Design-Phase in die Analyse-Phase, indem z. B. aus der Struktur ein Ablauf gewonnen wird. Dies ist ein interessanter Punkt, denn dadurch lässt sich der Übergang vom Use Case zum Klassendiagramm vereinfachen.

Releasewechsel optimieren

Je länger eine Software im Gebrauch ist, desto mehr neue Anforderungen werden an sie gestellt. Vielfach werden zusätzliche Anforderungen durch Erweiterungen der Software erfüllt. Dies führt aber mit der Zeit zu einem „Gebastel“, welches für spätere Anpassungen ungeeignet ist.

Um die Anwendung weiterzuentwickeln, muss man sie irgendwann restrukturieren.

> *Patterns sind nicht nur bei der Neuentwicklung von Software nützlich, sondern gerade auch beim Redesign: Design Patterns beschreiben Strukturen, die ein Software-Design flexibler und wieder verwendbarer machen.*

Eine Stärke von Patterns liegt darin, dass man sie parametrisieren kann. Parameter sind Größen, mit denen sich Objekteigenschaften innerhalb gesetzter Grenzen variieren lassen, ohne dass sich der Charakter der Objekte dabei grundlegend ändert.

Dadurch zeigen sie den Weg, wie ein unflexibles System restrukturiert werden kann, um mit bekannten Mustern auch zukünftigen Anforderungen gewachsen zu sein.

2.2 Pattern-Katalog

Damit Patterns in einem definierten Sinne zur Verfügung stehen, braucht es einen Katalog oder ein Archiv. In diesem Zusammenhang sind einige der Begriffe kurz zu erläutern. Ein Repository ist eine Art Lager für wieder verwendbare Software-Bausteine. Außerdem müssen die Beschreibungen in einer festgelegten Form erfolgen, damit man die Patterns miteinander vergleichen und in ein Schema einordnen kann.

Eine „Komponente“ ist ein eher problematischer Begriff, da er sowohl ganz allgemein ein physisches Softwareteil als auch das Prinzip einer Architektur mit standardisierten Schnittstellen und eigenen Editoren nach außen meint, also ein genau spezifiziertes Software-Objekt bezeichnet.

2.2.1 Einteilung

Üblicherweise unterscheidet man in einem Pattern-Katalog pro Muster die folgenden sechs fundamentalen Bestandteile. Im vorliegenden Buch werde ich jedoch ähnlich dem Original Problem, Kontext und Lösung unter dem Titel "Motivation" zusammenfassen, da so ein mittelbarer Zusammenhang zur jeweils folgenden Implementierung erkennbar ist. Die Idee des Abschnittes Motivation ist auch, einen generellen Einsatz des Patterns zu schildern, ohne direkt auf ein Beispiel zu steuern.

Eine Anmerkung zum Original: Die im Buch der GoF enthaltene Notation in Klassen- und Sequenzdiagrammen basiert auf einer veralteten Notation, die nicht UML-konform ist.

Mustername

Der Mustername benennt das Problem, die Lösung und die Konsequenzen in ein bis zwei Worten. Er erweitert das Vokabular von Entwicklern und Methodikern, die sich unter

dem festgelegten Namen das Muster mit den Strukturen und Interaktionen vorstellen können. Der Name soll das Problem und seine Lösung bezeichnen. Man sollte ihn sorgfältig wählen, da er direkt in das Vokabular aufgenommen wird. Ich füge hier jeweils als Zweck ein generelles Diagramm hinzu, das sofortige Übersicht bietet. Mit dem konkreten Diagramm bei den Beispielen lässt sich dann der Vergleich ziehen.

Problem

Das Problem beschreibt das Umfeld des Musters. Es wird zuerst der Kern des Problems allgemein beschrieben. Danach lassen sich Erfordernisse, Konditionen und wünschenswerte Eigenschaften des Ergebnisses beschreiben. Es werden die Kriterien für die Anwendung des Musters angegeben.

Kontext

Der Kontext ist eine Beschreibung der möglichen Situation, in der das Pattern sich einsetzen lässt. Meiner Meinung nach sollte im Zusammenhang mit dem Kontext auch immer ein Beispiel der Umgebung inklusive der Konstellation der Infrastruktur folgen. Und wie kann man vor allem die Situation erkennen, die zur Lösung führt?

Lösung

Die Lösung spezifiziert die Elemente, die den Entwurf des Musters ausmachen, ihre Beziehungen und die Interaktion bei der Lösung des konkreten Problems. Es folgt das grundsätzliche Prinzip, das der Lösung zugrunde liegt.

Konsequenzen

Konsequenzen ergeben sich aus der Anwendung des Patterns. Die Konsequenzen benennen Vor- und Nachteile der eingesetzten Lösung und sind wichtig für die Auswahl des richtigen Patterns. Es ist unwahrscheinlich, dass ein Pattern die Probleme vollständig lösen kann. Darum ist es hier wichtig, auf die vom Pattern ungelösten Probleme einzugehen. Die Beschreibung der Konsequenzen ist zudem das Kriterium für die Auswahl bei alternativen Patterns.

Struktur und Beispiel

Hier werden die beteiligten Komponenten und ihre Beziehungen untereinander beschrieben. Es wird in der Regel eine grafische Repräsentation zur Beschreibung verwendet. Ein Codebeispiel zeigt die praktische Umsetzung des Designs. Auch Tipps oder Techniken zur Implementation sind hier erwähnt. Das Beispiel hilft, die abstrakte Beschreibung des Patterns leichter zu verstehen.

Die Beispiele in C# sind außer im Code nicht weiter dokumentiert, da in den meisten Fällen auch die Beschreibung von Delphi zutrifft.

Weitere Bestandteile

Im Original sind noch mehr Bestandteile vorhanden, die vor allem die dynamischen Eigenschaften ausleuchten. Es sind dies die **Struktur** des Patterns, welche die Konfiguration der beteiligten Teile und ihre Beziehungen untereinander beschreiben. Im Buch erscheint die Struktur als einleitendes Klassendiagramm. Das **Laufzeitverhalten** geht auf die Zusammenarbeit der Komponenten ein, und die **Implementation** zeigt Hinweise auf mögliche Probleme und Besonderheiten beim Codieren des Patterns. **Varianten** sind eine kurze Beschreibung von Varianten oder Spezialisierungen des Patterns. Teilweise finden sich auch Codeteile aus bekannten Anwendungen als Beispiele zur Illustration einer möglichen Implementation.

2.2.2 Kategorisierung

Nun kommen wir zu den konkreten Patterns, die sich nach Aufgabenbereichen kategorisieren lassen. Nach dem Umfang werden Patterns für Klassen (z. B. Fabrikmethode) und für Objekte unterschieden, da man ausschließlich objektorientierte Patterns beschreibt. Die Muster haben also einen Gültigkeitsbereich, der klassen- oder objektorientiert ist. Nach Einsatz und Zweck sind Patterns in folgende Kategorien eingeteilt:

Creational Patterns beschäftigen sich mit der dynamischen Erzeugung und Wiederbelebung von Objekten unabhängig der Typen. Im Falle des TRadeRoboter:

- Abstract Factory
- Builder
- Factory Method
- Prototype (als Architekturmuster)
- Singleton

Structural Patterns beschreiben den statischen Zusammenhang von Objekten und Klassen, welche auch andere Klassen binden oder zu Strukturen zusammenführen. Strukturen sind eine Voraussetzung für Funktionen:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy
- Wrapper

Behavioral Patterns (Verhaltensmuster) charakterisieren das dynamische Verhalten von Objekten und Klassen. An dieser Stelle sind Patterns aufgeführt, die Algorithmen oder die Flusskontrolle beschreiben und steuern. Es sind die schwierigeren Muster:

- Chain
- Command
- Interpreter
- Iterator

- Lock
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor

Alternative Kategorisierung

Frank Buschman und seine Kollegen (Buschmann and Meunier, 1995; Buschmann et al., 1996) teilen Patterns nach ihrem Abstraktionsgrad ein. Er beginnt mit der höchsten Abstraktionsstufe und nennt diese **Architektur-**Patterns. Mehr dazu im dritten Teil des Buches. Diese beschreiben und erklären die fundamentale Struktur von Software-Systemen. Sie geben die nötigen Subsysteme an und definieren Funktionen und Beziehungen zueinander.

In der mittleren Abstraktionsebene werden die effektiven Design Patterns verwendet. Diese zeigen, wie sich Subsysteme, Module oder Komponenten verfeinern lassen. Sie lösen die immer wiederkehrenden Entwurfsprobleme.

Die geringste Abstraktion weisen so genannte **Idiome** auf. Idiome (Idioms) lassen sich spezifisch im Zusammenhang mit einer Programmiersprache anwenden, z. B. für Delphi die Frames oder das Exception Handling. Wichtige Idiome für eine Sprache können für eine andere wertlos bis belanglos sein. Man kann diese Idiome auch unter dem Begriff Code Patterns antreffen.

Man verwendet auch den Begriff des Analysemusters, der zuerst ohne jeden Praxisbezug daherkommt. Wie der Name sagt, wird Unterstützung während der Analyse eines Projektes gegeben, konkret können das fertige Use Cases oder ein Aktivitätsdiagramm für ein Fachproblem sein. Beispielsweise hat man eine Maßzahl in einer Auftragsbearbeitung wie Preis, Anzahl, Mindestmenge oder Umsatz zu definieren. Das triviale Analysemuster besagt nun, dass man nicht nur die Zahl als Mengenangabe speichern soll, sondern gleichzeitig auch die Einheit der Zahl, damit später in der Implementierung die Umrechnung klar ist und die Einheit austauschbar bleibt:

```
Zahl: 0470 / ArtikelXYZ / in Tonnen
Zahl: 4600 / Artikel_G / in Kilo
Zahl: 5600 / ArtikelZZ-Top /in Euro
```

2.3 Erzeugungsmuster

In Design Patterns lassen sich erfolgreiche Lösungen zu bestimmten, immer wiederkehrenden Problemen in der Software-Entwicklung bereitstellen.

Ich starte nun mit den einzelnen Patterns, die ich anhand der Beispiele in Modell und Code einzeln näher bringe. Als schöpferische Tätigkeit gelten sicher die Erzeugungsmuster.

Klassen und Typen

Kurz noch eine Definition im Zusammenhang mit der Erzeugung. Zwischen Klassen und Typen besteht ein Unterschied, der in vielen Programmiersprachen nur schwach zum Tragen kommt. Design Patterns basieren aber auf einer klaren Unterscheidung dieser beiden Begriffe. Klassen repräsentieren die Implementation eines Objekts, während Typen nur die Schnittstellen eines Objekts beschreiben.

Daraus resultiert, dass man bei einer Klassenvererbung auch eine zugehörige Implementation miterbt. Dies hat zwei Nachteile: Der eine ist, dass die erbende Subklasse mit einer engen Kopplung an die Superklasse gebunden wird und sich damit in eine Abhängigkeit begibt. Der zweite Nachteil ist, dass in Sprachen ohne Mehrfachvererbung nur von einer Klasse die Implementierung geerbt werden kann. Moderne OO-Sprachen lösen dieses Dilemma, indem sie die Möglichkeit anbieten, Schnittstellen zu vererben.

Dies hat zwei enorme Vorteile:

- Clients müssen nicht wissen, welchen spezifischen Typ die von ihnen verwendeten Objekte haben, solange diese Objekte die Implementation erfüllen, d. h. die Schnittstelle unterstützen.
- Clients wissen auch nicht, mit welchen Klassen diese Objekte implementiert sind. Sie kennen nur die abstrakten Klasse, welche die Schnittstellen definieren.

Durch die Schnittstellenvererbung umgeht man die Abhängigkeit von einer bestimmten Implementierung. Dies führt zu einem wichtigem Paradigma in der OO-Welt:

> *Program to an Interface, not an Implementation.*

Die nun folgenden Patterns sind der Kategorisierung nach eingeteilt in:

- Erzeugungsmuster (creational)
- Strukturmuster (structural)
- Verhaltensmuster (behavioral)

Es folgt bei jedem Pattern eine Kurzbeschreibung, die nach den Vorgaben des Originals – Zweck, Motivation, Verwendung – strukturiert ist. Die Verbindung des generellen Diagramms mit dem spezifischen Klassendiagramm des Beispiels erscheint mir wichtig, sodass der Abbildungstext jeweils die Verbindung zum Namen signalisiert. Sequenzdiagramme, wo nötig, und der zugehörige Code vervollständigen die 24 Design Patterns. Gut die Hälfte der Muster sind auch in C# codiert. Interessanterweise gibt es eine Analogie zwischen den Betriebsarten von Code und der Kategorisierung von Design Patterns:

- Erzeugungsmuster (creational) – zur Load Time
- Strukturmuster (structural) – zur Design Time
- Verhaltensmuster (behavioral) – zur Run Time

Die Analogie besagt, dass die Einteilung von Design Patterns mit einer gewissen Zeitphase im System übereinstimmt.

2.3.1 Abstract Factory

Ein objektbasiertes Erzeugungsmuster (Abstrakte Fabrik erzeugt Produkte).

Zweck

Ermögliche eine Schnittstelle, die zum Erzeugen von verwandten Objekten oder Methoden dient, ohne ihre konkreten Klassen zu kennen.

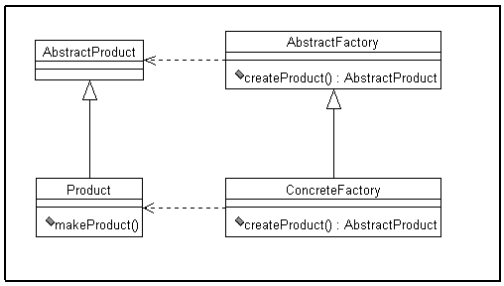


Abb. 2.3: Abstract Factory auch Konkret

Motivation

Betrachten wir eine Klasse oder eine Gruppe von Klassen, welche die gleichen Member enthalten sollen, aber je nach Anwendung unterschiedlich entstehen müssen. Diese Klasse nennt man die Fabrik und die unterschiedlichen Erzeugnisse daraus die Produkte. Zum Beispiel will man ein Dokument unterschiedlich drucken, abhängig von der Art des Druckers; der Inhalt des Dokuments ist aber jeweils gleich.

Die Grundidee ist folgende: Eine abstrakte Fabrik definiert die Schnittstelle zur Erzeugung abstrakter Produkte. Die Erzeugung einer Familie von konkreten Produkten erfolgt aber in den konkreten Fabriken. Damit lassen sich ganze Familien von Objekten auswechseln, indem man einfach die konkrete Fabrik auswechselt.

Diese konkrete Fabrik erzeugt dann die Produktobjekte, welche spezifische Implementierungen haben. Die gemeinsame Schnittstelle für die Übergabe der konkreten Fabrik könnte so aussehen:

```
type
TPanelMaker = class (TObject)
 public
   function CreatePanel(aOwner: TWinControl;
                    Factory: TAbstractFactory ): TPanel;
end;
```

Die vorher erzeugte konkrete Fabrik wird dem Konstruktor des `TPanelMaker` (Abstraktes Produkt) übergeben, sodass die konkrete Fabrik ein ganzes Panel mit Editboxen, Labels etc. als konkrete Produkte erzeugen kann.

Dadurch dass der Client nicht direkt auf die konkreten Produkte (Objekte wie Buttons, Editboxen, Labels etc.) zugreift, kann jede weitere Erweiterung mit bestehenden oder neuen Fabriken ohne Änderung der Konstruktorschnittstelle erfolgen. Noch größer ist der

Vorteil, wenn das Erstellen der Baukiste und deren Produkte zur Laufzeit wechseln müssen, sodass man einfach eine andere konkrete Fabrik instanziert und der bestehenden Schnittstelle übergibt.

Erst bei n-Produkten, die sich jeweils von n-Fabriken erzeugen lassen, leuchtet die Existenz einer gemeinsamen Schnittstelle (siehe Abb. 2.5) wie `TPanelMaker` ein.

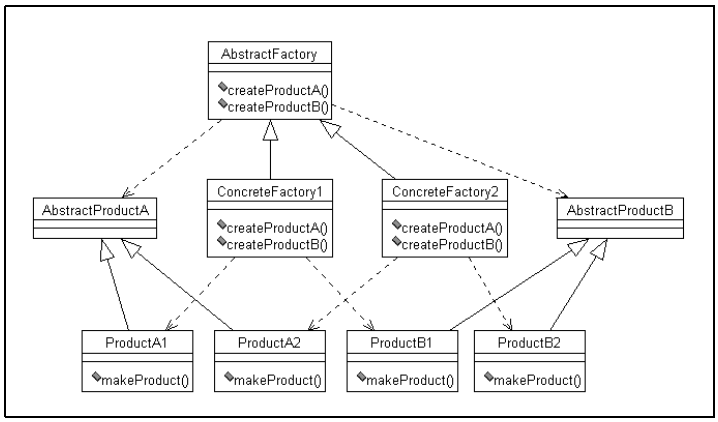


Abb. 2.4: Abstract Factory ausgebaut

Daraus ableitend sind Kombinationen möglich, wie sich eine Fabrik die einzelnen Objekte zusammenbaut, wobei die Kombination zur Designzeit bekannt sein muss. Meistens benötigt eine Anwendung genau eine konkrete Fabrik pro Produktefamilie, demzufolge man die Fabrik zusätzlich als Singleton implementiert.

Das Erzeugen der Produkte ist nicht auf GUI-Elemente beschränkt, vielfach ist die Verwaltung von Ressourcen wie Files, Sockets oder Ports als Fabrik realisiert.

Implementierung

Ausgehend vom Erzeugen eines Panels, lässt sich die Funktion dann konkretisieren. Die übergebene `Factory` produziert die in unserem Fall grafischen Controls und setzt noch den `Owner` mit. Weil die erzeugte Instanz `Factory` von der abstrakten Fabrik stammt, sind weitere Fabriken immer typenverträglich zu den Fabrikmethoden. Das fertig erzeugte Panel wird dann als Referenz dem Aufrufer zurückgegeben.

```
function TPanelMaker.CreatePanel(aOwner: TWinControl;
                          Factory: TAbstractFactory): TPanel;
var
  TmpPanel: TPanel;
  TmpEdit1, TmpEdit2, TmpEdit3: TEdit;
begin
  TmpPanel:= Factory.MakePanel(aOwner);
  TmpEdit1:= Factory.MakeEdit(TmpPanel, 'Test1',10,10);
```

```
   TmpEdit2:= Factory.MakeEdit(TmpPanel, 'Test2',
                        TmpEdit1.Top + TmpEdit1.Height + 10,
  .....
  CreatePanel:= TmpPanel;
end;
```

Im Klassendiagramm sind die Fabrikmethoden `MakeEdit` und `MakePanel` sichtbar:

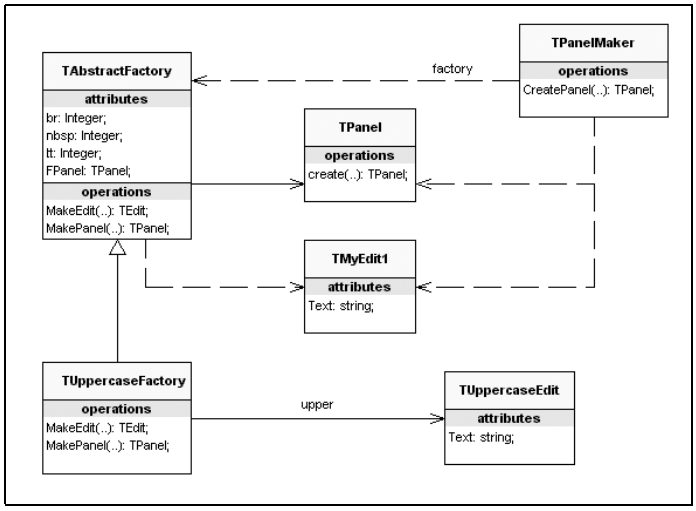


Abb. 2.5: Abstract Factory mit konkreten Produkten

Die abstrakte Fabrik selbst beinhaltet die erwähnten gemeinsamen Member, die aber unterschiedliche Produkte erzeugen können. In Abb. 2.5 ist die eigentliche Abstrakte Fabrik als Klasse nicht zu sehen, d. h., `TAbstractFactory` ist bereits eine Spezialisierung davon mit eigenen Instanzen. Hier darf man den semantischen Begriff Abstrakte Fabrik nicht unbedingt mit dem technischen Begriff der abstrakten Klasse gleichsetzen.

Eine abstrakte Methode ist in Delphi eine virtuelle oder dynamische Methode, die man nicht in der Klasse selbst implementiert, in der sie deklariert ist. Die Implementierung wird erst später durch eine Definition in einer abgeleiteten Klasse durchgeführt.

```
type
 TAbstractFactory = class (TAbstractGeneric)
 protected
   FPanel: TPanel;
 public
   function MakeEdit(aOwner: TWinControl;
                     vText: string;
                     vTop, vLeft: Integer): TEdit; virtual;
   function MakePanel(aOwner: TWinControl): TPanel; virtual;
 end;
```

Das weitere Hinzufügen der konkreten Fabrik `TUppercaseFactory` verändert unsere Schnittstelle `CreatePanel` nicht mehr. Unser Panel erhält als Ergänzung eine Editbox als Klasse mit automatischer Konvertierung in Großbuchstaben. Ein Abstraktes Produkt ist nicht vorhanden, da die Hierarchie mit `TWinControl` oder `TWidget` genügend Gemeinsamkeiten aufweist. Erstellen Sie Instanzen von `TWinControl` nicht direkt. Dieser Typ dient ja lediglich als Basisklasse für fensterorientierte Steuerelemente.

Die folgende Klasse ist ein Beispiel eines konkreten Produktes (in den meisten Fällen benötigt man auch das Subclassing nicht, d. h., das Produkt ist eine direkte Instanz der Komponente, z. B. von `TEdit`):

```
TUppercaseEdit = class (TEdit)
  protected
    FOnChange: TNotifyEvent;
    procedure Change; override;
    function GetUppercase: string;
    procedure SetUppercase(Value: string );
  public
    property Text: string read GetUppercase write SetUppercase;
end;
```

Verwendung

Ein generelles Vorwort zum jeweiligen Abschnitt Verwendung: Es geht hier nicht um das Erlernen oder Verstehen der Beispiele. Das Material kommt teilweise aus dem brutalen Umfeld echter Komponenten ;). Ich möchte einfach den Eindruck vermitteln, wie Patterns jenseits von Schulbeispielen in der harten Realität zum Tragen kommen. Die Verwendung zeigt auch nicht immer die reine Anwendung des Patterns, die den Puristen zufrieden stellen könnte. Als Schulbeispiel dient der Abschnitt Implementierung, wobei auch hier angewandte, sprich schwierigere, Beispiele Einzug finden können.

Eine generelle Verwendung ist bei Systemen anzutreffen, die Komponenten für 16-, 32- oder 64-Bit-Applikationen anbieten. Teilweise gibt es die VCL beziehungsweise die `FreeCLX` auch auf DOS mit dem DOSExtender, sodass ein generischer Layer die konkrete Fabrik übernimmt.[iii]

Im Falle eines Webscriptes ist das Erstellen von einzelnen Objekten unabdingbar. Das eigentliche Produkt ist die Website oder das Script, welches durch die Factory im entsprechenden Kontext entsteht. WebSnap beinhaltet einige Abstrakte Fabriken.

```
Result:= TAbstractScriptObjectFactory(FObjectList[I]);
```

Alle WebSnap-Module werden mithilfe eines Factory-Objekts für die WebSnap-Anwendung registriert. Das Factory-Objekt wird in der Regel im Initialisierungsabschnitt der Quelltextdatei der Unit erstellt. Durch Setzen entsprechender Flags für die Factory-Objekte können Sie festlegen, wann das Objekt Instanzen, d. h. konkrete Produkte, seines Webmoduls erzeugen soll und ob es Webmodulinstanzen für die Wiederverwendung mit späteren Anforderungsbotschaften zwischenspeichern kann.

Im Weiteren sind Abstrakte Fabriken im Einsatz, wenn Scripting von WebSnap-Adaptern zur Generierung dynamischer Seiten in Serveranwendungen im Web verwendet wird. Obgleich das „ServerSideScripting“ einen wertvollen Bestandteil von WebSnap darstellt, ist seine Verwendung in WebSnap nicht unbedingt erforderlich. Das Scripting dient ausschließlich zur Generierung von HTML-Produkten.

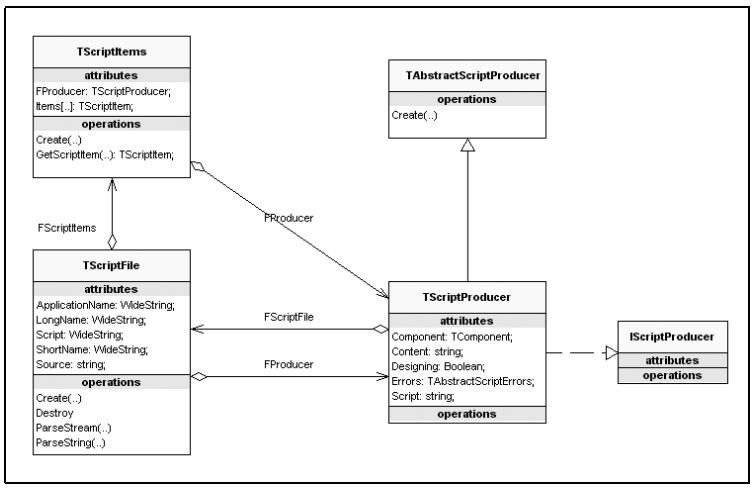


Abb. 2.6: Die Fabrik produziert Scripte

Nachkommen von `TBaseWebModuleFactory` generieren Webmodule, die Webseiten erstellen. Diese Webmodule können Komponenten (z. B. Adapter) enthalten, die bestimmte Operationen für die WebSnap-Anwendung ausführen oder Scripte aktivieren.

Erzeugen Sie keine Instanzen von `TAbstractWebModuleFactory`. Die meisten Methoden dieses Objekts sind abstrakt bzw. rein virtuell (C++-Terminologie), also nicht implementiert.

WebSnap-Anwendungen können mehrere so genannte Webseitenmodul-Factories (also auch mehrere Webseitenmodule) enthalten, die jeweils Webseitenmodule für unterschiedliche Webseiten erstellen, welche die Anwendung generiert. Im Folgenden sieht man den Aufruf der Fabrik über die gemeinsame Methodenschnittstelle:

```
procedure SetFactory(AFactory: TAbstractWebPageModuleFactory);
                                                    override;
begin
inherited;
if FName = '' then
  FName:= Copy(Factory.ComponentClass.ClassName, 2, MaxInt);
if (FFile <> '') then
  if AnsiCompareFileName(ExtractFileExt(FFile),FFile) = 0 then
    //FFile is a file ext
```

```
    FFile := ChangeFileExt(GetTypeData
         (Factory.ComponentClass.ClassInfo)^.UnitName, FFile)
end
```

Das in WebSnap verwendete Scripting ist objektorientiert und unterstützt bedingungsabhängige Logik und Schleifen, wodurch sich die zur Generierung der Seite erforderlichen Tasks wesentlich vereinfachen lassen.

In der Unit *webscript.pas* ist in der Klasse `TScriptObjectFactories` eine extreme Methode zu finden, welche die verschiedenen Fabriken in einer Liste verwaltet und durch die besagte Methode `GetFactory` dem Aufrufer ermöglicht, sich zuerst via Index die Fabrik zu holen, um später die globalen Produkte wie Page, Session, Request, EndUser oder ScriptEngine zu erzeugen.

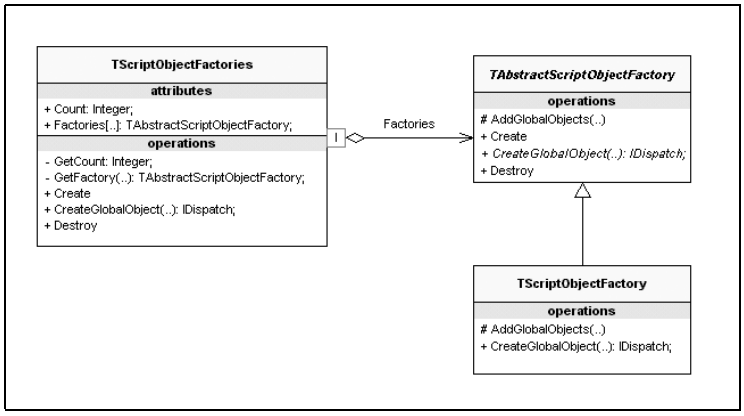


Abb. 2.7: Fabriken werden in einer Liste gehalten

Die `ScriptEngine` gibt die Script-Sprache an, die von dem Generator in den erzeugten Dokumenten verwendet wird. Wenn der Generator z. B. Code mit JScript erzeugt, enthält die Eigenschaft `ScriptEngine` den Wert `JSCRIPT`.

Wenn `ScriptEngine` definiert ist, erzeugt der Seitengenerator Inhalt mit einer speziellen Script-Generatorkomponente. Ist `ScriptEngine` nicht definiert, verwendet der Seitengenerator nur das Ereignis `OnHTMLTag` und alle anderen integrierten Übersetzungen, die von einer abgeleiteten Klasse bereitgestellt werden.

Aus den anfänglich einfachen Produkten wie dem Panel sind mittlerweile so stattliche Erzeugnisse wie Scripte, HTMLItems oder Web-Kontexte entstanden, die in „Real Life“ in der Klasse `TScriptProducer` deutlich zeigen, wohin die mächtige Technik einer Abstract Factory führen kann:

```
inherited Create(AWebModuleContext, AStripParamQuotes,
                 AHandleTag, AScriptEngine, ALocateFileService);
...
FContent:= TMemoryStream.Create;
FLocateFileService:= ALocateFileService;
```

```
FScriptEngine:= AScriptEngine;
if FScriptEngine = '' then
  FScriptEngine:= 'JSCRIPT'; //DSCRIPT
FErrors:= TScriptErrors.Create;
FHTMLItems:= THTMLItems.Create(Self);
FScriptFile:= TScriptFile.Create(Self);
FIncludeFiles:= TScriptIncludeFiles.Create(Self);
FWebModuleContext:= AWebModuleContext;
FHandleTag:= AHandleTag;
```

2.3.2 Builder

Ein objektbasiertes Erzeugungsmuster (Erbauer erzeugt ein komplettes Produkt).

Zweck

Trenne die Konstruktion eines Objektes von seiner Repräsentation, sodass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

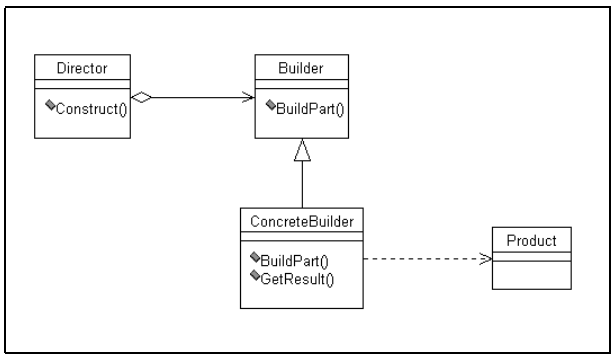


Abb. 2.8: Director lässt ein komplettes Produkt erzeugen

Motivation

Ein mächtiges Muster, welches nicht nur die visuellen Eigenschaften einer Konstruktion optimiert. Vom Konzept her ähnlich der Abstrakten Fabrik, wobei die Fabrik ganze Familien von konkreten Klassen oder Teilen erzeugt. Der Builder ist an einzelnen Methoden der gesamten Builderklasse interessiert, gibt aber ein konkretes und meistens kompliziertes Produkt mit `GetResult()` einzeln zurück. Wie das Objekt zusammengebaut wird, obliegt dem Builder.

Mit dem Hausbau vergleichbar produziert die Fabrik verschiedene Fenster oder Türen in Größe, Stil und Farbe für diverse Hausmodelle, der Builder jedoch erzeugt ein konkretes Haus mit den zugehörigen resultierenden Fenstern und Türen.

Man kann sich z. B. auch einen kompletten RTF-Leser vorstellen, der in Aggregation mit einem Konvertierer (als Builder) steht und RTF interpretieren kann. Der Konvertierer selbst ist dann fähig, unterschiedliche Repräsentationen zu erzeugen, wie ASCII, XML oder eine TeX-Datei. Nicht immer ist übrigens beim Builder eine gemeinsame abstrakte Oberklasse nötig.

Implementierung

Im Falle TRadeRoboter bezweckt der Builder das Erzeugen eines Forms mit Trackbars und einem Panel mit drei Progressbars, die einzeln steuerbar sind. Die gemeinsame Oberklasse fehlt in Abb. 2.9, sodass der Direktor direkt die Klasse `TSingletonForm` aufruft und als Resultat das gesamte Formular oder Form erhält.

Durch Änderung des `Owner` oder der Eigenschaften (Rezepturen) lassen sich durch `override` verschiedene Repräsentationen erzeugen. Die einzelnen Methoden sind entsprechend virtuell. Da ich keine gemeinsame Oberklasse einsetze, lässt sich die Methode `CreateBars` auch statisch aufrufen.

Reihenfolge und Konstruktion der Objekte bleiben gleich. Einzelne Klassen lassen sich im Konstruktor auch gezielt aktivieren oder deaktivieren, damit sind Kombinationen visueller Art möglich. Bei einer Buchung auf ein Konto kann also die Anzeige der Bars unterschiedlich reagieren.

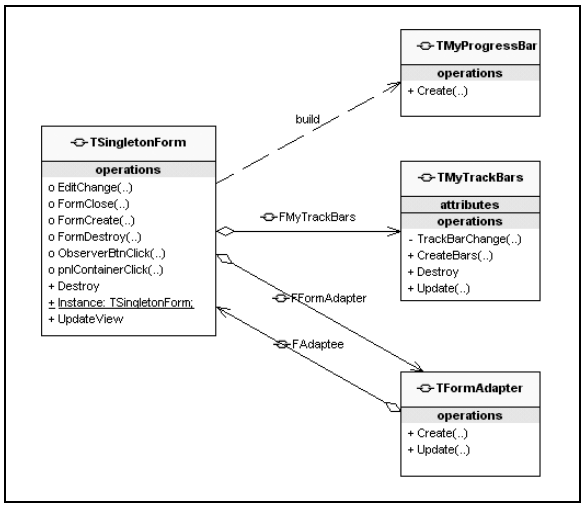


Abb. 2.9: Der Builder erzeugt ein Form mit Produkten

Erzeugt wird das `TSingletonForm`, welches gleichzeitig den Builder darstellt, durch eine Klassenmethode des Builders. Diese ruft den Konstruktor auf, der wiederum mithilfe eines Panels die einzelnen Produkte konstruiert und aggregiert.

Viele Objekte arbeiten mit anderen zusammen. Deshalb ist es bequem, gleichzeitig die Eigenschaften mehrerer Objekte festlegen zu können. So kann z. B. einer Panel-

Komponente ein Pop-up-Menü als eigene Repräsentation zugeordnet sein, das sich öffnet, sobald man das Panel mit der rechten Maustaste anklickt.
Die Klasse `TFormAdapter` lässt sich dann im Adapter Pattern erklären.

```
procedure TSingletonForm.FormCreate(Sender: TObject);
begin
  FSubject:= TMySubject.Create;
  FMyBars:= TMyBarsMaker.CreateBars(pnlContainer);
  FMyTrackBars:= TMyTrackBars.CreateBars(pnlContainer, FSubject);
  FFormAdapter:= TFormAdapter.Create(Self);
end;
```

Der konkrete Konstruktor von `TProgbars` erzeugt dann zur Laufzeit wieder drei Unterprodukte, die auf dem Panel zusammengefasst sind. Diese Erzeugung übernimmt der Builder. Somit gruppiert das Eltern-Panel die drei Kinder ProgressBars, die für verschiedene Ereignisse als Gruppe vorbereitet sind.

> *Für bestimmte Nachkommen von `TGraphic` wird `OnProgress` während länger andauernder Operationen, wie z. B. Laden, Speichern oder Umwandeln von Bilddaten, ausgelöst.*

Mit `OnProgress` wird dem Benutzer eine Rückmeldung über den Fortgang des Prozesses gegeben.

```
function TPanelMaker.CreatePanel(aParent: TWinControl;
                                 aBuild: TBuilder): TPanel
begin
  inherited Create;
begin
  with aBuild do begin
    BuildPanel(aOwner, 100, 200);
    BuildBars(10, 10, 21, 100);
    BuildBars(40, 10, 21, 100);
    BuildBars(70, 10, 21, 100);
  end;
  CreatePanel:= aBuild.Panel;
end;
```

Auffallend ist, dass keine lokalen Variablen in der Funktion sind. Diese werden lokal in der Instanz von `aBuild` des Builders gehalten. Die folgende Schnittstelle dient dem Übergeben des konkreten Builders:

```
type
  TPanelMaker = class (TObject)
  public
```

```
    function CreatePanel( aOwner : TWinControl;
                          aBuild: TBuilder ): TPanel;
  end;
```

Mit dem Etikett „dieselbe Konstruktion erzeugt diverse Repräsentationen“ greift das Pattern zu hoch, da außer einer gemeinsamen Methodenschnittstelle wiederum jede abgeleitete Klasse eine eigene Konstruktion der Elemente aufweisen muss. Konkret kann ich natürlich das Aussehen des Panels parametrisieren, aber die drei Progressbars sind fix auf dem Panel gegeben. Der Konstruktionsprozess sollte als Ganzes innerhalb der jeweiligen Klassen parametrisierbar sein, damit das Versprechen nicht als Etikettenschwindel daherkommt.

Eine flexiblere Parametrisierung bezüglich visueller Repräsentation erlaubt deshalb folgende generische Methode, die ein fast beliebiges visuelles Steuerelement, das von `TControl` abstammt, zu Laufzeit erzeugen kann.

`TControl` ist die abstrakte Basisklasse für alle visuellen Komponenten, die über eine große Anzahl von als `protected` deklarierten Eigenschaften verfügt und Methoden, die von ihren Nachkommen als `published` deklariert werden. Als `parent` wird dann ein Typ von `TWinControl` benötigt, da hier eine Fensterorientierung der Steuerelemente gefragt ist:

```
function TFrmTrans.createControl(aControl:TControlClass;
     const aName:string; aOwner: TControl;
        aParent: TWinControl; x,y,w,h: Integer): TControl;
begin
 result:= aControl.create(aOwner);
 with result do begin
   parent:= aParent;
   name:= aName;
   setBounds(x,y,w,h);
   visible:= true;
 end;
end;
```

Zum Abschluss des zweiten Patterns, ein Sequenzdiagramm, das eine kleinere Einheit eines Builders als Prinzip darstellt. Denn ein Form oder ein Panel sind zwei Formen desselben Konstruktionsprozesses. Abb. 2.10 zeigt einen „Director“ mit `TfrmBank`. Der benötigt ein Panel mit drei Progressbars, somit ruft man den Mini-Builder `TProgBars` auf, der das Panel komplett als Produkt intern erzeugt und als Gesamtes zurückgibt (siehe Abb. 2.10).

Verwendung

Die Funktionalität zum Erzeugen von Forms oder Komponenten in einer IDE ist ähnlich dem Konzept des Builders. Der Klassen-Konstruktor in der CLX manifestiert mit `Application.CreateForm` eine gemeinsame Schnittstelle, die mit den Ressourcen aus der .dfm-Datei die unterschiedlichen Repräsentationen bildet. Auch die abgeleiteten

Klassen profitieren von dieser Konstruktionstechnik, indem das Form die visuellen Klassen bequem verwaltet.

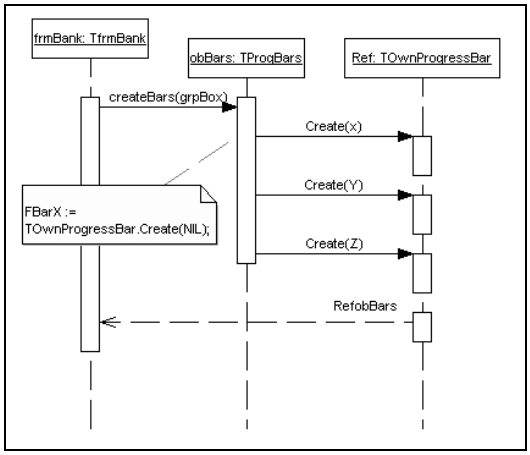


Abb. 2.10: Konstruktion und Repräsentation

Enthält das Formular visuelle Komponenten, muss der Konstruktor deshalb eine externe .xfm-Datei einlesen, um die visuellen Komponenten an ihre Klassen zu binden. Verfügt das neue Formular-Objekt schon über eine externe .xfm-Datei, ruft man nach `CreateNew` die Methode `InitInheritedComponent` auf.

Formulare, die man im Formular-Designer erstellt, sind als Nachkommen von `TForm` implementierbar. Formulare kann man als Hauptfenster einer Anwendung, als Dialogfelder oder untergeordnete MDI-Fenster verwenden. Ein Formular kann natürlich andere Objekte enthalten, z. B. `TButton`, `TCheckBox` und `TComboBox`.

```
procedure TApplication.CreateForm(InstanceClass:TComponentClass;
                                  var Reference);
var
  Instance: TComponent;
begin
  Instance:= TComponent(InstanceClass.NewInstance);
  TComponent(Reference):= Instance;
  try
    Instance.Create(Self);
  except
    TComponent(Reference):= NIL;
    raise;
  end;
  if not FMainFormSet and (Instance is TForm) then begin
    TForm(Instance).HandleNeeded;
    FMainForm:= TForm(Instance);
```

```
    if TForm(Instance).ActiveControl = NIL then
      TForm(Instance).SetFocusedControl(TForm(Instance));
    FMainFormSet:= True;
  end;
end;
```

Eine weitere, letzte Verwendung zeigt das Erzeugen eines kompletten Forms im Ansatz:

```
TAbstractFormBuilder = class
  private
    FForm: TForm;
    procedure BuilderFormClose(Sender: TObject;
                               var Action: TCloseAction);
  protected
    function GetForm: TForm; virtual;
  public
    procedure CreateForm(AOwner: TComponent); virtual;
    procedure CreateSpeedButton; virtual; abstract;
    procedure CreateEdit; virtual; abstract;
    procedure CreateLabel; virtual; abstract;
    property Form: TForm read GetForm;
  end;

TRedFormBuilder = class(TAbstractFormBuilder)
  private
    FNextLeft, FNextTop: Integer;
  public
    procedure CreateForm(AOwner: TComponent); override;
    procedure CreateSpeedButton; override;
    procedure CreateEdit; override;
    procedure CreateLabel; override;
end;
```

Zur Laufzeit befiehlt der Client der Klasse mit der Übergabe des konkreten Builders, den Bau des Forms einzuleiten. Die Rückgabe besteht hier aus dem read only Property `Form` des abstrakten Builders! Anstelle des RedForm sind weitere Formulare vorstellbar.

```
procedure TForm1.Create3ComponentForm(ABuilder:
                                      TAbstractFormBuilder);
var
  NewForm: TForm;
begin
  with ABuilder do begin
    CreateForm(Application);
    CreateEdit;
    CreateSpeedButton;
```

```
    CreateLabel;
    NewForm := Form;
    if NewForm <> nil then NewForm.Show;
  end;
end;
```

2.3.3 Factory

Ein klassenbasiertes Erzeugungsmuster (Objekt wird von Unterklasse erzeugt).

Zweck

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber delegiere die Erzeugung des Objekts an eine Unterklasse, die selbst entscheiden kann, von welcher Klasse das zu erzeugende Objekt stammt.

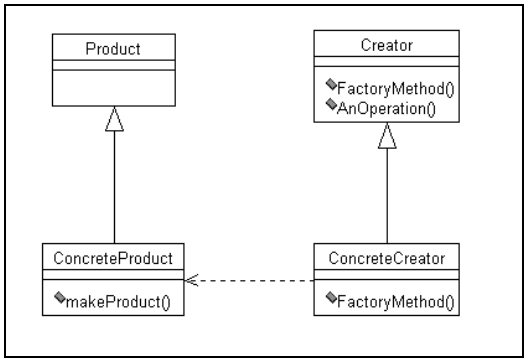


Abb. 2.11: Die Factory Method

Motivation

Eine Abstrakte Klasse, welche ein gemeinsames Interface für Subklassen definiert und daher nicht instanzierbar ist, lässt sich mit Factory-Methoden erweitern. Im Falle des TRadeRoboter weichen wir hier vom ursprünglichen Sinne einer abstrakten Fabrik ein wenig ab, indem sich das Pattern nur Factory oder Factory-Methode nennt und gegenüber der Abstrakten Fabrik eindeutig klassenbasiert ist.

Eine Abstrakte Fabrik lässt sich als eine Kollektion von Factory-Methoden anschauen.

Da das Factory klassenbasiert ist (das sind nur wenige Patterns), erzeugt die Hauptarbeit die Unterklasse. Factory-Methoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren. Die Unterklasse ist in Abb. 2.11 mit dem konkreten Produkt ersichtlich, welches dann die Wahlmöglichkeit zur Instanzierung hat.

Implementierung

In der Unit `uObserver` setzen wir eine Factory-Methode beim Anzeigen der verschiedenen Finanztransaktionen ein. Die Methode kreiert drei Fortschrittsbalken, welche eine Kontrolle über die Berechnung der Chartanalyse bieten. Auch `MakePanel` ist eine Fabrikmethode mit Delegationsprinzip.

Erzeugt werden die Progressbars indirekt in der Methode `CreateBars`, indirekt, weil der Konstruktor von `TProgresBar` an `TMyProgressBar` delegiert wird. Diese Unterklasse entscheidet dann, von welcher Klasse das zu erzeugende Objekt stammen soll. Wie ein Geburtshelfer, der nochmals schnell zum lieben Gott aufschaut.

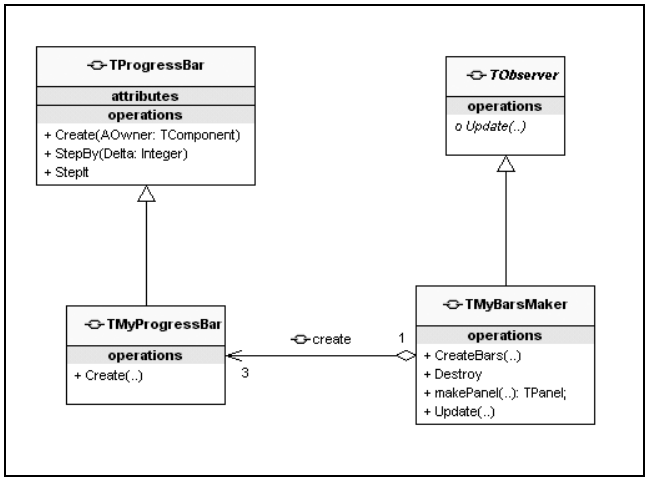


Abb. 2.12: Die Factory-Methode mit Fortschritt

`TMyProgressBar` ist auch für die Darstellung verantwortlich. Die einzelnen Konstruktoren sind im Moment nicht parametrisierbar, d. h., jeder Erbauer baut den gleichen Teil. Die Repräsentation erfolgt dann in der Klasse des jeweiligen Forms.

```
constructor TMyBarsMaker.CreateBars(aParent: TWinControl);
var
  tmpPanel: TPanel;
begin
  inherited Create;
  tmpPanel:=makePanel(aParent);
  tmpPanel.height:= 60;
  tmpPanel.align:= alBottom;
  FBarX:= TMyProgressBar.Create(NIL);
  FBarY:= TMyProgressBar.Create(NIL);
  FBarZ:= TMyProgressBar.Create(NIL);
  FBarZ.Parent:= tmpPanel;
```

```
  FBarX.Parent:= tmpPanel;
  FBarY.Parent:= tmpPanel;
end;
```

C#-Code-Beispiel:

```
    public TMyBarsMaker(Control aParent) {
      tmpPanel = this.makePanel(aParent);
      tmpPanel.Height = 80;
      tmpPanel.Dock = DockStyle.Bottom;
      FBarX = new TMyProgressBar(tmpPanel);
      FBarY = new TMyProgressBar(tmpPanel);
      FBarZ = new TMyProgressBar(tmpPanel);
    }
```

Gebaut werden die Fortschrittsbalken dann in einem gemeinsamen Konstruktor. Mit `Align` kann man ein Steuerelement an der oberen, unteren, linken oder rechten Seite eines Formulars, Panels oder eines Frames ausrichten. Die Komponente bleibt auch dann an dieser Position, wenn sich die Größe des Containers ändert.

Damit ein Steuerelement eine bestimmte Beziehung mit dem Rand seiner übergeordneten Komponente beibehält, ohne notwendigerweise direkt am Rand ausgerichtet zu sein, verwendet man stattdessen die Eigenschaft `Anchors`:

```
constructor TMyProgressBar.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  Min:= 0;
  Max:= 100;
  Step:= 1;
  orientation:= pbVertical;
  Align:= alBottom;
end;
```

C#-Code-Beispiel:

```
  public TMyProgressBar(Control aOwner) {
    FProgressBar = new ProgressBar();
    FProgressBar.Parent = aOwner;
    FProgressBar.Minimum = 0;
    FProgressBar.Maximum = 100;
    FProgressBar.Step = 1;
    FProgressBar.BackColor = System.Drawing.Color.Red;
    FProgressBar.ForeColor = System.Drawing.Color.Red;
    FProgressBar.Dock = DockStyle.Bottom;
  }
```

Wenn ich nun in einem weiteren einfachen Beispiel die Abstrakte Fabrik mit der Fabrik-Methode kombiniere, zeigt sich mit der gemeinsamen Schnittstelle `createCharge`, dass sich auch bei nicht visuellen Klassen das Prinzip sofort durchsetzt.

Der gemeinsamen Methodenschnittstelle wird eine Instanz einer konkreten Fabrik übergeben, die wiederum in der konkreten Fabrik die Fabrik-Methode `setCharge` aufruft. Die Produkte selbst sind die verschiedenen Arten, Gebühren zu berechnen.

In einer künftigen Version ist es vorstellbar, mit weiteren Gebührenarten, nicht nur im Kreditkonto, rechnen zu müssen, sobald sich eine weitere abstrakte Fabrik aufdrängt.

Wenn man noch einen Schritt weiter gehen und Gebühren, Dividenden, Zinsen und dergleichen für einen Kunden auf einmal berechnen möchte, sei der Builder oder der Visitor als Pattern empfohlen. Zudem gehe ich bezüglich des Strategie Patterns noch näher auf den Kontext von Kosten und ihren Berechnungen ein.

```
type
 TChargeMaker = class (TObject)
 public
   function createCharge(Factory: TAbstractFactory ):
                                          TKreditCharge;
 end;
```

Erst die Methode `createCharge` erzeugt aus dem übergebenen Factory das Objekt mit der aktuellen Berechnung, die jeweils mit `setCharge` der konkreten Berechnung als Produkt zugewiesen wird. Das Produkt versteht sich als ein erzeugtes Objekt, das je nach übergebener Formel unterschiedliche Gebühren ermittelt.

```
type
  TConcreteFactory = class (TAbstractFactory)
  protected
    FCharge: TKreditCharge;
  public
    function setCharge(const Formula: TStrings;
                       calcVersion: Integer;): TStrings; virtual;
  end;
```

Verwendung

Transaktionale Datenmodule verwalten Datenbankverbindungen, die man über ADO oder (falls Sie MTS einsetzen und MTS POOLING aktivieren) über die BDE einrichtet, automatisch in einem besonderen Ressourcenpool.

Gibt ein Client seine Datenbankverbindung frei, kann diese von einem anderen Client wieder genutzt werden.

> *Der Datenverkehr im Netzwerk wird reduziert, da die Mittelschicht keine Abmeldung und anschließende Neuanmeldung einer Verbindung beim Datenbankserver durchführen muss.*

Wenn die Verwaltung der Datenbank-Handles genutzt wird, sollte die Datenbankkomponente in der Eigenschaft `KeepConnection` den Wert False enthalten, damit die Anwendung die Freigabe von Verbindungen optimiert. Auch über Parameter lässt sich ein Datenbank-Handle einrichten:

```
stmtParams := TParams.Create;
  Database1.Execute(SQLstmt, stmtParams, False, nil);
```

Die Fabrik-Methode gibt dabei eine Instanz aus dem Pool zurück, welche eine Referenz auf den Produkte-Erzeuger beinhaltet. Der Aufruf von `Close` schließt dann nicht eine Verbindung, sondern kontaktiert den Erzeuger, der die Verbindung zurück in den Pool von einsatzbereiten Instanzen legt. Dieses Vorgehen verhindert das Belegen von Ressourcen wie eben Datenbank-Handles, wenn man diese nicht benötigt.

Alle automatisch generierten Aufrufe eines MTS-Datenmoduls sind transaktionsbezogen (es wird davon ausgegangen, dass das Datenmodul nach dem Ende des Aufrufs deaktivierbar ist und man alle aktuellen Transaktionen definitiv eintragen oder rückgängig machen kann).

2.3.4 Singleton

Ein objektbasiertes Erzeugungsmuster (Klasse existiert mit genau einem Objekt)

Zweck

Stellen Sie sicher, dass eine Klasse mit genau einer Instanz existiert und ermöglichen Sie genau einen Zugriffspunkt auf das Kreieren der einzigen Instanz.

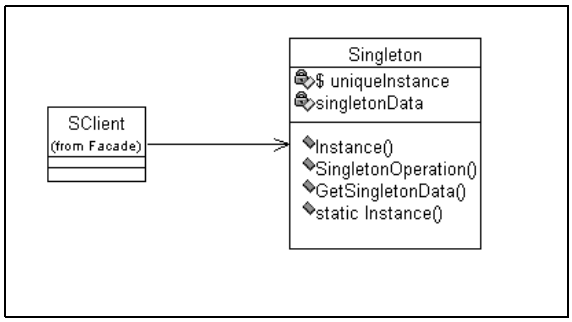


Abb. 2.13: Das Singleton

Motivation

Der wichtigste Aspekt am Singleton ist, dass Sie mit ihm die einzige Instanz der Klasse oder eines Programms steuern können. Dadurch löst man Synchronisationsprobleme und Inkonsistenzen bei mehrfachem Verbindungsaufbau oder Datenzugriff. Sie führen eine

einzelne Instanz, eine Funktion oder einen ganzes Programm wie durch einen zentralen „Kanal“ aus.

Auch die Eindeutigkeit ist von Interesse, sei es, dass ein Name oder eine Instanz zur Laufzeit eindeutig bleibt oder ein Modul als Singleton geladen wird. Die Namen von Packages müssen bspw. innerhalb eines Projekts eindeutig sein, und wenn sogar die Instanz eindeutig sein muss, ist eben das Singleton erste Wahl.

Eine autonome Executable ist auch eine Instanz der Programmklasse im Hauptprogramm. Indem Sie Ihr Programm als Singleton laufen lassen und anhalten, können Sie Mehrfachinstanzen vermeiden, die sich erwiesenermaßen als problematisch verhalten, und sich so auf die lokalen Programmteile konzentrieren, die Probleme verursachen.

Implementierung

Im Falle TRadeRoboter kommt das Singleton beim Erzeugen der Instanz `bankView2` oder der `TSingletonForm` innerhalb des Portfolio zum Einsatz, d. h., ein Form wird als Objekt nur einmal instanziert. Auch bei mehrmaligem Starten wird keine neue Instanz mehr erzeugt.

Ein Singleton-Form hat keinen direkten Zusammenhang mit einem modalen Form. Ein modales Dialogfenster oder Form muss vom Benutzer explizit geschlossen werden, bevor er mit einem anderen Formular arbeiten kann. Die meisten Dialogfenster sind modal. Ein Singleton muss jedoch nicht geschlossen werden, obwohl die Instanz aus der Sicht der Speicherverwaltung wie „modal“ wirkt.

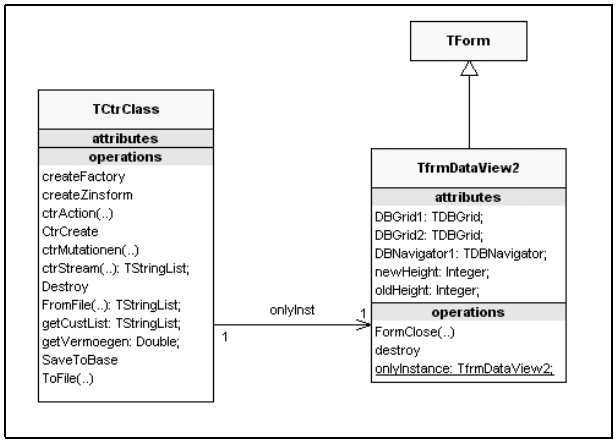


Abb. 2.14: Das Singleton mit einer Instanz

Der Aufruf erfolgt aus dem Controller, der auch zur Laufzeit die datensensitiven Verbindungen herstellt. Somit benötigt die Unit `bankView2` keine fixe Verbindung zum Datenmodul, wie `DataSource` etc., und das Form lässt sich flexibler einsetzen:

```
procedure TCtrClass.ctrMutationen(tableID: byte);
var onlyInst: TFrmDataView2;
```

```
  begin
  case tableID of
   1:begin
     onlyInst:= TFrmDataView2.onlyInstance; //singleton
       with onlyInst do begin
         Show;
         dbGrid1.dataSource:=DataModule1.dSrcCust;
         dbGrid2.dataSource:=DataModule1.dSrcAccount;
         dbGrid2.visible:=true;
         dbGrid2.ReadOnly:=false;
         DBNavigator1.dataSource:=DataModule1.dSrcCust;
       end;
     end;
```

Die einzige Instanz wird nun über eine Klassenmethode erzeugt.

> *Eine Klassenmethode ist eine Methode, die nicht mit Objekten, sondern mit Klassen arbeitet, weil ja die Instanz zum Zeitpunkt des Aufrufes noch gar nicht existieren kann.*

Da es sich bei `Create` und `CreateRemote` auch um Klassenmethoden handelt, benötigen Sie keine Instanz der Creator-Klasse, um diese aufzurufen. Sie rufen also eine Methode nur durch Angabe des Klassentyps, nicht aber der Instanz auf. Dies ist nur bei Klassenmethoden und natürlich bei Konstruktoren, nicht aber bei normalen Methoden und Destruktoren zulässig. Unser Aufruf im TRadeRoboter:

```
onlyInst:= TFrmDataView2.onlyInstance;
```

Mit `Assigned` lässt sich dann prüfen, ober der übergebene Zeiger bzw. die Prozedur NIL ist. `FInstance` muss eine Variablenreferenz eines Zeigers oder prozeduralen Typs sein. `Assigned` kann übrigens keine Zeiger erkennen, die auf keine gültigen Daten mehr verweisen, aber dennoch nicht NIL sind. Solche Nirvana-Zeiger werfen dann meist einen GPF[2], auch Zombie-Zeiger genannt, die man meist durch Umcodieren oder Löschen aus einer Liste verloren hat und somit nicht mehr freigeben kann.

```
class function TfrmDataView2.onlyInstance: TfrmDataView2;
  begin
    if not assigned(FInstance) then
    FInstance:= TfrmdataView2.Create(application);
    with FInstance do
      oldHeight:= height;
    result:=FInstance ;
  end;
```

[2] Who the hell is General Failure and why does he read on my disk?

C#-Code-Beispiel mit einem `static`:

```
  public static TSingletonForm onlyInstance() {
    //lazy initialization: Singleton wird erst beim ersten
    //                                Aufruf instanziert
    if (FInstance == null) {
      FInstance = new TSingletonForm();
    }
    return FInstance;
  }
```

Das Wechseln der Views über die Case-Struktur im Controller garantiert auch das jeweilige Checken mit `assigned`, sodass wirklich nur eine Instanz existiert. Beim Schließen des Forms kommt schlussendlich noch der Destruktor zum Einsatz:

```
destructor TfrmDataView2.destroy;
  begin
    FInstance:= NIL;
    inherited destroy;
  end;
```

Nicht ganz korrekt ist die Deklaration der Instanz selber, die im lokalen Bereich der Unit den einzigen Zugriffspunkt darstellt und als typisierte Konstante daherkommt (typisierte Konstanten können im Gegensatz zu echten Konstanten auch Werte mit Array-, Record- und Zeiger-Typ enthalten):

```
Const
  FInstance: TfrmDataView2 = NIL;
```

C#-Code-Beispiel:

```
  //Speziell beim Singleton: Ein privater Konstruktor
  private TSingletonForm() {
    InitializeComponent();
  }
```

Wenn nun der böse Hugo den Standard-Konstruktor aufruft, ist all unsere Mühe dahin. Es gibt aber einen komplizierten Weg mit Exception Handling, um auch diese Bösartigkeit zu vermeiden (siehe Verwendung). Da Delphi keine `Static` oder private Konstruktoren kennt, das sind lokale Klassenvariablen, die über alle Instanzen der Klasse hinweg gleich bleiben, müsste bei einer besseren Zugriffsmethode noch eine Case-Struktur eingebaut werden, die ungefähr so aussieht:

```
case accessRequest of
  0: ;
  1: if not assigned(FInstance) then Finstance:= createInstance;
  2: Finstance:= NIL;
```

Nun sollte dem ordentlichen Gebrauch des Einzelexemplares nichts mehr im Codeweg stehen, und schizophrene Instanzen gibt es ja noch nicht ;). Übrigens, zum Sicherstellen, dass eine Applikation als Gesamtes nur einmal gestartet wird, genügt es, einen Mutex als Synchronisationsvariable einzusetzen:

```
var hMutex: THandle;
begin
  hMutex:= createMutex(NIL, true, 'troboter');
  if getLastError = ERROR_ALREADY_EXISTS then begin
    showmessage('TRadeRoboter already in use');
    HALT;
  end
end;
```

Verwendung

Standardmäßig wird in Delphi das Hauptformular einer Anwendung im Speicher erzeugt, indem der folgende Quelltext durch den Linker in den Haupteinsprungspunkt der Anwendung eingefügt wird und somit `TApplication` als Singleton implementiert ist:

```
Application.CreateForm(TForm1, Form1);
```

Diese Anweisung erstellt eine globale Variable mit demselben Namen wie das Formular. Für jedes Formular einer Anwendung ist also eine entsprechende globale Variable vorhanden. Diese Variable, die ein Zeiger auf eine Instanz der Formularklasse ist, wird zur Laufzeit für den Zugriff auf das Formular verwendet.

Jede Unit, deren uses-Klausel eine Referenz auf das Formular enthält, kann über diese Variable auf das Formular zugreifen. Formulare, die auf diese Weise in der Projekt-Unit angelegt wurden, werden nach dem Start der Anwendung angezeigt und bleiben während der gesamten Laufzeit der Anwendung im Speicher. Auch `TScreen` oder `TClipboard` in der CLX kommen als Singleton daher.

Interessant wird es, wenn ein Datenmodul als Singleton zu gebrauchen ist, man aber die Klasse nicht mit einer Klassenmethode erweitern will oder kann. Hier hilft im wahrsten Sinne des Wortes eine Hilfsfunktion, die man global oder lokal deklarieren kann und den eigentlichen Konstruktor somit kapselt:

```
const
  _myDataModule: TMyDataModule = NIL;

function singleKeeper: TMyDataModule;
begin
  if not assigned(_myDataModule) then
    myDataModule:= TMyDataModule.Create(Application);
  result:= _myDataModule;
end;
```

Dass jemand den öffentlichen Konstruktor aufruft, sei hier noch nicht abgefangen, dies ist wie erwähnt mit einer komplizierten Ausnahmebehandlung möglich:

```
type
  ESingleton = class(Exception);
  TTimeKeeper = class;
  TTimeKeeperClass = class of TTimeKeeper;
  TTimeKeeper = class(TInvalidateDestroy)

  TInvalidateDestroy = class(TObject)
  protected
    class procedure SingletonError;
  public
    destructor Destroy; override;
  end;

  class procedure TInvalidateDestroy.SingletonError;
  begin
    raise ESingleton.CreateFmt('Illegal use of %s singleton
                                  instance!', [ClassName]);
  end;
```

Eine weitere Verwendung ist das Pooling oder „Singeling“ von Remote-Datenmodulen. Um in Delphi oder CLX das Pooling von Remote-Datenmodulen zu ermöglichen, rufen Sie `RegisterPooled` in der Methode `UpdateRegistry` einer Remote-Datenmodulklasse aus.

Wenn für Datenmodulinstanzen das Pooling verwendet wird, unterhält der Server einen Zwischenspeicher mit den Instanzen. Jede Client-Anforderung wird von der ersten verfügbaren Instanz aus diesem Zwischenspeicher bedient. Da die Instanzen gemeinsam genutzt werden, stehen dem Datenmodul keine zuverlässigen persistenten Statusinformationen zur Verfügung.

Singleton gibt nun an, dass man alle Client-Aufrufe an dieselbe Datenmodulinstanz leitet. Es gibt nur eine einzelne Instanz anstelle eines Pools verfügbarer Instanzen. Singleton sollte nur auf true gesetzt werden, wenn das Remote-Datenmodul das freie Threading verwendet. Diese Singleton-Instanz ist dann ohne Zeitbegrenzung vorhanden.

```
procedure RegisterPooled(const ClassID: string; Max, Timeout:
                         Integer; Singleton: Boolean = False);
begin
  CreateRegKey(SClsid + ClassID, SPooled, SFlagOn);
  CreateRegKey(SClsid + ClassID, SMaxObjects, IntToStr(Max));
  CreateRegKey(SClsid + ClassID, STimeout, IntToStr(Timeout));
  if Singleton then
    CreateRegKey(SClsid + ClassID, SSingleton, SFlagOn);
end;
```

2.4 Strukturmuster

2.4.1 Adapter

Ein objektbasiertes Strukturmuster (Klasse erzeugt Schnittstellenanpassung)

Zweck

Passe die Schnittstelle einer Klasse an eine andere erwartete Schnittstelle vom Typ her an. Ein Adapter lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen sonst nicht fähig dazu wären.

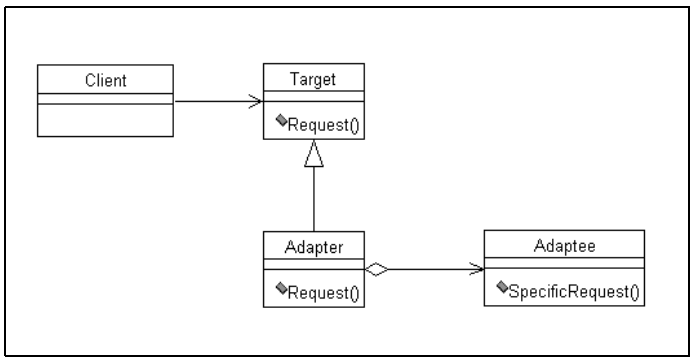


Abb. 2.15: Der Adapter in Reinkultur

Motivation

Die klassenbasierte Variante geht nur mit Mehrfachvererbung, somit ist das Muster auch objektbasiert!

Grundsätzlich ist das Problem eine bestehende Klasse, die schon von einer anderen Klasse erbt, die aber im Zuge eines Redesigns von einer weiteren Klasse erben sollte. Da in den meisten Sprachen keine Mehrfachvererbung möglich ist, bietet das Adapter-Muster einen Ausweg. Außer bei Schnittstellen sollte die Mehrfachvererbung auch vermieden werden, da mit der Zeit eine Entkopplung oder ein Refactoring kaum mehr machbar ist.

Zweite Motivation ist das Weiterleiten einer Anfrage an eine Methode, die sich nicht innerhalb der Vererbungshierarchie befindet, aber trotzdem Nachrichten oder Anfragen empfangen möchte. Wie eine Steckernorm, die mit einem Adapter einen neuen Stecker kompatibel, sprich aufnahmefähig, macht.

Viele Klassenbibliotheken kennen den Adapter, wie z. B. `TCustomAdapter`, die als eine Basisklasse für Objekte, die VCL-Objekte mit OLE-Schnittstellen oder ADO verbinden, wirkt. Die Prozedur `GetOleFont` z. B. erstellt ein Adapter-Objekt, das die Eigenschaften eines nativen TFont-Objekts von Delphi auf ein OLE-Schriftobjekt abbildet.

Implementierung

Im Falle TRadeRoboter kommt der Adapter dann zum Einsatz, wenn ein Form nicht von `TObserver` abstammen kann (weil es schon von `TForm` abstammt), aber trotzdem kompatibel sein muss, da es auf die Methoden der Schnittstelle von `TObserver` angewiesen ist. Das heißt, das Form muss kompatibel zu einer zweiten Klasse sein, in meinem Falle eben zu `TObserver`.

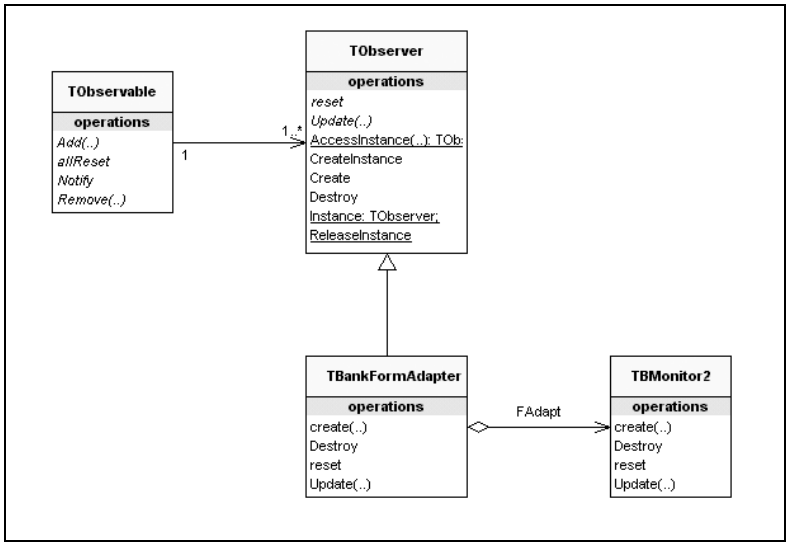


Abb. 2.16: Der Adapter vermeidet inkompatible Schnittstellen

Problem, Kontext und Lösung zeigen jetzt folgende Punkte auf:

- In den meisten Fällen sind Typen nicht kompatibel zueinander.
- Das Form klinkt sich in einen Adapter ein, d. h., es macht seinen Typ dem Adapter über den Konstruktor bekannt.
- Da der Adapter schon von `TObserver` abstammt, stammt nun indirekt auch das neue Form von `TObserver` ab.

Der Begriff Schnittstelle ist hier wohl etwas zu hoch gegriffen, in den meisten Fällen sind eigentlich die Typen bezüglich des Aufrufs nicht kompatibel zueinander und auch nicht die Daten mit ihren Formaten (Dataface versus Interface). Nun der Reihe nach bis zum Adapter. Im TRadeRoboter stammt im Normalfall ein visuelles Control auf dem Hauptfenster von `TObserver` ab:

```
TBMonitor = class(TObserver)
private
  frmMon: TForm;
  memMon: TMemo;
public
```

```
  constructor create(aOwner: TComponent);
  destructor Destroy; override;
  procedure Update(dispCenter: TObservable); override;
  procedure reset; override;
end;
```

Was nun, wenn ein neues Form hinzukommt, das schon von `TForm` abstammt und keine Mehrfachvererbung zulässt oder Probleme mit der Methodenvererbung hat. Wäre es nicht einfach, das Form klinkt sich in einen zusätzlichen Adapter ein, d. h., es macht seine Referenz dem Adapter über den Konstruktor bekannt.

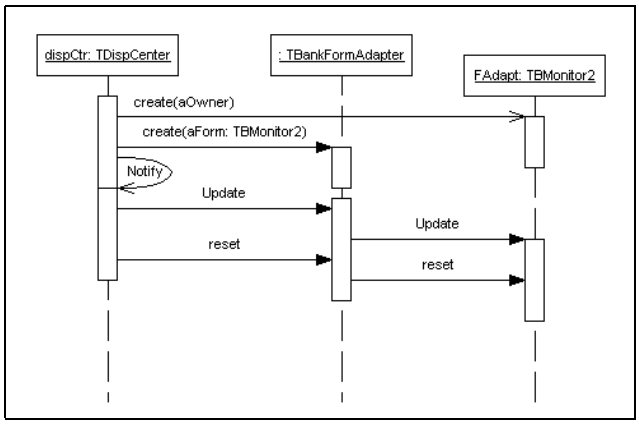


Abb. 2.17: Der Adapter delegiert an seinen Member

Dann rufen Clients auch die Operationen von Adapter-Objekten auf, schließlich ruft der Adapter über seinen Member die Operation der umwickelten oder adaptierten Referenz, also den neuen Form `TBMonitor2`, auf.

Da der Adapter schon von `TObserver` abstammt, stammt nun indirekt auch das neue Form `TBMonitor2` von `TObserver` ab:

```
TBankFormAdapter = class(TObserver)
  private
    FAdapt: TBMonitor2;
  public
    constructor create(aForm: TBMonitor2);
    procedure Update(dispCenter: TObservable); override;
  end;
```

Die eigentliche Kopplung des Objektes an den Adapter geschieht im Konstruktor, indem er die übergebene Referenz dem privaten Feld `FAdapt` zuweist:

```
constructor TBankFormAdapter.create(aForm: TBMonitor2);
begin
```

```
  inherited Create;
  FAdapt:= aForm;
end;

procedure TBankFormAdapter.update(dispCenter: TObservable);
begin
  FAdapt.Update(dispCenter);
end;
```

C#-Code-Beispiel mit Namensraum:

```
  public class TFormAdapter : TObserver {
    private TSingletonForm FAdaptee;
    public TFormAdapter(TSingletonForm aForm) {
      FAdaptee = aForm;
    }
    public override void Update(TObservable ChangedSubject) {
      FAdaptee.UpdateView();
    }
  }
```

Da es in C# keine Trennung von Interface und Implementierung in der Datei gibt, ist die Deklaration und Definition in der Klasse selbst zu finden, wie obiger Vergleich explizit zeigt. Es macht auch wenig Sinn, Klassenstrukturen in C# zu zeigen, da jeweils die ganze Implementierung mitkommt und somit die Struktur überlastet. In C# sieht der Unitkopf wie folgt aus, Namespace steht schlicht für den Unitnamen:

```
using ch.kleiner.patternskonkret.uObserver;
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
namespace ch.kleiner.patternskonkret.uSingletonF {
```

Verwendung

Als Verwendung soll wieder mal ein Webserver Framework herhalten, die Rede ist von WebSnap. Adapter definieren dort eine Script-Schnittstelle zu Ihrer Server-Anwendung. Sie erlauben es, Scripts in eine Seite einzufügen und aus dem Scriptcode heraus durch Aufrufe des zugewiesenen Adapters Informationen abzurufen.

Z. B. kann man durch einen Adapter Datenfelder definieren, die sich in einer HTML-Seite anzeigen lassen. Eine HTML-Seite im Scriptformat kann HTML-Inhalt enthalten, ebenso wie Scriptanweisungen, welche die Werte dieser Datenfelder abfragen.

Dies entspricht in etwa den transparenten Tags, die in Web-Broker-Anwendungen wie PHP, ASP oder eben WebSnap[3] zum Einsatz kommen.

Adapter unterstützen des Weiteren Aktionen, die Befehle ausführen. Beispielsweise kann man durch Klicken auf einen Hyperlink oder Absenden eines HTML-Formulars Adapteraktionen auslösen.

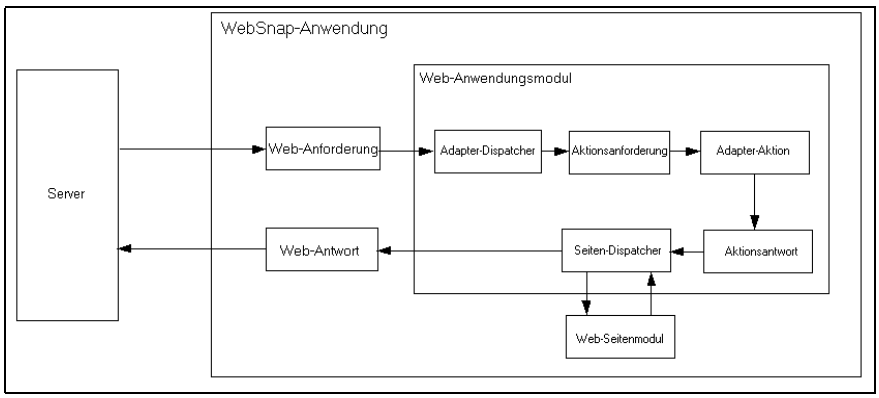


Abb. 2.18: Im Webbroker sind Adapter im Einsatz

Wenn der Adapter-Dispatcher eine Client-Anforderung erhält, erstellt er die Adapter-Anforderungs- und Antwort-Objekte, in denen Informationen zu dieser Anforderung gespeichert werden (Request und Response). Aktionsanforderungen dienen dazu, die HTTP-Anforderung in die Informationsbestandteile zu zerlegen. Die Adapter-Objekte sind im Web-Kontext gespeichert, um den Zugriff während der Verarbeitung der Client-Anforderung zu ermöglichen.

```
function TBaseAdapterAction.GetAdapter: TCustomAdapter;
begin
  if FAdapter = nil then
    FAdapter:= FindAdapterInParent(Self);
  Result:= FAdapter as TCustomAdapter;
end;
```

Adapter vereinfachen das dynamische Erstellen von HTML-Seiten. Durch die Verwendung von Adaptern in einer Anwendung ist die Möglichkeit gegeben, ein objektorientiertes Script aufzunehmen, das bedingte Logik oder Loops unterstützt. Ohne Adapter und serverseitiges Scripting muss sonst der größere Teil der HTML-Generierungslogik in Ereignisbehandlungsroutinen zum Einsatz kommen. Der Einsatz von Adaptern kann die Entwicklungszeit ziemlich verkürzen.

[3] IntraWeb ist stark im Kommen.

Eine weitere eher optische, aber echte Verwendung ist das Wandeln eines Farbmodells in ein anderes Farbmodell. Es ist deutlich zu sehen, dass im Konstruktor das Feld `FRGB8bit` von der Schnittstelle `IRGB8bit` nun kompatibel zu `TRGBColorRefAdapter` gemacht wird.
Ein `IRGB8bit`-Objekt wird sozusagen einem `IColorRef`-Objekt zugeordnet:

```
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit,
                                          IColorRef)
private
  FRGB8bit: IRGB8bit;
  FPalRelative: Boolean;
public
  constructor Create(rgb: IRGB8bit);
  property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
  property PalRelative: Boolean read FPalRelative
                                    write FPalRelative;
  function Color: Integer;
end;

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit:= rgb;
end;
```

Die Direktive `implements` ermöglicht es, die Implementation einer Schnittstelle an eine Eigenschaft der implementierenden Klasse zu delegieren, d. h. eine andere Klasse mit der Implementierung zu beauftragen (siehe Teil 1).

Die Klasse, die zur Implementierung der delegierten Schnittstelle verwendet wird, muss von `TInterfacedObject` abgeleitet sein. Standardmäßig führt die Verwendung des Schlüsselwortes `implements` zur Delegation aller Schnittstellenmethoden.

2.4.2 Bridge

Ein objektbasiertes Strukturmuster (Trennung Abstraktion von Konkreter Klasse)

Zweck

> *Entkopple eine Abstraktion von ihrer Implementierung, sodass beide Klassen unabhängig voneinander variieren können.*

Motivation

In der Regel ist eine Oberklasse als Abstraktion (Abstrakte Klasse oder Oberklasse mit Grundverhalten) mit den zugehörigen Unterklassen als Vererbung realisiert. Diese dauerhafte Bindung zur Design- wie zur Laufzeit möchte man bei Bedarf aufbrechen, um mehr

Flexibilität zu erlangen. Eine Bridge arbeitet mittels Delegation (Objektkomposition), sodass ein Objekt auch zur Laufzeit auswechselbar bleibt.

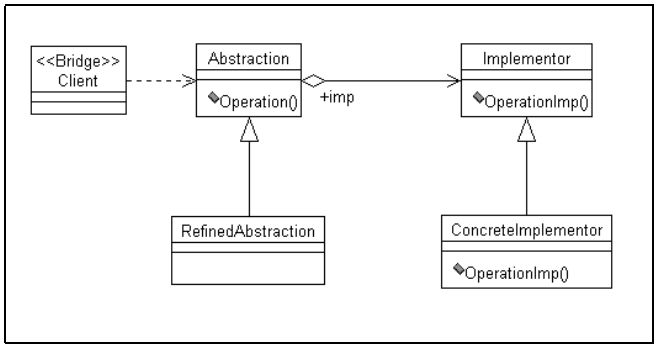


Abb. 2.19: Mit der Bridge delegieren

Durch die Delegation kann die Abstraktion die Operation einfach an diese ausgewechselte Implementierung weiterleiten. Der Delegat muss lediglich die Implementierung der Methoden zur Verfügung stellen. Die Schnittstelle muss hier nicht deklariert werden. In Teil 1 unter Delegation bin ich detaillierter auf die grundsätzliche Designfrage Delegation versus Vererbung eingegangen.

> *Eine durchgängige Delegation liegt dann vor, wenn das aufrufende Objekt auch eine Referenz auf sich selbst dem Implementierer mitliefert.*

Verwandte Muster sind die Fabrik und der Adapter. Vom Konzept her ähnlich der Abstrakten Fabrik, wobei die Fabrik ganze Familien von konkreten Klassen oder Teilen erzeugt. Ein abstrakte Fabrik kann eine bestimmte Brücke erzeugen und konfigurieren. Eine Bridge ist von vornherein als Designgrundsatz im Einsatz, ein Adapter wird meist nach der Entwicklung eines Systems eingesetzt.

Implementierung

Im Falle TRadeRoboter bezweckt die Bridge das Erzeugen von SMS-Nachrichten als Bestätigung der erfolgten Aufträge (Trades). Genauer gesagt erfolgt eine Entkopplung zwischen dem Abstrakten Controller, der den Port als Member beinhaltet, und dem konkreten Port, welcher das SMS sendet. Der SMSPort wiederum lässt sich mit einer MMS-Funktionalität erweitern, welche die Charts auch grafisch aufs Handy bringen.

Für Clients sind die inneren Objekte der SMS-Generierung nicht sichtbar. Obgleich der Zugriff auf die Schnittstellen des inneren Objekts über den Controller als äußeres Objekt erfolgt, ist deren Implementierung völlig transparent.

Das heißt auch, der SMSPort entscheidet, ob nur Text oder bei Bildinformation auch ein MMS-Typ gesendet wird. Dazu wird auch ein `FilePath` auf einem Webserver aktiviert, wenn der Empfänger kein MMS eingerichtet hat.

Somit kann die äußere Objektklasse den Typ der inneren Objektklasse mit jeder Klasse austauschen, die einen identischen SMSController implementiert. Entsprechend lässt sich

der Code für die inneren Objektklassen (SMS- und MMSPort) von anderen Klassen auch wieder verwenden.

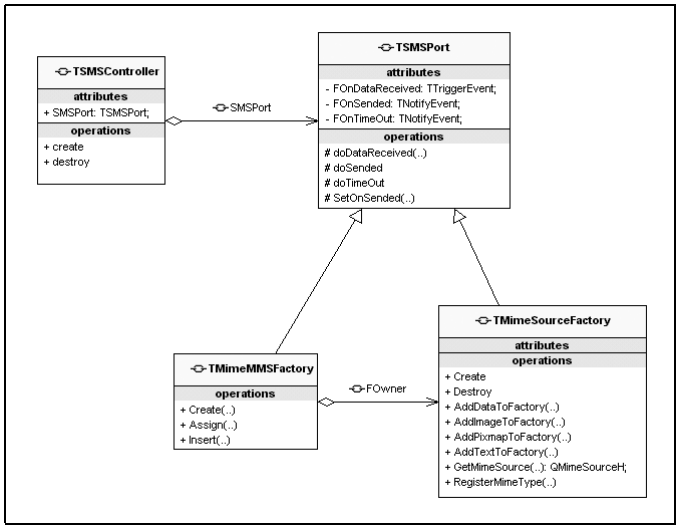


Abb. 2.20: Die Bridge als MMS-Generator

Erzeugt wird der SMSController über eine eigene Objekthierarchie, die auch Unterklassen haben kann. Vorstellbar ist das Auswechseln des SMSPort zur Laufzeit, indem eine Nachricht auch über FTP, Fax oder Pager gesendet wird.

Eigentlich ist die Abstraktion `TSMSController` ziemlich generisch, einzige Bedingung der generischen Operation ist, dass der Text nicht mehr als 156 Zeichen enthalten darf. Wenn der Text länger ist, müsste eine spezialisierte Abstraktion herhalten oder der Implementierer wird ausgewechselt.

Die Unit *uLooktrans.pas* hält die Deklaration der Typen:

```
Type
  TLiterale = array[0..50] of string;
  TSMS_Box = class;
  TMessageClass = class;
  TShortStr = string[55];
  TSMSText = string[156];
```

Verwendung

Die Funktionalität, wie ein Thread als Abstraktion eine Entkopplung zur konkreten Klasse aufbaut, dient als eindrückliche Verwendung des Bridge. Die Klasse `TCustomIpClient` kapselt Client-Netzwerkfunktionen. Sie verbindet Sockets mit Netzkwerkprotokollen wie TCP/IP und UDP sowie mit „Raw Sockets“ und benutzerdefinierten Protokollen. Typischerweise vermeidet man, Instanzen von `TCustomIpClient` zu erzeugen.

Der Typ dient als Basisklasse für nicht visuelle Komponenten, die man in der Komponentenpalette installiert und im Formular-Designer verwendet.

Die Eigenschaften und Methoden von `TCustomIpClient` stellen abgeleiteten Klassen verschiedene Grundfunktionen zur Verfügung und lassen sich überschreiben, um das Verhalten an die jeweilige Anwendung anzupassen.

Ein Thread kommt dann zum Einsatz, wenn der Blockmode auf blockiert gesetzt ist (bedeutet synchron) und ein separater Thread damit die Überwachung im Hintergrund, z. B. als „Listener", übernehmen muss, ohne dass die gesamte Serveranwendung blockiert ist.

Das Socket befindet sich dann auch im Blocking-Modus. Das bedeutet, dass das Socket das Lesen oder Schreiben über die Verbindung initiieren muss und dass das Lesen oder Schreiben synchron auftritt (die Ausführung wird erst fortgesetzt, wenn Lesen oder Schreiben vollständig erfolgt ist).

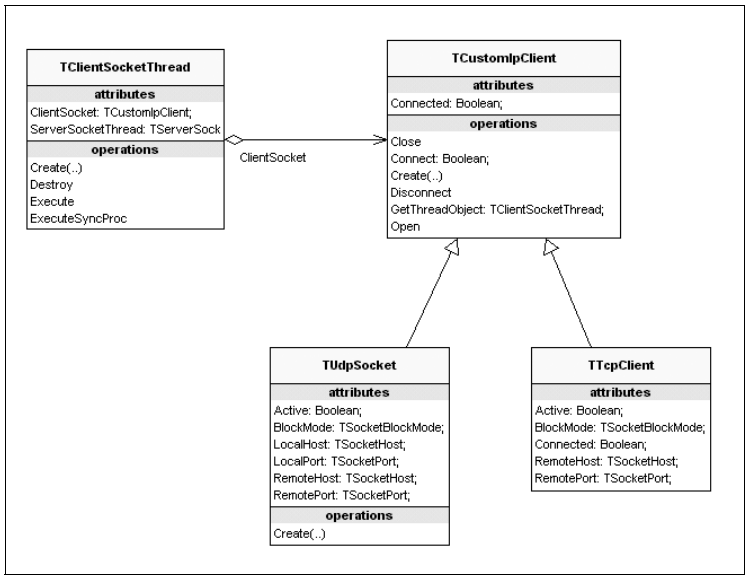


Abb. 2.21: Ein Thread delegiert an den Implementierer

Jede neue Instanz von `TClientSocketThread` (ein Nachkomme von `TThread`) repräsentiert einen neuen Ausführungs-Thread. Die Verwendung eigener `TClientSocketThread`-Objekte erlaubt es den Client-Sockets, ihre Ausführung zu unterbrechen, ohne die gesamte Anwendung zu blockieren.

```
procedure TClientSocketThread.Execute; override;
begin
ThreadObject:= Self;
while not Terminated do begin
  if Assigned(FServerSocketThread) and not
  FServerSocketThread.Terminated and
```

```
  Assigned(FServerSocketThread.ServerSocket) then begin
    FClientSocket:= TCustomIpClient.Create(nil);
    try
      FServerSocketThread.ServerSocket.Accept(FClientSocket);
    finally
      FClientSocket.Free;
      FClientSocket:= nil;
    end;
  end;
  if not Terminated then
    Suspend;
end;
```

Zu sehen ist die Delegation mit dem Member `ClientSocket`, der die Brücke zum Implementierer `TCustomIPClient` konkretisiert. In `Execute` stellt man die Implementierung bereit, die sich bei der Aktivierung des Threads ausführen lässt. `Execute` prüft den Wert der Eigenschaft `Terminated` und ermittelt dadurch, ob der Thread beendet werden muss.

Kompliziert wird es dann, wenn andere Objekte während des Threads ins Spiel kommen. Verwenden Sie deshalb Eigenschaften und Methoden anderer Objekte nicht direkt in der Methode `Execute` eines Threads.

Diese Operationen sollte man in einem separaten Prozeduraufruf durchführen; mithilfe der Methode `Synchronize` lassen sich diese zusätzlichen Prozeduren anderer Objekte als Parameter übergeben.

> *Synchronize löst den Aufruf einer bestimmten Methode aus, die vom Haupt-Thread ausgeführt werden soll. Durch dieses indirekte Verfahren werden Konflikte in Multithread-Anwendungen vermieden.*

Unsichere Methoden können Sie auch durch kritische Sektionen oder mithilfe einer Synchronisation, die mehrfaches Lesen, aber nur exklusives Schreiben zulässt, schützen.

2.4.3 Composite

Ein objektbasiertes Strukturmuster (Struktur und Zusammensetzung von Objekten)

Zweck

> *Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Composite ermöglicht es Clients, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.*

Motivation

Das Composite ist ein zentrales Muster, da rund acht weitere Muster vom Composite gebraucht werden oder in Abhängigkeit zu ihm stehen. So steht z. B. der Interpreter für

das Definieren seiner Grammatik als rekursive Struktur in Abhängigkeit zum Composite, das Composite wiederum setzt für das Traversieren der rekursiven Struktur den Iterator ein oder benötigt bei gesamthaften Operationen den Visitor.[4]

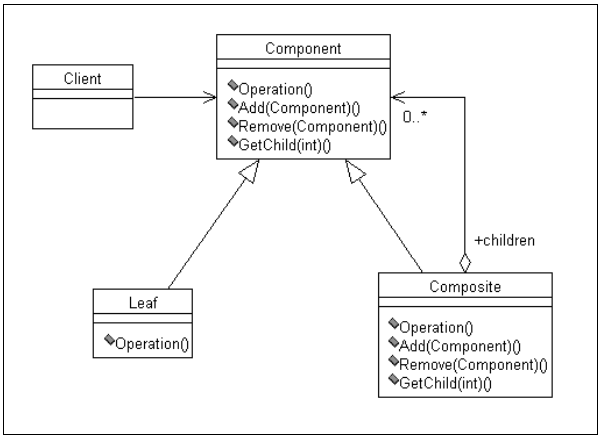


Abb. 2.22: Das Composite als Ganzes mit Teilen

Stichwort Rekursion:

> *Das Muster hat seinen größten Nutzen, wenn eine rekursive Objektstruktur vorliegt und der Container wie auch das Traversieren von dieser kompakten, aber nicht unbedingt verständlichen Technik Gebrauch machen können.*

Worin liegt nun der Nutzen des Composite: Wenn ohne Composite eine Objektstruktur aus Teilen besteht, welche wiederum ein Ganzes ergeben, besteht die Schwäche, dass jedes Objekt Methoden enthält, die dann nicht auf das Ganze anwendbar sind.

Lassen Sie mich ein Beispiel geben: In einer Suchmaschine ist es möglich, nach Dokumenten zu suchen, Bookmarks zu setzen und auszudrucken, die Möglichkeit aber, eine gefundene Gruppe von Dokumenten auszudrucken, fehlt in den meisten Websites.

Die Grundidee des Composite ist, in einer abstrakten Klasse Funktionen zu definieren, welche sowohl elementare Objekte als auch ihre Container beeinflussen und repräsentieren. Gemeinsame Operationen aller Objekte sind in der abstrakten Klasse als Typ enthalten, die man Komponente (Kindelement) nennt.

> *Dieses abstrakte Kindelement (Component) hat aber keine Daten oder Objekte, wie man vielfach meint, die Daten und Objekte sind nur im Composite oder in den Blättern (Leafs) zu finden.*

In Abb. 2.23 ist das Kindelement die Enterprise als Typ des Unternehmens, sodass n-Elemente einen Business-Process ergeben, diese wiederum fasst man in eine Gruppe zusammen. Die Hierarchie ist im Composite also vom Einzelnen zum Ganzen zu sehen,

[4] James Noble: Basic Relationship Patterns, Program Design #4 (PLOPD4).

das Ganze entspricht dann der Komposition aus Einzelteilen (Blätter vom Typ der Kindobjekte), die sich am Schluss der Hierarchie befinden.

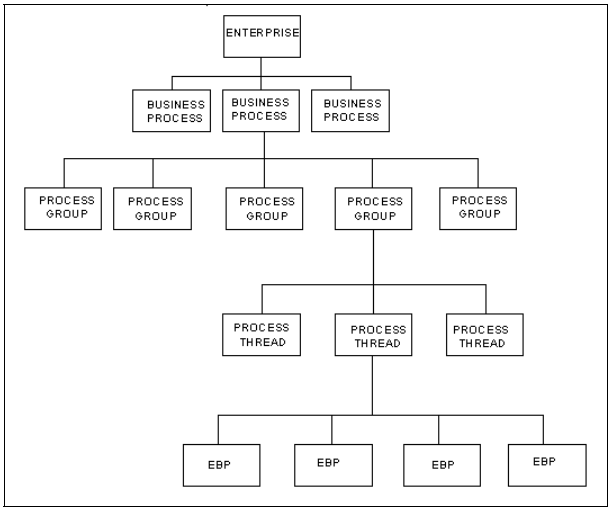


Abb. 2.23: Composite mit Enterprise-Elementen

Diese Kindobjekte hält das Muster, Kompositium genannt, meist in einer Liste zusammen. Daher die modellierte Aggregation von Eltern (Composite) zu Kind (Component) wie in Abb. 2.22 zu sehen.

> *Der Begriff Component entspricht hier aber nicht dem Begriff und Kontext der Software-Komponenten-Lehre.*

Auch nicht zu verwechseln mit dem UML-Begriff Composition, der eine enge Klassenbeziehung durch Delegation meint (auch Objektkomposition).

Komponenten in der Sprache des Musters sind primitive (einfache) Objekte wie:

- ein Element als Symbol oder Teil
- ein Record als Datenbestandteil
- eine Grafik als Muster oder Bildelement

die das Muster zur Baugruppe, zur Tabelle oder zum Seitenbild aggregiert und mit den einzelnen Blättern im Container verwaltet.

Man legt also zuerst den Container fest, der rekursiv die Elemente traversiert. Um eine Komponente in einen Container einzufügen, muss man im Composite zuerst den Container markieren, bevor man die gesuchten Elemente erhält oder neue hinzufügt. Sie alle kennen diese rekursiv geschachtelten Bäumchen mit Zweigen und Blättern, wovon die Zweige ständig neue Blätter erhalten können; dieses Bild prägt das Muster.

> *Ein Kompositium als Klasse speichert Kindobjektkomponenten, die gemeinsame Operationen durchgängig ermöglichen.*

Verwandte Muster sind der Dekorierer, der das Composite mit Zusatzoperationen unterstützt. Ein Frame (von `TFrame`) hat als Composite solche Zusatzoperationen und kann z. B. wie ein Formular als Container für andere Komponenten verwendbar sein.

Beide verwenden denselben Eigentümer-Mechanismus für das automatische Erstellen und Freigeben der eingebetteten Komponenten sowie dieselben Beziehungen zwischen übergeordneten und untergeordneten Objekten für das Synchronisieren der Komponenteneigenschaften.

Implementierung

Im Falle TRadeRoboter bezweckt das Composite das Verwalten und Traversieren einer rekursiven Tabellenstruktur, welche die Portfoliokunden repräsentiert. Das Einlesen der Records in die Objekte und die darauffolgende Darstellung der Teile wie dem Ganzen als Baum entspricht dem Composite. Als Beispiel sei die Tabelle `department` aus der InterBase-Demodatenbank empfohlen. Die Tabelle `department` ist rekursiv nach `dept_no` aufgebaut und nach dem Primärschlüssel sortiert.

Dept_No	Department	HeadDept	Location	Budget
000	Corporate Headquarters		Monterey	1350000
100	Sales and Marketing	000	Frisco	70000
120	European Headquarters	100	Bern	900000
140	Pacific Headquarters	100	Kuaui	1690000
150	Andromeda Headquarter	140	Delphi	2345670

Tab. 2.1: Die rekursive Tabelle

Die einzelnen Element sind als einfache Objekte mit den jeweiligen Records verbunden, die sich dann in der Containerklasse in einer Liste zusammenführen lassen. Für Clients sind die inneren Objekte als einzelne Records nicht sichtbar. Nur die aggregierte Liste im Container der Klasse `TDims` ist greifbar und lässt sich mit der zugehörigen Operation eben rekursiv traversieren.

Zuerst erzeugt das Muster die Abfrage als Ergebnismenge, die dann mithilfe einer Stringliste über den Konstruktor die einzelnen Records in den Container `TDims` abfüllt. Dann erfolgt mit `FillTree` die grafische Darstellung des Baumes. Die gemeinsame Operation `FillTree` gehört zu jedem einzelnen Objekt, zudem traversiert die Operation die einzelnen Listen mit den zugehörigen Kindern:

```
procedure TForm1.btnTreeSelfClick(Sender: TObject);
var
  dimlist: TStringList;
begin
  dimlist:= TStringlist.create;
```

```
  datdepartment:= TBusinessObj.Create(NIL);
  try
    if datDepartment.open_recursiveQuery then
    with TDims.create(NIL, dimList) do begin
      FillTree(treeview1, NIL);
      Free;
    end;
    treeview1.FullExpand;
    btnLTree.visible:=true;
  finally;
    dimlist.Free;
    datDepartment.Free;
  end;
end
```

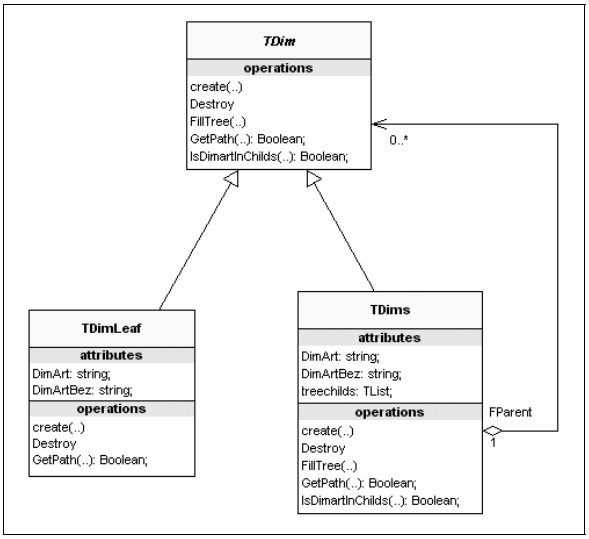


Abb. 2.24: Das Composite sucht seinen Weg

Die eigentliche Rekursion erfolgt in der Containerklasse selbst, sodass jedes Kindobjekt mit der Operation `Add()` eine weitere Instanz mit einer zugehörigen `TList` aufbaut, die wiederum neue Kinder enthält oder enthalten kann.

In der `TList` lassen sich also die Kinder aufnehmen. Wenn weitere Kinder als Elemente vorhanden sind, wird das Kind automatisch zu Eltern befördert, die wiederum Kinder aufnehmen können. Ansonsten bleiben sie Kinder als einfache Objekte, in der Musterlehre auch Blätter genannt.

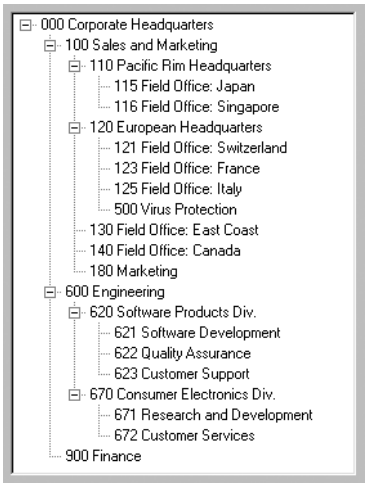


Abb. 2.25: Die rekursive Tabelle als Composite dargestellt

Mit `AddObject` fügt man zusätzlich der Stringliste einen String hinzu und ordnet dem String ein Objekt zu.

`AddObject` gibt den Index des neuen String und des neuen Objekts zurück.

Diese globale Liste (über alle Objekte hinweg) dient dann zum Darstellen eines Records und der zugehörigen Objektidentität durch `self`, die beim Auslesen oder Navigieren nützlich ist. Für das Prinzip des Composite ist `myRegister` in Form einer Liste nicht unbedingt erforderlich. Diese Liste dient zu Kontrollzwecken.

Das Beispiel zeigt außerdem keine Oberklasse, sodass ich direkt mit einem rekursiven Konstruktor arbeite und die erzeugte Instanz der Liste hinzufügen kann:

```
constructor TDims.create(Sender:TDims; myRegister: TStringList);
var
  bmAkt: TBookmark ;
begin
  inherited Create ;
  with datDepartment.qrySelfTree do begin
    FstrDimArt:= fieldByName('DEPT_NO').AsString;
    FstrDimArtBez:= fieldbyName('DEPARTMENT').AsString;
    myRegister.AddObject(Format('%10s',[FstrDimArt]), self) ;
    FChilds:= TList.Create ;
    FParent:= Sender;
    bmAkt:= GetBookmark ;
    if Locate('DEPT_NO', FstrDimArt,[]) then
    while Not (EOF) Do Begin
      if (fieldByName('HEAD_DEPT').Asstring = FstrDimArt) then
        FChilds.Add(TDims.create(self, myRegister));
```

```
      Next ;
    end ;
    GotoBookmark (bmAkt) ;
    FreeBookmark (bmAkt) ;
  end;
end;
```

Es entsteht im Weiteren eine gegenseitige Verknüpfung, da die Routine im Member `FParent` jeweils den Vorgänger als Elternteil durch `Self` speichert. `Self` wird einfach dem Parameter `Sender` übergeben. Die letzten Objekte einer Hierarchie haben dann jeweils keine Kinder mehr, sodass man hier eben von Blättern spricht.

Da einfache Objekte keine Kindobjekte haben, implementiert auch keine dieser Blattklassen kindbezogene Operationen, d. h., das Objekt benötigt keine Liste.

```
Procedure TDims.FillTree(aOl: TTreeview; xnode: TTreenode);
var
 i: integer ;
 dbcontent: string[255];
begin
  dbcontent:= dimart +' '+ dimartbez;
  xnode:= aOl.items.addchild(xnode, dbcontent);
  for i:= 0 to treechilds.Count -1 do
     TDims(treechilds.items[i]).FillTree(aOl, xnode);
end;
```

Auch das Auslesen erfolgt wieder rekursiv, da die Routine `FillTree`() jeden Knoten nach zugehörigen Zweigen durchsucht und schlussendlich mit dem übergebenen Treeview auch sichtbar macht. `Treechilds` ist ein Property vom Typ `TList`, welches Zugriff auf das Feld `FChilds` des Containers hat.

Auf die verschiedenen Knoten eines `TTreeview` kann man mit dem `TTreeNodes`-Objekt in der Eigenschaft `Items` zugreifen. Knoten kann man beliebig hinzufügen, löschen, einfügen und innerhalb des Baumdiagramms verschieben. Mit der Eigenschaft `Items` des Baumdiagramms können Sie auch auf die Baumknoten zugreifen.

Verwendung

Wer jetzt vor lauter Bäumen die Kinder nicht mehr sieht: Zugegeben, das Beispiel ist infolge der Rekursion nicht einfach. Dafür wird die Verwendung mit Verzeichnissen einfacher. Jede Verzeichnisstruktur in einem Betriebssystem kann ein Composite als Kandidat einsetzen. Jede Library oder Sammlung macht Gebrauch von diesem Muster.

Mithilfe eines `TDirectory`-Objekts können andere Komponenten mit einem Verzeichnis interagieren. Die andere Komponente (der Client) kann durch die `TDirectory`-Instanz Informationen zum Inhalt des Verzeichnisses verwalten und verschiedene Dateioperationen durchführen.

Der Client muss zur Kommunikation mit dem Verzeichnis-Objekt die Schnittstelle `IDirectoryClient` implementieren. Ein `TDirectory`-Objekt führt eine interne Liste

der Dateien im Verzeichnis. Auf diese gefilterte Liste kann die Operation mit den Eigenschaften und Methoden des Objekts zugreifen.

> *Die Klasse `TDirectory` lässt sich direkt oder als Basisklasse für neue Verzeichnis-Objekte verwenden. Sie enthält verschiedene virtuelle Methoden, die man in abgeleiteten Klassen überschreiben kann oder sogar muss.*

Auch mit `CustomSort` wird die Knotensortierung bzw. die Umsortierung an- bzw. ausgeschaltet, und zwar anhand einer Vergleichsroutine, die vom Parameter `SortProc` bestimmt wird.

Der Parameter `Data` wird an die Vergleichsroutine übergeben. Der optionale Parameter `ARecurse` (Vorgabewert: true) legt fest, dass beim Sortiervorgang das Baumdiagramm rekursiv durchlaufen wird und jeweils auch die Unterzweige sortiert werden.

Als weitere konkrete Verwendung dieser theoretischen Einführung sei ein Beispiel von Jochen Fromm aufbereitet, das ein Verzeichnis als eine Komposition von Dateien betrachtet, die eine entsprechende durchgehende Kopieroperation implementiert.

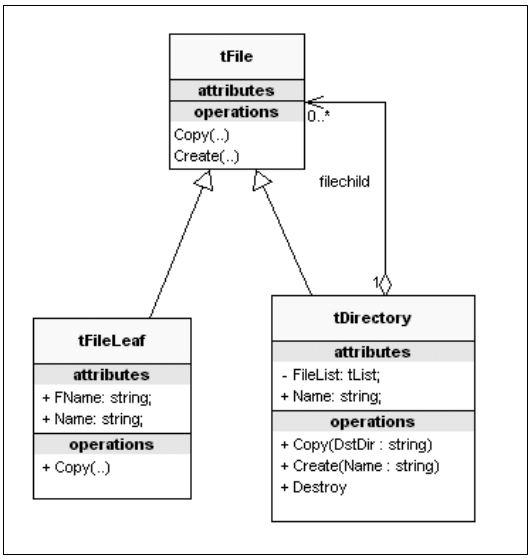


Abb. 2.26: Traversieren mit dem Composite

Typisch ist auch hier wieder die direkte Übergabe der erzeugten namenlosen Instanz in eine Liste, welche die Kinder repräsentiert. Ein Verzeichnis kann Dateien, aber auch sich selbst enthalten, die Datei ist ja nur ein Typ, anders ausgedrückt, ein Verzeichnis ist ein spezieller Dateityp. Auch die Rekursion innerhalb des Containers `TDirectory` zeigt das Prinzip der „fliegenden“ Instanzierung direkt aus dem Konstruktor.

```
    then FileList.Add(tDirectory.Create(Root+sr.Name))
```

`FindFirst` sucht im Verzeichnis `Path` nach dem ersten Eintrag, der mit dem angegebenen Dateinamen und den festgelegten Attributen übereinstimmt. Das Ergebnis wird im Parameter `sr` zurückgegeben. `FindFirst` gibt im Erfolgsfall 0 zurück, ansonsten einen Fehlercode. `FindClose` beendet eine `FindFirst/FindNext`-Aufruffolge.

```
constructor TDirectory.Create(Name: string);
var Root,s: string;
    sr: tSearchRec;
begin
  inherited Create(Name);
  FileList:= tList.Create;
  Root:=IncludeTrailingPathDelimiter(Name);
  s:=Root+'*.*';
  if FindFirst(s, faAnyFile, sr) = 0 then begin
    repeat
      if (sr.Name = '.') or (sr.Name = '..') then continue;
      if ((sr.Attr and faDirectory) <> 0)
       then FileList.Add(tDirectory.Create(Root+sr.Name))
       else FileList.Add(tFile.Create(Root+sr.Name));
    until FindNext(sr) <> 0;
    FindClose(sr);
  end;
end;
```

Der Typ `TSearchRec` definiert die Dateiinformationen, nach denen die Routine mit `FindFirst` oder `FindNext` sucht.

Mit `RootDirectory` kann das Verzeichnis angegeben werden, das als root-Knoten in der Verzeichnishierarchie auftaucht. Der Eigenschaft lässt sich jeder gültige Verzeichnisname zuweisen. Sie gibt aber immer einen vollständigen Pfadnamen ohne abschließendes Begrenzungszeichen zurück.

C#-Code-Beispiel mit Array-Struktur:

```
public TDirectory(string Name) {
//string Root;
  FName = Name;
  string[] arrFiles = Directory .GetFiles(FName);
  for (short int16Counter = 0;
    int16Counter < arrFiles.Length; int16Counter++){
    FileList.Add(new TFile(arrFiles[int16Counter]));
    Console.WriteLine("TFile-Objekt create for: " + FName);
  }
  string[] arrDirectories =
    Directory.GetDirectories(FName);
  for (short int16Counter = 0;
    int16Counter < arrDirectories.Length; int16Counter++){
```

```
    FileList.Add(new TDirectory
            (arrDirectories[int16Counter]));
    Console.WriteLine("TDirectory-Objekt create: " + FName);
  }
}
```

Abschließend noch die Hauptfunktion mit `Copy()`, die eine Datei oder ein ganzes Verzeichnis mit Untereinträgen kopiert. Dies erlaubt eben, den Unterschied zwischen einer Komposition von Objekten und einzelnen Objekten nicht machen zu müssen, da das Ganze immer mehr ist als die Summe seiner Teile ;).

```
procedure tDirectory.Copy(DstDir: string);
var i : integer;
    RelPath : string;
begin
  if not DirectoryExists(DstDir) then
    ForceDirectories(DstDir);
  for i:=0 to FileList.Count-1 do
   if tFile(FileList[i]) is tDirectory then begin
             relPath:=ExtractRelativePath
             (IncludeTrailingPathDelimiter(Name),
                   tDirectory(FileList[i]).Name);
     tDirectory(FileList[i]).Copy(DstDir+'\'+RelPath)
   end else
     tFile(FileList[i]).Copy(DstDir)
end;
```

C#-Code-Beispiel mit Path-Operationen: Im Unterschied zu erstem Bsp. kann `DstDir` sowohl Verzeichnis wie auch Datei sein:

```
  public void Copy(string DstDir) {
    string RelPath;
    if(!Directory.Exists(DstDir)) {
      Directory.CreateDirectory(DstDir);
    }
    for (int i = 0; i < FileList.Count; i++) {
      string strKomponentenName =
                        (IKomponente)FileList[i]).Name;
      string strNeu = strKomponentenName.
         Substring(strKomponentenName.LastIndexOf("\\"));
      ((IKomponente)FileList[i]).Copy(DstDir + strNeu);
    }
  }
```

Mit `ExtractRelativePath` kann man eine vollständige Pfadangabe in eine relative Pfadangabe umwandeln. Der Parameter `DestName` gibt den zu konvertierenden Datei-

namen (einschliesslich Pfad) an. `BaseName` ist der vollständige Pfad des Verzeichnisses, zu dem der zurückgegebene Pfadname relativ sein soll. `BaseName` kann einen Dateinamen oder muss eine endgültige Pfadbegrenzung enthalten.

2.4.4 Decorator

Ein objektbasiertes Strukturmuster (Zuständigkeiten ohne neue Unterklassen)

Zweck

> *Erweitere ein Objekt dynamisch mit Methoden und Zuständigkeiten, ohne viele neue Unterklassen zu bilden. Dekorierer erweitern die Funktionalität mittels Delegation. (Gut dekoriert ist halb gebaut!)*

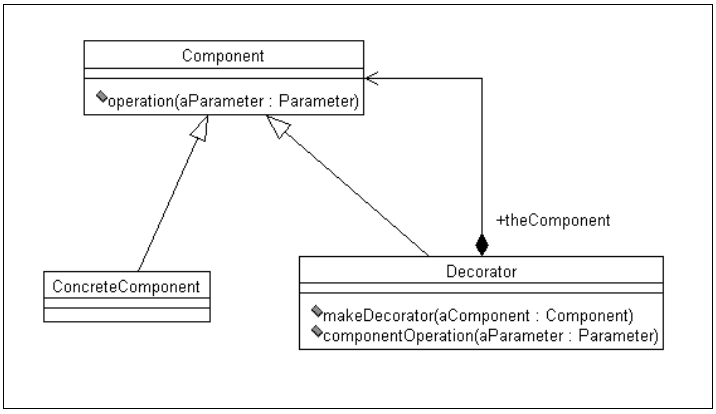


Abb. 2.27: Mit dem Decorator erweitern

Motivation

Gelegentlich wollen wir Methoden einzelner Objekte ändern, ohne die bestehenden Klassen zu verändern. Vielleicht kennen Sie die „Tales of Crypt“ (Geschichten aus der Gruft), die ihre Geschichten auch ändern, ohne die Story zu belangen.

Anhand einer kleinen Verschlüsselungsroutine möchte ich den Decorator vorstellen, um einen weiteren Baustein der OO-Welt näher zu bringen. Der klassische Decorator hat zudem Eigenschaften des Composite Patterns (siehe oben), welches ziemlich zentral in der Patternwelt verankert ist.

Dekorierer bieten eine Alternative zum Subclassing, sodass sich flexiblere Beziehungen ergeben. Vielfach erweitert man Klassen einfach mit neuen Methoden.[5] Es gibt zwar etliche Klassen, die mit Methoden erweitert werden, aber diese Methoden stammen nicht

[5] Patterns setzen eigene Klassen voraus, welche das Prinzip kapseln.

von einer eigenen Klasse. Zumal hat der Decorator eine Ähnlichkeit mit dem Wrapper, dem ich mich in 2.4.8 widme.

Die Unterklassenbildung wird meist mit klassenbasierten Mustern wie Vererbung gelöst, um die Basisfunktionalität der Klasse dynamisch zu erweitern. Aber durch die Vererbung alleine ist nur eine statische Erweiterung möglich, die auch in die bestehende Vererbungshierarchie eingreifen muss.

Das Problem stellt sich so dar: Die Funktionalität eines Objekts soll erweitert werden, ohne die bestehende Klasse zu ändern. Die Vererbung ist inflexibel und nicht zur Laufzeit möglich, zumal Klassendefinitionen versteckt sein können oder aus anderen Gründen zum Ableiten nicht ersichtlich sind.

Mit dem Decorator lässt sich nun die zusätzliche Funktionalität um das Objekt umschließen (auf gut Deutsch einhüllen), sozusagen als funktionelle Dekoration. Dies hat den Vorteil, dass die Vererbungshierarchie übersichtlich bleibt und die erweiterte Funktionalität auch wieder entfernt werden kann. Weitere Vorteile sind:

- Bessere Flexibilität im Vergleich zu statischer Vererbung
- Erlaubt schlanke, leichte Komponentenklassen
- Dekorierer lassen sich verschachteln
- Stabile Basisklasse bleibt unangetastet (ohne nachträglich `virtual` einzubauen)
- Kombinierbar mit dem Composite Pattern

Der Decorator ist ein spezieller Fall des Composite, da nur genau eine Rückführung (1..1) als Kardinalität vorhanden ist. Denn der Decorator leitet seine Aufrufe an sein Komponenten-Objekt weiter und verwaltet keine Listen. Somit lassen sich auch vor oder nach dem Weiterleiten des Aufrufs zusätzliche Operationen ausführen (siehe Beispiel der folgenden Kryptoklasse).

Wichtig ist auch die Abgrenzung zum Adapter, da sowohl Adapter wie Decorator gelegentlich Wrapper genannt werden. Der Adapter ändert die Schnittstelle und behält die Funktionalität bei, der Decorator dagegen erweitert die Funktionalität und lässt die Schnittstelle unverändert.

Implementierung

Als Erstes benötige ich eine kleine Basiskomponente, die exemplarisch einen String verschlüsselt. Unser Ziel ist es, eine einfache XOR-Verschlüsselung zu implementieren, wobei ich das Gebiet der Kryptologie nur am Schluss noch kurz streife, da der Schwerpunkt ja auf dem Design liegt. Bevor wir mit dem Ableiten des Patterns beginnen, bauen wir mal die Klasse, die dann dekoriert, d. h. zur Laufzeit erweitert wird:

```
TCrypto = class (TObject)
 private
   FcryptKey: LongInt;
   FReferenceCnt: Integer;
 protected
   function Referenced: Boolean;
   procedure SetcryptKey(const Value: LongInt);
```

```
    procedure SetReferenced(IsReferenced: Boolean);
  public
    procedure AddReference;
    function encrypt(const cstring: string): string;
    property cryptKey: LongInt read FcryptKey write SetcryptKey;
end;
```

Viele Tools, z. B. ModelMaker, bieten einen Pattern-Generator an, den wir nun exemplarisch aktivieren. Die Möglichkeit, bestehende Klassen mit Patterns zu erweitern, ist bei den Strukturmustern am ehesten gegeben (wie auch Adapter oder Wrapper). Bei den Erzeuger- oder Verhaltensmustern muss teilweise auch die bestehende Klasse angepasst oder von Anfang an geplant werden.

Der folgende Dialog zeigt die nötigen Angaben, die zur Generierung des Patterns vonnöten sind, vor allem die zu erweiternden (zu dekorierenden) Funktionen, in unserem Fall die Methode `encrypt`. Die muss man in der bestehenden Klasse markieren, sodass sie sich dann in einem Rutsch hinzufügen lässt. Der Generator erzeugt also eine neue Klasse `TCryptoDecorator2`.

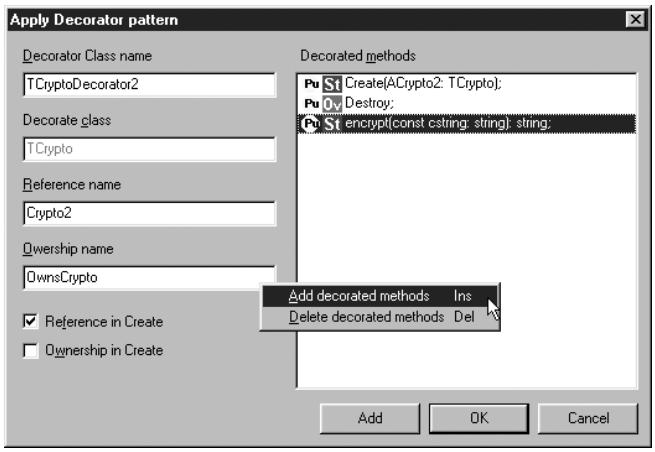


Abb. 2.28: Mit dem Decorator erweitern

Die Unit *cryptobox.pas* wird schlussendlich aus vier Klassen bestehen, der Komponentenklasse und den drei Dekorierern, die ich (resp. ModelMaker) nacheinander erzeugt habe. Den eigentlichen Nutzen will ich dann mit einer zweifachen Referenz zeigen. Am besten stellt man sich das so vor: Beim Aufruf erfolgt zwischen den Klassen eine horizontale Kommunikation innerhalb des Konstruktors, da im Parameter jeweils die nächste Referenz bis zum eigentlichen Dekorieren weitergegeben wird. Normalerweise hat man eine vertikale Kommunikation entlang derVererbung.

Die abgeleitete Klasse als Dekorierer benötigt dann diesen Referenznamen (Crypto2), der dann als Zeiger zur Basisklasse dient. Die generierte Klasse hat nach einigen Minuten folgende Struktur erhalten:

```
TcryptoDecorator2 = class (TCrypto)
  private
    FCrypto: TCrypto;
    function GetCrypto: TCrypto;
    procedure SetCrypto(Value: TCrypto);
  public
    constructor Create(ACrypto: TCrypto);
    destructor Destroy; override;
    function encrypt(const cstring: string): string;
    procedure SetcryptKey(const Value: LongInt);
    //property Crypto2: TCrypto read GetCrypto write SetCrypto;
end;
```

Das Property `Crypto2` wird nicht unbedingt benötigt. Augenfällig ist der Konstruktor, der als Parameter die Instanz der Basisklasse erhält. Welche Idee dahinter steht, sehen wir gleich später. Zusätzlich erhält die neue Klasse ein privates Feld, den erwähnten Member `FCrypto`, das dann zur Laufzeit die Instanz der Basisklasse `TCrypto` (als ConcreteComponent) erhält.

Das nun Typische am Decorator ist die doppelte Relation von Vererbung und Aggregation zwischen den Klassen, welche maximale Flexibilität erlaubt und sich durch die Aggregation gut steuern lässt. Schlussendlich bestimmt ja der Client zur Laufzeit, wann welcher Dekorierer zum Zuge kommt.

Der eigentliche Konstruktor übernimmt die Referenz der Basiskomponente und weist sie dem Member `FCrypto` oder `Crypto2` zu. Gewissermaßen besteht eine Ähnlichkeit zum Proxy, welches „nur“ eine reine Weiterleitung bewirkt und keine Erweiterung an sich ist. Nicht zu vergessen sei das Freigeben des Member im Destruktor:

```
constructor TCryptoDecorator2.Create(ACrypto: TCrypto);
begin
  inherited Create;
  FCrypto:= ACrypto;
end;

destructor TCryptoDecorator2.Destroy;
begin
  FCrypto.free;
  FCrypto:= NIL;
  inherited Destroy;
end;
```

C#-Code-Beispiel: (C# benötigt keine expliziten Destruktoren)

```
  public class TCryptoDecorator2 : TCrypto {
    private TCrypto FCrypto;
    private long FCryptKey;
```

```
    //Konstruktoren
    public TCryptoDecorator2 (TCrypto ACrypto) {
      FCrypto = ACrypto;
    }
..}
```

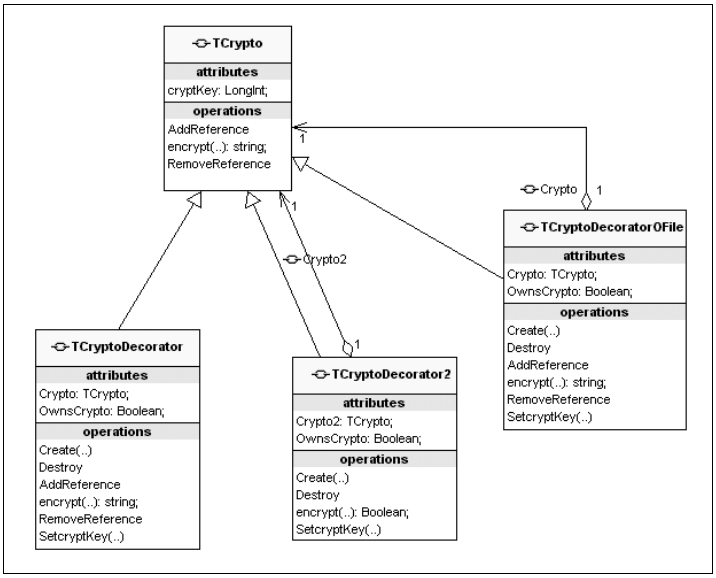


Abb. 2.29: Eine Basisklasse lässt sich nach Wahl dekorieren

Nachdem die Unit soweit gebaut ist, werfen wir einen Blick auf den Aufruf des Clients sowie auf das erwähnte Dekorieren um die Methode `encrypt` herum.

> *Die aufrufende Klasse, Client, erzeugt die Konkrete Komponente und den oder die Dekorierer und übergibt die konkrete Komponente als Parameter in den Dekorierer. Unter Umständen geschieht dies durch mehrfach verschachtelte Konstruktoren.*

Beim Aufruf werden gleich zwei Konstruktoren angeworfen, der eigentliche Dekorierer und die Referenz der Basisklasse zugleich, welche sich direkt als Parameter erzeugen lassen. Man spricht hier auch von Indirektion, eine Technik, die auch das Strategie Pattern effektvoll einsetzt.

Jeder weitere Aufruf wird mit der horizontalen Aufrufkaskade an die Basisklasse mit oder ohne „Dekoration“ weitergeleitet. Das Weiterleiten ist durch den Member `FCrypto` möglich, den ich ja vorher als Parameter erhalten habe:

```
result:= FCrypto.encrypt(cstring); //dispatch
```

Die Methode `encrypt` lässt sich nun dekorieren, in unserem Beispiel mit einer Validierung **vor** und dem Schreiben in ein File **nach** der Methode. Hier wird nun die mögliche Verschachtelung deutlich, die auch rekursiv möglich ist. Man kann hier einwenden, dies sei auch mit `override` möglich, aber das Nachteilige ist eben: einmal virtuell, immer virtuell, somit ist auch nur vertikale Kommunikation möglich.

```
with TCryptoDecorator.create(TCrypto.create) do begin
  setCryptKey(ccrypt_key); //longint
  edtGeheim.text:=encrypt(crstring);
  free;
end;
```

Nach dem horizontalen Einspringen in die Methode `encrypt` ist das vertikale Weiterleiten an die Oberklasse angesagt:

```
function TCryptoDecorator2.encrypt(const cstring: string):string;
var cryptoF: TextFile;
begin
  if (length(cstring) > 255) then raise
    ECryptError.create('only shortstring possible');
  result:= FCrypto.encrypt(cstring); //dispatch
  AssignFile(cryptoF, 'mycrypt.txt');
  Rewrite(cryptoF);
  writeln(cryptoF, result);
  CloseFile(cryptoF);
end;
```

C#-Code-Beispiel mit Schreiben in Datei:

```
 public override string encrypt(string cstring) {
  TextWriter q;
  string strReturn;
  //Aufruf der dekorierten Methode
  strReturn = FCrypto.encrypt(cstring);
  try  {
  //dekorierende Zusatzfunktionalität: In Datei schreiben
    FileStream objFileStream =
    new FileStream(TCrypto.CRYPTFILE, FileMode.OpenOrCreate,
                                        FileAccess.ReadWrite);
    q = new StreamWriter(objFileStream);
    q.WriteLine(strReturn);
    q.Flush();
    q.Close();
  } catch (Exception objException) {
```

```
        Console.WriteLine(objException);
    }
 return strReturn;
 }
```

Der faszinierende Teil passiert nun: Das Pattern erlaubt mir, zwei oder n Dekorierer gleichzeitig einzusetzen. Zudem erreiche ich immer die Oberklasse, obwohl keine der Methoden virtuell ist! Ich möchte z. B. das Chiffrat nebst dem Speichern in ein normales File noch in ein XML-File speichern. Dies ist mit einer Verschachtelung der Referenzen tatsächlich zur Laufzeit möglich. Die Reihenfolge der Aufrufkaskade über den Konstruktor bedeutet im folgenden Aufruf von rechts nach links gelesen:

1. Verschlüsseln (mit `TCrypto`)
2. in ein File schreiben (mit `TCryptoDecorator2`)
3. in ein XML schreiben (mit `TCryptoDecoratorOFile`)

```
with TCryptoDecoratorOFile.create
           (TCryptoDecorator2.create(TCrypto.create))//client
do begin
   setCryptKey(23);
   self.caption:=encrypt('c~d7~d7zvo7qexz7gvo');
   free;
end;
```

C#-Code-Beispiel mit Verschachtelung der Konstruktoren:

```
 TCrypto objTCrypto = new uCrypto.TCrypto();
  objTCrypto.SetcryptKey(23);
  string strTest = "c~d7~d7zvo7qexz7gvo";
  rtxtbAusgabe.Text = (new TCryptoDecorator2
                          (objTCrypto)).encrypt(strTest);
  rtxtbAusgabe.Visible = true;
 }
```

Achtung! Wettbewerb 1: Wer zuerst das Chiffrat `('c~d7~d7zvo7qexz7gvo')` mit untenstehender XOR-Routine entschlüsseln kann, gewinnt einen UML-Kurs; bitte Mail an mich (es sind noch zwei weitere Wettbewerbe in Teil 2 enthalten).

Zum besseren Verständnis sei noch das Sequenzdiagramm beigefügt, welches das Weiterleiten mit dem horizontalen Zusatznutzen des Decorators darstellt.

Beim Kryptoalgorithmus handelt es sich um einen simplen symmetrischen XOR-Schlüssel, der wie eine normale binäre Addition pro `Char` funktioniert. In diesem Bereich gibt es andere Kaliber[iv], zu finden unter: www.torry.ru\security oder DelphiSuperPage. Der Operator `XOR` hat die nette Eigenschaft, umkehrbar zu sein, d. h., `encrypt` (Verschlüsseln) und `decrypt` (Entschlüsseln) bedeuten in meiner Anwendung dieselbe Funktion.

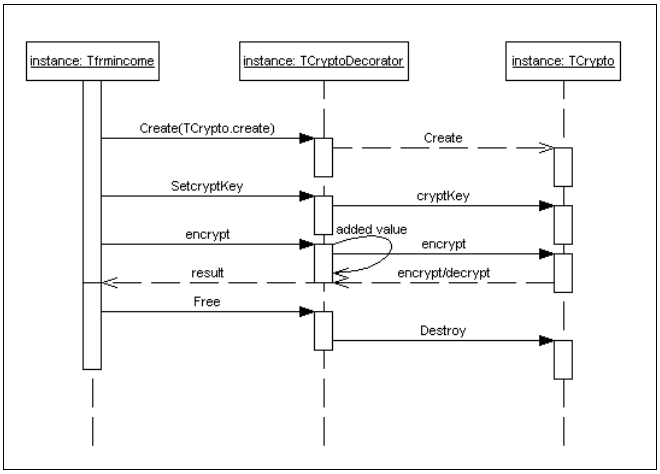


Abb. 2.30: Weiterleiten von der horizontalen Unter- zur Oberklasse

Beispiele für symmetrische Verfahren sind DES und IDEA,[6] während RSA das bekannteste (und erste) asymmetrische Verfahren ist. Abschließend seien noch ein paar Gedanken zur ewig umstrittenen Schlüssellänge erwähnt. Der Aufwand für eine Brute-Force-Attacke ist einfach zu berechnen. Bei einer Schlüssellänge von 8 Bit ergeben sich 2 hoch 8 (256) Möglichkeiten.

Den Schlüssel zu finden, benötigt demnach 256 Versuche, wobei eine Chance von 50 % besteht, den Schlüssel nach der Hälfte der Versuche gefunden zu haben. Bei einem 56-Bit-Schlüssel rechnet ein Mainframe, welcher in der Sekunde 1 Million mal probiert, 2286 Jahre für das Finden des exakten Schlüssels. Bei 64 Bit dauert die Suche 58500 Jahre und bei 128 Bit dauert das Hacken des Schlüssels mittels Brute-Force 10 hoch 25 Jahre.

```
function TCrypto.encrypt(const cstring: string): string;
var
  s: string[255];
  c: array[0..255] of Byte absolute s;
  i: Integer;
 begin
  s:=cstring;
    for i:= 1 to length(s) do c[i]:= c[i] XOR FcryptKey;
  result:= s;
end;
```

[6] Hier liegt eine Software-Patentierung durch ascom.com vor.

C#-Code-Beispiel mit Konverter:

```
public virtual string encrypt(string cstring) {
  string s;
  char[] c;
  s = cstring;
  c = s.ToCharArray();
  for (int i = 0; i < s.Length; i++) {
    c[i] = Convert.ToChar(Convert.ToInt16(c[i]) ^ FCryptKey);
  }
  s = new String(c).ToString();
  return s;
}
```

Verwendung

Als mögliche Anwendung sind auch Komprimierungstools, Filter für Soundboxen, Formatierer oder Streamer anzutreffen, die meistens eine Basiskomponente aufweisen und zusätzliche Klassen als Dekorierer.

Ein interessantes Beispiel habe ich in der CLX gefunden, das in Kombination mit einem Adapter die konkrete Klasse eines Interfaces dekoriert. Auch das ist möglich. Wie bereits in der Einleitung erwähnt, unterscheidet ein Decorator sich von einem Adapter in dem Sinne, dass ein Decorator das Funktionieren eines Objekts verändert, nicht aber die neue Schnittstelle wie der Adapter.

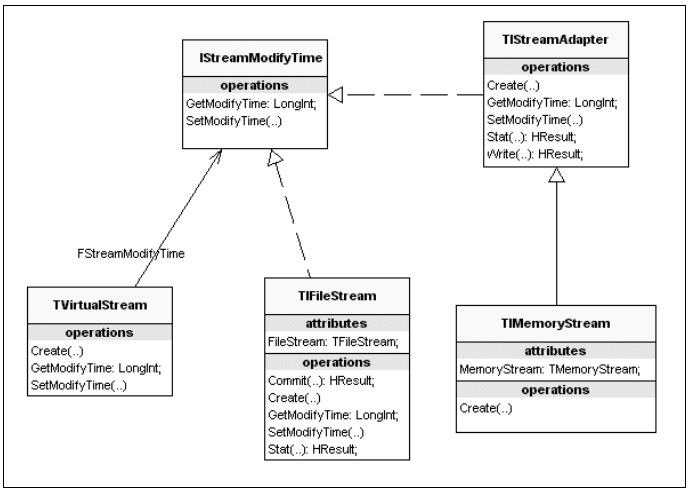


Abb. 2.31: Das Dekorieren einer Schnittstelle

Die erweiterte Funktion, die der Decorator hinzufügt, ist das Setzen eines Zeitstempels mit `SetModifyTime`. Unter Windows ermöglicht `TStreamAdapter`, dass ein `TStream`-Objekt als COM-basierte `IStream`-Schnittstelle verwendet wird.

```
procedure TVirtualStream.SetModifyTime(Time: Longint);
begin
  if FStreamModifyTime <> nil then
    FStreamModifyTime.SetModifyTime(Time);
end;
```

Wenn also ein Objekt von `TFileStream` eine Bewegung schreibt, lässt sich mit dem Schnittstellen-Decorator zusätzlich der Zeitstempel setzen. Die von `IStream` definierte Methode `Commit` trägt dann die Änderungen ein, die im Rahmen einer Transaktion in einen Stream geschrieben wurden.

2.4.5 Facade

Ein objektbasiertes Strukturmuster (Schnittstelle zu einem Subsystem)

Zweck

Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Facade-Klasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht.

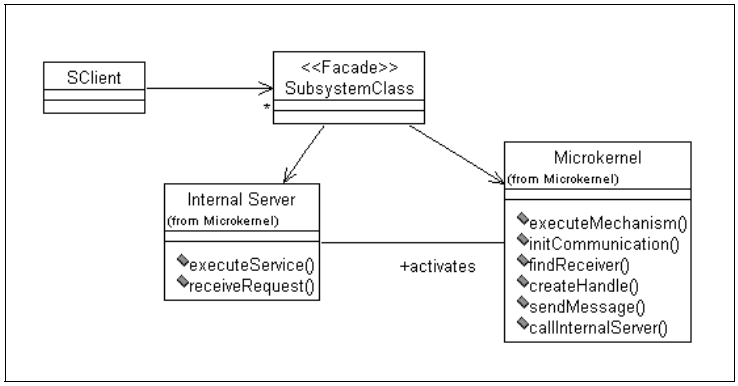


Abb. 2.32: Ein Subsystem wird gekapselt

Motivation

Patterns lassen sich ja direkt als Kontext beschreiben: So z. B. besteht ein Design letztendlich aus vielen Klassen, die in einem System schwer zu kontrollieren sind. Um dieses Problem zu entschärfen, also ganze Subsysteme zu einer einzelnen Instanz zusammenzufassen, wurde das Facade Pattern entwickelt. Dieses ermöglicht, ein ganzes Subsystem wie ein Objekt zu behandeln. Doch es ist ein Ansatz, der nicht in jedem Fall zum gewünschten Ergebnis führt.

Komplexität zu reduzieren, erreicht man meist, indem man ein System in Subsysteme unterteilt. Will nicht heißen, dass es auch Subversionen gibt ;). Die Systemarchitektur mit der zugehörigen Plattform beschreibt dann die Gesamtorganisation des Systems in Komponenten, Klassen oder Teilsystemen.

Die Möglichkeit, die das Facade Pattern bietet, ist, eine gesamte Schnittstelle für all die Funktionalität im Subsystem anzubieten. Eine Anwendung kann dann auf die Schnittstelle der Facade zugreifen, die sich auf demselben Computer wie die Anwendung oder auf einem anderen in das Netzwerk eingebundenen Computer befindet.

Eine Schnittstelle besteht in der Facade aus einer Reihe von Methoden-Prototypen, deren Funktionalität die Schnittstelle definiert.

> *Zu diesen Methodendefinitionen (Prototypen) gehören die Anzahl und die Typen der Parameter, der Rückgabetyp und das erwartete Verhalten.*

Die Facade kann auch eine Sammlung von Methoden beinhalten, die im Subsystem dann überschreibbar sind. Die Art und Weise, in der man diese Methoden implementiert, ist nicht festgelegt, sodass in Bezug auf die Schnittstelle die tatsächliche Implementierung immer vollständig verborgen ist.

> *Man verwendet eine Facade, um einen bekannten Eintrittspunkt zu jeder Subsystemschicht, die voneinander abhängig sind, anzubieten*

Im zusätzlichen Begriff der **Session Facade** meint man das Verstecken der Komplexität von Business-Objekten und die zentralisierte Workflow-Verarbeitung.

Die Bedeutung von COM besteht im Wesentlichen darin, dass dieses Modell durch klar definierte Schnittstellen die Kommunikation zwischen Komponenten, zwischen Anwendungen und zwischen Clients sowie Servern ermöglicht.

Verwandtes Muster ist die Abstrakte Fabrik, das sich mit der Facade verwenden lässt. Eine abstrakte Fabrik ist als Alternative einsetzbar, um plattformspezifische Klassen zu kapseln. Facades sind meistens als Singleton implementiert.

Implementierung

Im Falle TRadeRoboter ist die ganze Mechanik des Chartgenerators als Facade im Einsatz. Für Clients sind die inneren Objekte und Methoden des Generators nicht sichtbar. Der Client will einfach den Chart als Gesamtes, d. h. mit Statistik, Speicherverwaltung, Optimierung der Kauf- und Verkaufssignale, auf den Schirm bringen.

Durch die Instanz `myGen` hat der Client freie Wahl unter den benötigten Subsytemklassen, die meist auch Routinen anderer Units sind, wie die Berechnung der Statistikwerte. `MeanAndStdDev` berechnet z. B. das arithmetische Mittel (Mean) und die Standardabweichung (StdDev) in einem Aufruf. Die Funktion ist dadurch doppelt so schnell wie die getrennte Berechnung. Ist der Mittelwert extrem groß (>10e7) oder die Varianz sehr klein, können Ungenauigkeiten auftreten.

Das Subsystem ist auch für die Datenhaltung verantwortlich. `Data` enthält die zu analysierenden Daten. In C++ bezeichnet der Parameter `Data_Size` den Index des letzten Elements im Array `Data` (eins kleiner als die Anzahl der Elemente).

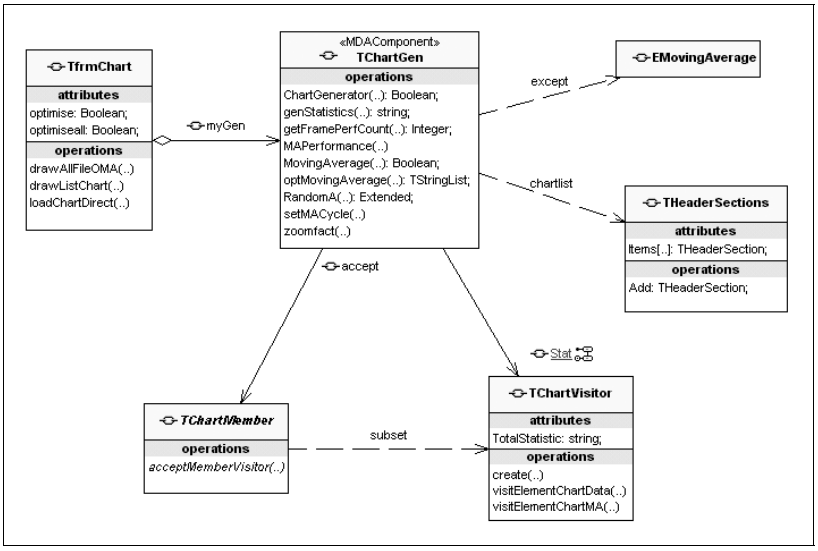


Abb. 2.33: Ein Zugriffspunkt für den Generator

M1 bis M4 geben in der CLX die ersten vier Momente zurück (M1 ist der Mittelwert, M2 die Varianz usw.). `Skew` gibt die Symmetrie der Verteilung an und `Kurtosis` gibt den Häufigkeitsgrad an.

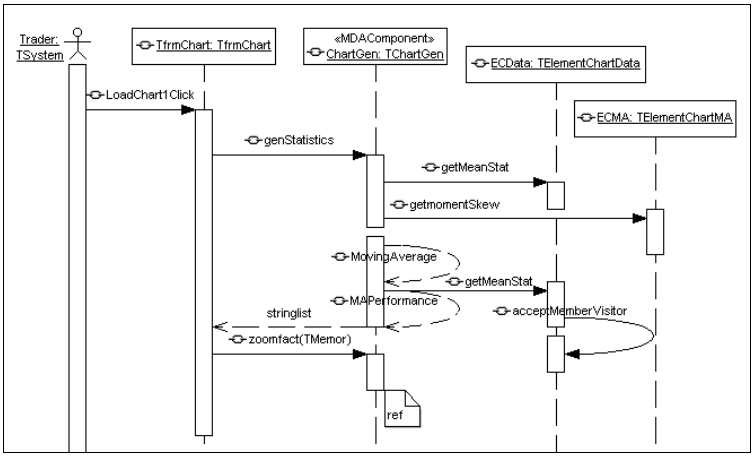


Abb. 2.34: Das Fassaden-Objekt delegiert

Die Unit *Math.pas* hält die Deklaration der Typen als `Array of double` parat, welche die Statistik-Methoden des Chartgenerators benötigen:

```
procedure TChartGen.letPrimeStatistics(const chartData:
                  TMemor; var mn, std: extended);
```

```
begin
  MeanAndStdDev(chartData, mn, std);
end;

procedure TElementChartMA.getmomentSkew(const data:TMAverage;
                var m1,m2,m3,m4, skew, kurtosis: extended);
begin
  momentSkewKurtosis(data, m1,m2,m3,m4, skew, kurtosis)
end;
```

Verwendung

Hier ist das ToolAPI von ModelMaker zu erwähnen, auch andere ToolAPIs setzen das Muster in etwa ein. Alle API-Deklarationen werden in einer einzigen Unit namens `MMToolsAPI` zusammengefasst. Das Framework ist dem Delphi ToolAPI äußerst ähnlich, was auch beabsichtigt war.

Das ToolAPI besitzt zwei Arten von Schnittstellen: Schnittstellen, die man selbst implementieren muss, und Schnittstellen, welche die IDE implementiert. Die meisten Schnittstellen fallen in letztere Kategorie: Die Schnittstellen definieren den Funktionsumfang der IDE, verbergen jedoch dessen eigentliche Implementierung. Die Schnittstellen, die man selbst aufbaut, lassen sich in drei Kategorien einteilen: Experten, Notifier und Erzeuger.

Die Experten für den Builder und Delphi sind in den meisten Fällen austauschbar. Sie können also einen Experten für C++ in Delphi erstellen und compilieren (und umgekehrt). Dabei sollte die Versionsnummer der Produkte möglichst übereinstimmen – es ist aber auch möglich, Experten für zukünftige Produktversionen zu erstellen.

Ein Notifier ist ein weiterer Schnittstellentyp im ToolAPI. Die IDE benachrichtigt Ihren Experten durch einen Notifier, wenn ein ihn betreffendes Ereignis auftritt. Nachdem Sie eine Klasse zur Implementierung der Schnittstelle erstellt und den Notifier registriert haben, ruft die IDE das Objekt auf, wenn der Benutzer eine Datei öffnet, Quelltext bearbeitet oder ein Formular ändert usw.

Ein Creator ist ein letzter Schnittstellentyp, den man implementieren muss. Das ToolAPI benutzt Creators, um neue Units, Projekte und andere Dateien zu erstellen oder um vorhandene Dateien zu öffnen.

Um eine Dienstschnittstelle zu verwenden, wandeln Sie mithilfe der globalen Prozedur `InitializeExpert`, die in der Unit `MMToolsAPI` definiert ist, die Schnittstellenreferenz `MMToolServices` in den gewünschten Dienst um, d. h., der Interfacezeiger `Srv` wird in die Variable `MMToolServices` typensicher umgewandelt und kopiert:

```
procedure InitializeExpert(const Srv: IMMToolServices);stdcall;
begin
  MMToolServices:= Srv as IMMToolServices;
  Srv.AddExpert(TMMDemoExpert.Create);
```

```
  Application.Handle:= Srv.GetParentHandle;
end;
```

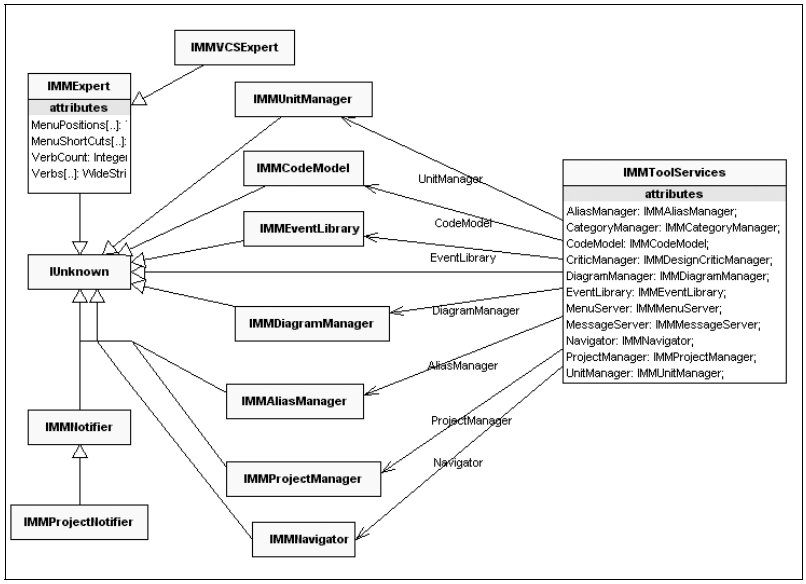


Abb. 2.35: Einen Servicepunkt mit der Facade

Das Interface der Facade (man könnte „InterFacade“ sagen) transportiert mir dann den Zeiger zum Experten der IDE, sodass mein Tool im Expertenmenü der IDE erscheint. Die Aktion ist an den Menübefehl und die Schaltfläche zu binden. Wenn Sie den Experten zerstören, müssen Sie auch die von ihm erzeugten Objekte bereinigen. Wählt der Benutzer dann einen Menübefehl, ruft die IDE die Funktion `Execute` des Experten auf.

```
procedure TMMDemoExpert.Execute(Index: Integer);
begin
  if Index = 0 then CreateUnitReport;
end;
```

Im diesem Sinne kann auch die COM-Architektur als Facade zum Einsatz kommen. Die COM-Implementierung gehört zum Win32-Subsystem, das eine Reihe von zentralen Diensten zur Verfügung stellt, welche die Grundspezifikation unterstützen.

Die COM-Bibliothek enthält eine Reihe von Standardschnittstellen, welche die Hauptfunktionen eines COM-Objekts definieren. In der Bibliothek sind API-Funktionen enthalten, mit deren Hilfe man COM-Objekte erstellt und verwaltet.

2.4.6 Flyweight

Ein objektbasiertes Strukturmuster (Optimieren von vielen Objektressourcen)

Zweck

Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können.

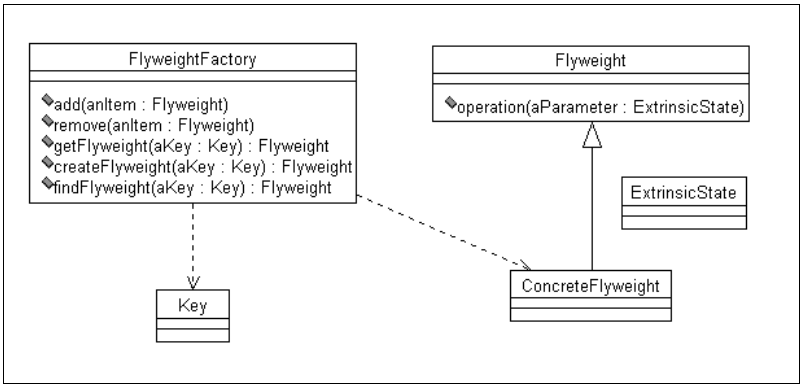


Abb. 2.36: Optimieren von Ressourcen mit dem Flyweight

Motivation

Wenn eine große Anzahl von Objekten in den Einsatz, sprich Arbeitsspeicher, gelangen muss, kann das Verwalten von Zustandsvariablen oder die Allozierung von lokalen Daten in den vielen einzelnen Objekten zu Engpässen führen. Diese Zustände mit den Objekten müssen aber speicherbar bleiben und eine gemeinsame Nutzung ermöglichen.

Von all den Patterns hatten wir noch nie den Einsatz eines Fliegengewichts geplant. Ich weiß, dieses Bekenntnis gehört nicht unbedingt zur Motivation, will aber den exotischen Charakter des Patterns aufzeigen. Charakter auch im doppelten Sinne, da die meisten Beispiele mit der Verwaltung von Buchstaben und Schriften zu tun haben.

Ein Fliegengewicht ist ein Objekt, das man gleichzeitig in unterschiedlichen Kontexten verwenden kann. Es sollte aber nicht von einem Einzelexemplar derselben Klasse unterschieden werden können. Die zentrale Idee erklärt, dass es einen Unterschied zwischen dem intrinsischen und extrinsischen Zustand des Objektes gibt.

- **Intrinsischer** Zustand: wird im Objekt als kontextunabhängiger Zustand gespeichert. Gemeinsam nutzbar.
- **Extrinsischer** Zustand: ist nicht speicherbar und ist kontextabhängig. Nicht gemeinsam nutzbar.

Demzufolge modellieren Fliegengewichte Gegenstände oder Objekte, die in der Regel so gehäuft auftreten, dass man nicht jeweils eigene Objekte einsetzen kann.

Wenn Sie jetzt an Tausende von Records denken, die man zu Objekten befördern möchte, kann ein Fliegengewicht einen Ansatz bieten.

> *Objekte sind ebenfalls Sammlungen von Datenelementen. Aber Objekte enthalten, anders als Records, auch die kontextabhängigen Prozeduren und Funktionen zur Verarbeitung ihrer Daten.*

Wenn Sie Records aus Pascal oder Structs aus C kennen, dann ist das Objektkonzept auch einfacher zu verstehen.

Zum Beispiel kann auch ein Dokumenteneditor ein Fliegengewicht für jeden Charakter im Alphabet erzeugen. Das Muster speichert dann lediglich den Zeichencode als intrinsischen Zustand. Bei der Darstellung holt sich das Fliegengewicht die Position und den Stil des Buchstabens aus dem Kontext des extrinsischen Zustandes, konkret erhält das Muster die Information aus dem aktuellen Layout und den Formatierungen.

Es liegt in der Natur von Dokumenten, dass sie in einer Anwendung in den verschiedensten Kombinationen und in wechselndem Kontext vorkommen, der Zeichencode als ANSI oder Unicode wechselt aber viel seltener.

Implementierung

Im Falle TRadeRoboter bezweckt der „geplante“ Einsatz des Musters eine Aufteilung eines Charts in einzelne Objekte. Als Chart seien die Kurven eines Wertpapiers mit ihren Berechnungen definiert. Ein Zeichnen des Charts ist dann die Erzeugung von spezifischen Grafikelementen, beispielsweise einer Linie oder einer Figur (Buy- oder Sell-Signal), durch die Routine.

Diese Routine ermöglicht dann, ein bestimmtes Grafikelement als Linie oder Symbol an einer bestimmten Stelle der Zeichenfläche zu platzieren, indem die Routine eine Zeichenmethode der Zeichenfläche aufruft.

Wie Grafiken in einer Anwendung dargestellt werden, hängt vom Typ des Objekts ab, auf dessen Zeichenfläche gezeichnet wird. In meinem Fall sind es Punkte und Linien, die anhand des internen Wertes des Wertpapiers jeweils als Pixelinformation auf die Zeichenfläche gelangen.

Wie gesagt, das Muster ist eine Planung und dient hier einer theoretischen Vorstellung bezüglich Fliegengewicht. Sein intrinsischer Zustand ist der Kurs, codiert nach Open, High, Low und Close. Dieser Kurs ergibt in der zeitlichen Verkettung die jeweiligen Kurslinien, wie ein Wochenverlauf oder eine Durchschnittslinie. Diese Linien wiederum überlagert, ergeben den Gesamtchart, wie Abb. 2.37 schematisch verdeutlicht.

Erzeugt wird der Chart durch das Aufbauen des Bildes. In der Regel ist das Bildaufbauen mit Zeichnen auf einer Fläche verbunden, sodass ein Objekt auf `OnPaint`-Ereignisse meistens mit dem Zeichnen von Grafikelementen reagiert.

> *Da Zeichenflächen nur zur Laufzeit verfügbar sind, erfolgt die Arbeit mit diesen Objekten ausschliesslich über den Code. Mithilfe von Feldern, in denen die verschiedenen Punkte gespeichert sind, kann ich die Liniendarstellung in der Anwendung verbessern.*

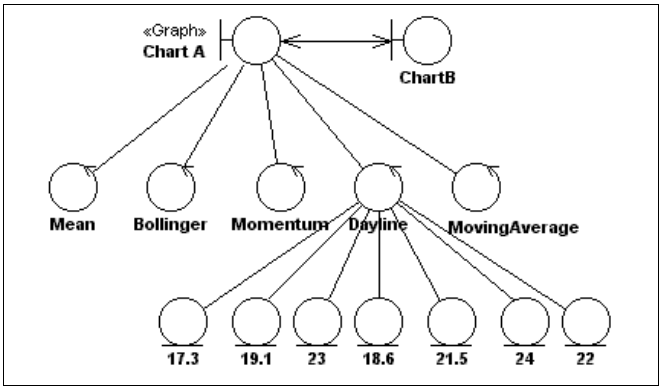


Abb. 2.37: Das Schema zum Fliegengewicht

Die momentane Datenstruktur besteht aus einem Objekt, das n-`Array of Records` beherbergt. Nach dem Fliegengewicht ist es möglich, die Records durch Objekte zu ersetzen, die nur wenig Zustandsinformation enthalten und den Rest in den Kontext verlagern. Darstellung, Farbe und Positionierung des Kurswertes sind demnach extrinsische Zustände.

Ein erster Entwurf sei ein simpler Aufzählungstyp der Grafikelemente.

```
Type
  TChartSym = (dtLine, dtBuy, dtSell, dtClose);
const
  dtLine = 0;
  dtBuy = 1;
  dtSell = 2;
  dtClose = 3;

TChartForm = class(TForm)
 public
   drawing: Boolean;
   Origin, MovePt: TPoint;
   chartTool: TChartSym;
```

Verwendung

Auch die Verwendung sei nur angedacht. Das Fliegengewicht sollte man bei folgenden Bedingungen anwenden:

- Eine große Anzahl von Objekten ist im Einsatz.
- Die Speicherkosten wie die Laufzeitkosten sind zu hoch.
- Viele Objektzustände lassen sich in den Kontext verlagern, sind somit extrinsisch.
- Die Objekte lassen sich auch gemeinsam nutzen.

Die Fliegengewicht-Schnittstelle ermöglicht eine gemeinsame Nutzung, erzwingt sie aber nicht. Am Beispiel der Fraktale können die nicht gemeinsam genutzten Objekte mit einer Aggregation als Kindobjekte in die Gesamtgrafik einfließen.

Bei einem Fraktal als Fliegengewicht sei die folgende Unterscheidung zu machen:

- Intrinsischer Zustand: ist die Codierung des Farbwertes, der gemeinsam nutzbar ist und als kontextunabhängig gilt.
- Extrinsischer Zustand: sei die generierte Bitmap mit dem entsprechenden Zoomfaktor, der ein fast beliebiges Eintauchen in die Fraktale erlaubt.
- Der Client verwaltet eine Referenz auf die Fraktale, berechnet und speichert den extrinsischen Zustand wie ein Vergrößerungsfaktor.

Zwischen dem Chartgenerator und den Fraktalen besteht auch ein Zusammenhang: Ein Fraktal besteht aus einer Menge von Punkten oder Linien, die sich mit `setup()` skalieren und mit `process()` selbst zeichnen können. Zumal die Finanzmärkte wie die Fraktale auch der Chaoslehre folgen.

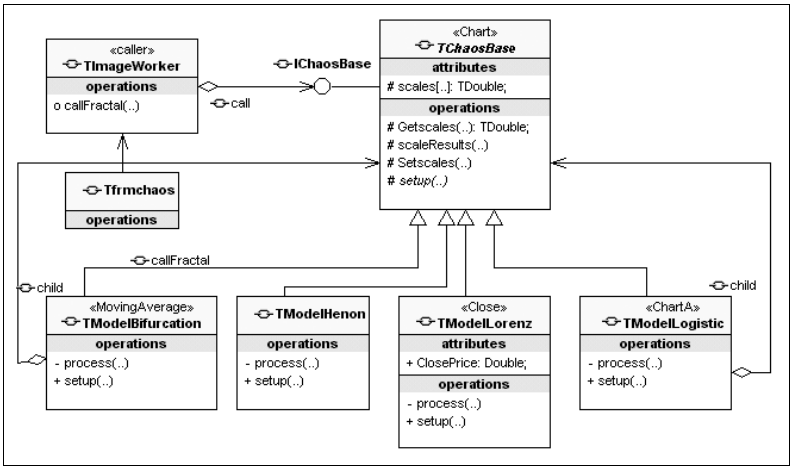


Abb. 2.38: Fraktale als Fast-Fliegengewichte organisiert

```
TmFractStandardShape = class (TSizeableShape)
  private
    FLineColour: TColor;
    FShapeType: TShapeType;
    procedure SetShapeType(Value: TShapeType);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property LineColour: TColor read FLineColour
```

```
                        write FLineColour default clRed;
    property ShapeType: TShapeType read FShapeType
                        write SetShapeType;
end;
```

2.4.7 Proxy

Ein objektbasiertes Strukturmuster (Zugriff auf einen Stellvertreter)

Zweck

Kontrolliere den Zugriff auf ein Objekt mithilfe eines vorgelagerten Stellvertreter-Objektes, das als Stellvertreter oder Gatekeeper erscheinen soll.

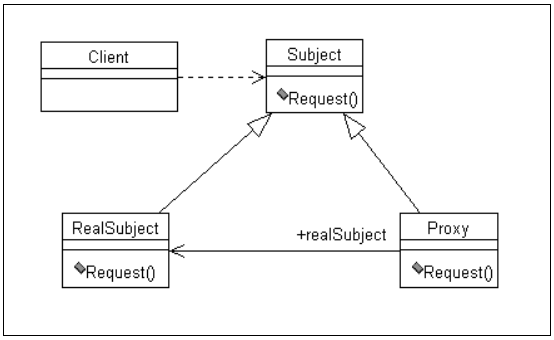


Abb. 2.39: Der Proxy sucht Realität

Motivation

Ein Proxy, auch als Surrogat bekannt, will den Zugriff auf ein Objekt kontrollieren, um z. B. aus Performancegründen erst bei Bedarf das reale Objekt zu laden. Diese teuren oder schweren Objekte werden sozusagen auf Verlangen erzeugt und bereitgestellt.

Nehmen Sie als Beispiel eine medizinische Anwendung. Die in verschiedenen Datenbanken gespeicherten Datensätze entsprechen dem persistenten Status der Anwendung, wie beispielsweise die Krankengeschichte eines Patienten. Die so genannten transaktionalen Objekte aktualisieren dann erst den Status bei Änderung: neue Patienten, Testergebnisse, Röntgendateien oder Diagnoseergebnisse usw.

Transaktionale Objekte sind normalerweise klein und werden für einzelne Geschäftsfunktionen verwendet, und mit ihnen lassen sich die Business-Rules einer Anwendung implementieren sowie Änderungen des Programmstatus bereitstellen.

Die Verbindung zwischen dem Client und dem transaktionalen Objekt wird durch einen Proxy auf dem Client und einen Stub auf dem Server verwaltet. Die Verbindungsinformationen sind unter Kontrolle des Proxys.

> *Die Client-Proxy-Verbindung bleibt so lange geöffnet, wie der Client eine Verbindung mit dem Server benötigt, und erscheint daher diesem wie eine permanente Verbindung. In Wirklichkeit aktiviert und deaktiviert der Proxy aber das Objekt bei Bedarf, damit sich die Verbindung auch von anderen Clients verwenden lässt.*

Stellvertreter sind bekannte Muster, so finden sie auch in der COM-Architektur ihre Bedeutung. Unter Sequenzbildung (Marshaling) wird der Mechanismus verstanden, der es einem Client ermöglicht, die Schnittstellenfunktionen von Remote-Objekten in einem anderen Prozessraum oder auf einem anderen Computer durchzuführen. Dieser Mechanismus funktioniert folgendermaßen:

> *Der Mechanismus übernimmt einen Schnittstellenzeiger im Prozess des Servers und stellt dem Quelltext im Client einen Zeiger auf einen Proxy-Server zur Verfügung. Dieser Stellvertreter tut so, als ob es der eigene Prozessraum wäre.*

Implementierung

Im TRadeRoboter lassen sich die aktuellen Chartdaten mit einem Webservice laden oder als `WideString` in die zentrale Bank zurückspeichern. Der Webservice hat dann den Charakter eines **physischen** Proxys, wie z. B. „Squid“ ein Proxy-Server ist und entsprechende Charts einen Tag lang zwischenspeichert. Als logischen Proxy im Sinne des Patterns habe ich einfach den vorhandenen Adapter zum Proxy erweitert.

Jeder Adapter hat ein verwandtes Muster im Proxy. Im Gegensatz zum Adapter verändert das Proxy-Muster aber nicht die Schnittstelle des Ersatzobjektes. Veränderung der Schnittstelle würde bedeuten, das Objekt der Klasse `TBMonitor2` stammt von `TObserver` ab und erbt somit die Typensignatur mit all den Parametern. Im Falle des Proxys besteht nur eine Abhängigkeit zu `TObserver`, die Typensignatur sowie die Vererbung von `TForm` bleibt beim Proxy gleich. Der Proxy kontrolliert nur den Zugriff auf das weitere Objekt.

Dabei existiert die Möglichkeit, Objekte zu generieren, welche als Stellvertreter für andere Objekte dienen. Es ist somit möglich, eine Verbindung zwischen einem Interface und einer Klasse herzustellen, ohne dass die Klasse das Interface implementieren muss. Hier wird dann konkret an eine andere Klasse delegiert.

Die Veränderung der Typen ist also beim Adapter möglich und erwünscht:

```
procedure TBankFormAdapter.update(dispCenter: TObservable);
begin
  FAdapt.Update(dispCenter);
end;
```

Veränderung der Typen beim Proxy sind jedoch nicht erwünscht, Delegation macht Sinn. Die Methode `Update` muss den Parameter `dispCenter` nicht übernehmen:

```
procedure TBankFormProxy.update;
begin
```

```
  FAdapt.Update; //Delegieren
end;
```

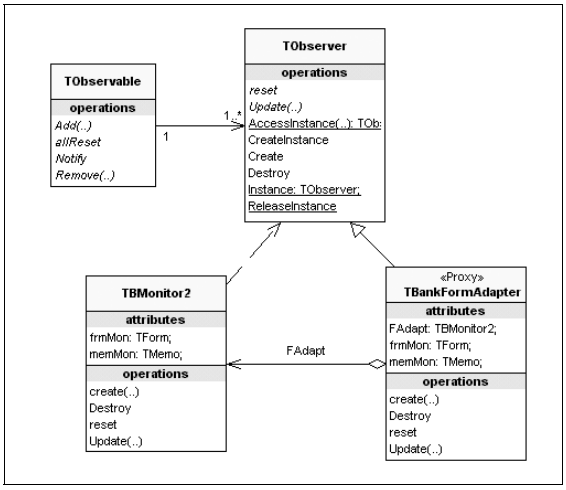


Abb. 2.40: Der Proxy findet Realität

Hier ein letztes Proxy-Beispiel, das die Anrufe von Dateioperationen weiterleitet:

```
function TCtrClass.FromFile(reportList:TStringList):TStringList;
begin
  result:=busObj.readFromFile(reportList);
end;
```

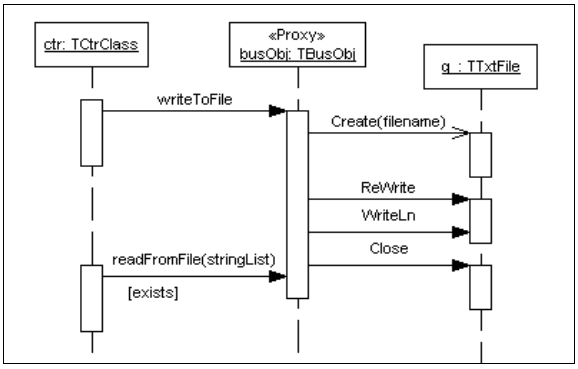


Abb. 2.41: Der Proxy im Dienst des Stellvertreters

Verwendung

Verwenden Sie einen **physischen** Proxy, wenn die Web-Verbindungskomponente den Wert von `Host` nicht lokal auflösen kann. Dies ist eine typische Proxy-Funktion. Bei einem lokal registrierten Server kann die Eigenschaft einen leeren String enthalten. Andernfalls gibt sie die Hostnamen der Proxy-Server an, die Anfragen an den gewünschten Server weiterleiten oder eben delegieren können.

Somit verwendet man einen Proxy, wenn die Client-Anwendung den Host-Anteil der URL nicht lokal auflösen kann. Wenn der Server lokal registriert ist, kann der Proxy einen leeren String enthalten. Ansonsten sind alle Hostnamen aufgelistet, die Anforderungsbotschaften zu dem gewünschten Server weiterleiten können.

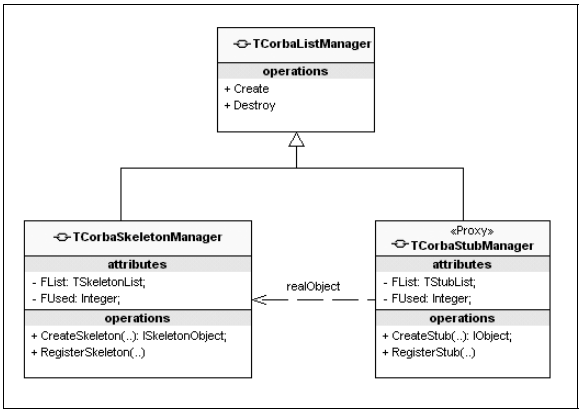


Abb. 2.42: Der Proxy als Stub

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 sich Parameterwerte und Rückgabewerte korrekt übermitteln lassen.

Anwendungen müssen entweder eine Stub- oder eine Skeleton-Klasse verwenden bzw. das „Dynamic Invocation Interface“ (DII) benutzen, das alle Parameter in spezielle Varianten konvertiert (damit diese ihre eigenen Typinformationstabellen enthalten). Der Stub ist der eigentliche Proxy, siehe Abb. 2.42.

Übrigens werden CORBA-Verbindungen von `DataSnap` nicht mehr unterstützt. `DataSnap` ermöglicht Client-Anwendungen die Verbindung mit einem Provider in einem Applikationsserver, sodass Anwendungen mehrschichtige Datenbanken nutzen können. Es sind Verbindungskomponenten, die sich von Client-Anwendungen einsetzen lassen. Mehr dazu in Teil 3 der Architektur-Patterns.

2.4.8 Wrapper

Ein klassenbasiertes Strukturmuster (Ermöglicht einer Funktionssammlung OO-Zugriff)

Zweck

Umhülle eine Gruppe von Funktionen mit einer objektorientierten Schnittstelle und erweitere somit eine andere Klasse mit der erzeugten Umhüllung.

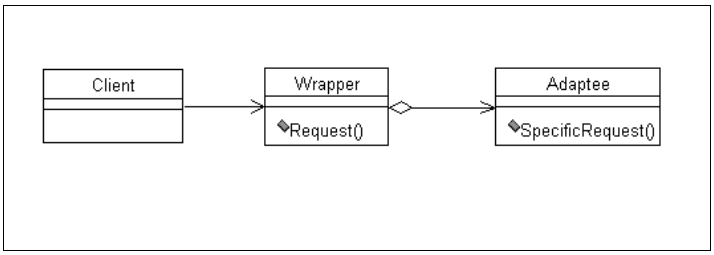


Abb. 2.43: Der Wrapper als OO-Lieferant

Motivation

Viele alte prozeduralen Anwendungen wie COBOL oder AS400, so genannte Altlasten oder Legacy-Systeme, warten darauf, im neuen Kleid eine objektorientierte Schnittstelle zu erhalten. Damit modernisiert man das GUI und den Zugriff. Heutzutage nennt man die Geschichte der Modernisierung auch „Enterprise Application Integration“ (EAI), um das unschöne Wort Altlasten zu vermeiden.

Um Informationen über einen alten Server der Client-Anwendung zur Verfügung zu stellen, kann man die Informationen über den Server importieren, die in der Typenbibliothek des Servers gespeichert sind. Die Anwendung kann dann die generierte Klasse verwenden, um die alten Server zu steuern.

Mit dem OO-Zugriff auf alte Prozedursammlungen erreicht man mit einem Wrapper, dass keine direkten Änderungen in den kritischen Kernfunktionen auf dem Mainframe oder dem Server nötig sind.

Implementierung

Schon von Wrap gehört? Wrap wird im Zusammenhang mit einem Zeilenumbruch gebraucht. Die Funktion `WrapText` z. B. verteilt einen String über mehrere Zeilen, wenn die Länge des Strings sich einer angegebenen Größe nähert. Der Text lässt sich also einhüllen. Nach diesem Intermezzo folgt ein File-Wrapper.

Im Falle TRadeRoboter erweitern wir die Funktionalität in der Business-Klasse um einen instanzierbaren OO-Zugriff auf den prozeduralen File Import/Export. Dieser steht stellvertretend für ein Legacy-System, welches seit Jahren zum Einsatz kommt. Es sind dies die bekannten Ein- und Ausgaberoutinen wie `ReSet`, `AssignFile` oder `ReWrite`, die nun eine objektorientierte Hülle erhalten.

Das Vorgehen eines Umhüllers ist etwa wie folgt:

- Die Routinen werden gesammelt und in eine Klasse gelegt.
- Die Routinen erhalten die Methodenschnittstelle der Klasse.
- Variablen der Routinen lassen sich in Properties auslagern.
- Ereignisse sind bei Komponenten wenn nötig hinzuzufügen.
- Der Konstruktor wird entsprechend parametrisiert.

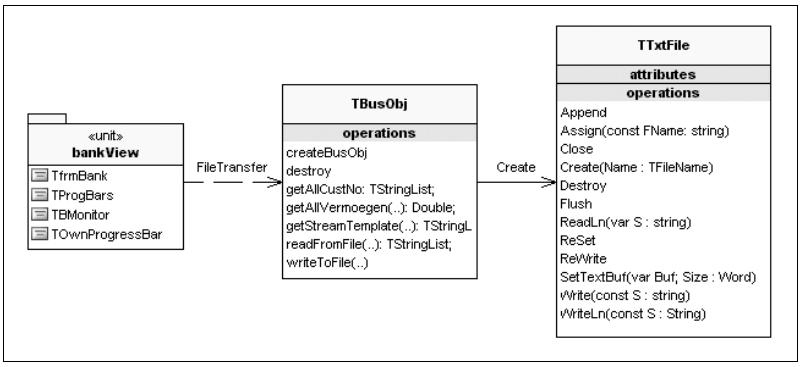


Abb. 2.44: Der Wrapper umhüllt den Im- und Export

Die dynamisch erzeugte Instanz ist genau für einen File Export oder Import der Chartdaten oder sonstigen Finanzdaten zuständig. Wrapper bieten natürlich dort einen Vorteil, wo lose, verteilte Prozeduren und Funktionen durch ein Kapseln in Klassen übersichtlicher und einfacher im Aufruf werden:

```
procedure TDataModule1.writeToFile(kList: TStringList);
var
  q: TTxtFile;
  i: Integer;
begin
  q:= TTxtFile.Create(FILENAME);
  q.ReWrite;
  try
    if kList.count > 0 then begin
      for i:= 0 to kList.count -1 do
      q.Writeln(kList.strings[i]);
    end;
  finally
    q.CloseFile;
    q.Free;
  end;
end;
```

Wegen Namenskonflikten ersetzt `CloseFile` die alte Prozedur `Close`. Verwenden Sie anstelle von `Close` die Prozedur `CloseFile`, um die Zuordnung zwischen einer Dateivariablen und einer externen Datei zu beenden.

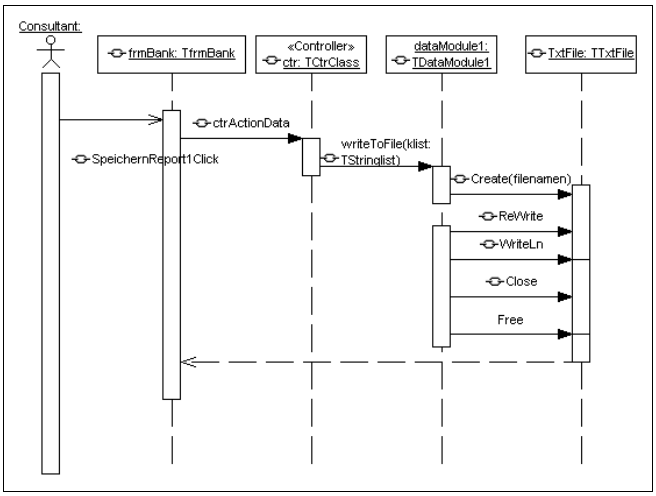


Abb. 2.45: Wrapper umhüllt Standardprozeduren

Im Konstruktor selbst verwendet die Routine den Dateinamen für den internen Gebrauch von `Assign` als exemplarische Altlastenprozedur, die entsprechend umhüllt ist. Diverse Zustände wie `fmClosed` oder `fmOutput` sind als zusätzliche Sicherheit implementiert, die den jeweiligen Dateizugriff wie in einer Statusmaschine überwachen:

```
constructor TTxtFile.Create(Name: TFileName);
begin
  inherited Create;
  TTextRec(FTextFile).Mode:= fmClosed;
  FDefaultExt:= 'trb';
  if Length(Name) > 0 then
     Assign(Name)
end;
```

Mit `Assign()` ist es sogar möglich, eigene Treiber für Textdateien zu bauen. Um die Gerätetreiberfunktionen einer bestimmten Datei zuzuordnen, muss die Prozedur Folgendes erfüllen:

> *Die Prozedur `Assign()`muss den vier Funktionszeigern der Textdateivariablen die Adressen der vier Gerätetreiberfunktionen zuweisen.*

Zusätzlich sollte sie dem Feld `Mode` die Konstante `fmClosed`, dem Feld `BufSize` die Größe des Dateipuffers, dem Feld `BufPtr` einen Zeiger auf den Dateipuffer und dem Feld Name einen Leerstring zuweisen.

Mit dem Namespace `System.AssignFile` zeigt sich die Wrapperklasse:

```
procedure TTxtFile.Assign(FName: string);
begin
  if Mode <> fmClosed then
    raise EInOutError.Create(
       'Cant perform this op. on open text file');
  if (Length(ExtractFileExt(FName)) = 0) and
   (FName[Length(FName)] <> '.') then
   AppendStr(FName, '.'+ DefaultExt);
  System.AssignFile(FTextFile, FName)
end;
```

Verwendung

Die VCL macht auch regen Gebrauch von Wrappern, zumindest dem Sinne nach, wie z. B. `TProgressBar` oder `FdataLink`, die eine Kapselung für eine Fortschrittsanzeige darstellt. `TFieldDataLink` wird als Element in allen datensensitiven Objekten verwendet, die eine Verknüpfung zu einem `TField`-Objekt benötigen und auf Datenereignisse reagieren oder Datenbankinformationen überwachen müssen. Das datensensitive Objekt muss ein Nachkomme von `TWidgetControl` sein.

Ein datensensitiver Kalender kombiniert wie folgt den Wrapper und den Mediator zusammen im Konstruktor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly:= True;
  FDataLink:= TFieldDataLink.Create;
  FDataLink.OnDataChange:= DataChange;
end;
```

Ein ähnliches Muster ist beim Erzeugen eines ActiveX als so genannter ActiveX-Wrapper mit den Assistenten zu beobachten.

Der ActiveX-Experte erzeugt nämlich eine Implementierungs-Unit, die einerseits vom Objekt `TActiveXControl` (für die spezifischen Komponenten von ActiveX-Steuerelementen) und andererseits vom VCL-Objekt des Steuerelements, das eingekapselt werden soll, abstammt.

Der typische ActiveX-Client auf der anderen Seite dient als Host eines visuellen Steuerelements und verwendet es weit gehend wie ein beliebiges Steuerelement auf der Komponentenpalette. ActiveX-Server sind immer In-Process-Server.

Wenn man einen Komponenten-Wrapper für ein Server-Objekt generiert hat, unterscheidet sich die Erstellung einer COM-Anwendung nicht sehr von der Erstellung irgendeiner anderen Anwendung, die VCL-Komponenten enthält. Die Eigenschaften, Methoden und Ereignisse des Server-Objekts sind bereits in der VCL-Komponente gekapselt.

Der Begriff Wrapper wird in der angelsächsischen Literatur dem Adapter Pattern gleichgesetzt, was eigentlich nicht zutrifft. Der Adapter hat eine klare Aufgabe, ein Wrapper ist vom Kontext her vielseitiger.

Im TRadeRoboter sieht die umhüllte Klasse als Verwendung wie folgt aus:

```
TTxtFile = class
  private
    FTextFile: TextFile;
    FDefaultExt: TFileExt;
    function GetEof: Boolean;
    procedure SetActive(state: Boolean);
    procedure SetMode(const NewMode: TFileMode);
  public
    constructor Create(Name: TFileName);
    destructor Destroy; override;
    procedure Append;
    procedure Assign(FName: string);
    procedure Close; virtual;
    procedure Flush;
    procedure ReadLn(var S: string);
    procedure ReSet; virtual;{open file for reading}
.........
```

2.5 Verhaltensmuster

2.5.1 Chain

Ein objektbasiertes Verhaltensmuster (eine Anfrage bearbeiten oder weiterleiten).

Zweck

Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein zuständiges Objekt sie erledigt.

Motivation

Viele Prozessketten basieren auf einem fortlaufenden Weiterleiten von Anfragen, bis sich ein Objekt dafür verantwortlich zeigt und die Anfrage bearbeitet. Soweit die Lehre. Die Grundidee liegt bei der Entkopplung von Sender und Empfänger, indem n-Objekte die Gelegenheit erhalten, eine Anfrage zu bearbeiten. Das Spiel geht so lange entlang der Objektkette (Zuständigkeitskette), bis eines von ihnen darauf reagiert.

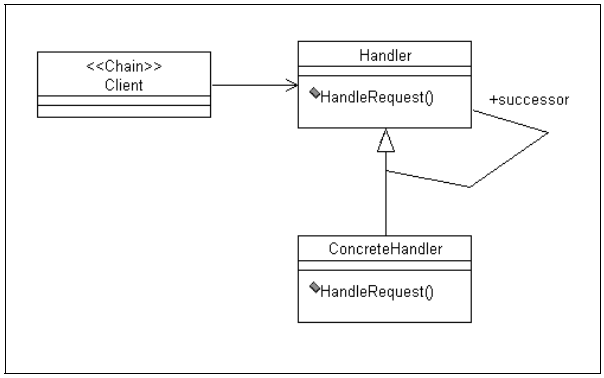


Abb. 2.46: Der Chain leitet weiter

Die Prozessdarstellung einer Unternehmung stellt auch betriebliche Abläufe bis hin zu Geschäftsprozessen mit ereignisgesteuerten Prozessketten (EPK) dar. Innerhalb der ARIS-Architektur[7] wurden die EPK als Beschreibungstechnik von Geschäftsprozessen entwickelt. Die beiden Hauptelemente einer Prozesskette sind die Ereignisse und die Funktionen der Objekte.

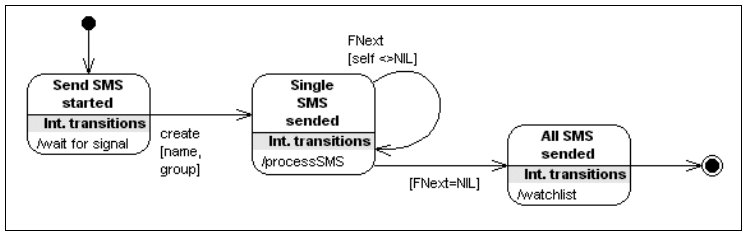


Abb. 2.47: Eine Anfrage wird bearbeitet

Die Lehre einer Zuständigkeitskette besagt nun[v], dass Funktionen Tätigkeiten darstellen, z. B. «versende SMS-Text», und Ereignisse eher Bedingungen oder Anfragen, z. B. «SMS für Gruppe erstellt». Das bedeutet, im „Chain of Responsibility“ beantwortet man ein Ereignis (Anfrage) nicht direkt, sondern kann die Anfrage entlang der Kette an einen oder mehrere Empfänger weiterleiten, bis der Zustand „SMS an Gruppe versendet“ eintritt. Achtung: Mit dem Begriff Empfänger ist die zuständige Methode der Klasse gemeint, nicht der eigentliche Empfänger als Handy-Besitzer.

Zwei Probleme stellen sich nun: Wie verknüpft man die einzelnen Objekte als Nachfolgeobjekte und wie lässt sich der Zuständige finden (das arme Kerlchen ;)), der die Anfrage bearbeitet. Das Muster bietet folgende Lösung:

Beim Erstellen der Objekte lässt sich die Verknüpfung gleich als Referenz mitgeben und der Verantwortliche wird durch ein Kriterium (Parameter) gefunden.

[7] Auch mit Aktivitätsdiagramm möglich.

Implementation

In allen Systemen treten so genannte Anfragen auch als Ereignisse[8] auf, beispielsweise der Ausfall einer Kasse, der Eingang eines Auftrages oder das Erstellen einer Nachricht.

Setzen wir das Pattern in ein praktisches Beispiel im TRadeRoboter als Service um.

Bezogen auf die zunehmende Mobilisierung könnte eine Anforderung an die Middleware lauten: Das Team hat ein SMS-Gateway zu entwickeln (ShortMessageService), mit dem sich aktuelle Nachrichten von Kauf und Verkauf an einzelne Personen als auch an vordefinierte Gruppen mithilfe einer Kettenfunktion über eine Basisstation versenden lassen. Ein Gruppe besteht aus einzelnen Kunden (Closed User Group).

Als Grundlage dient eine Liste von abonnierten Kunden, welche die Kursnachrichten auf dem Mobilphone lesen möchten. Technisch gesprochen ist der Empfänger in der Klasse `TSMS_Box` enthalten, die dank einer Verkettung situativ ermittelt, wer der richtige, menschliche Empfänger ist. Das eigentliche SMS wird dann via Webservice aus der Serveranwendung übermittelt.

Als weitere Grundlage für das Muster dient eine Kette von Empfänger-Objekten, die einen Namen und eine Gruppenzugehörigkeit besitzen. Man kann also ein SMS an eine Einzelperson oder an eine Gruppe senden.

Wenn nun ein SMS in die Runde geht, checkt jedes Empfänger-Objekt, ob es für die Nachricht zuständig ist; wenn nicht, lässt sich innerhalb der verketteten Liste das Nachfolgeobjekt aufrufen, sodass der oder die Zuständige(n) schlussendlich die zugehörige Methode des Versendens aktivieren. In meinem Beispiel ist also mehr als ein Objekt vorhanden, das die Anfrage bearbeiten kann. Damit ergibt sich folgende Anwendbarkeit:

- Bei mehreren Objekten lassen sich die zuständigen Objekte, die reagieren müssen, einzeln zur Laufzeit ermitteln.
- Der Empfänger in der Kette muss nicht explizit bekannt sein.
- Die Menge der Objekte in einer Kette sowie die Verknüpfung mit dem Nachfolger derselben legt man dynamisch fest.

Als Erstes beginne ich mit dem Aufbau der Verkettung der einzelnen Objekte. Für die Adressaten der Personen und Gruppen setzte ich zwei private Stringlisten ein und initialisiere die erste Referenz auf NIL, da noch kein Vorgänger bekannt ist:

```
 FPerson:= TStringlist.create;
 FGroup:= TStringlist.create;
 FNextPers:= NIL;

 readPers('Amadeus Musicus', 'Crypto AG'); //client call

procedure TfrmTrans.readPers(uname, group: TShortStr);
begin
  FPerson.Add(uname);
  FGroup.Add(group);
```

[8] Ereignisse erfolgen dann am Anfang einer Prozesskette.

```
  FNextPers:= createPers(uname, group, FNextPers);
end;
```

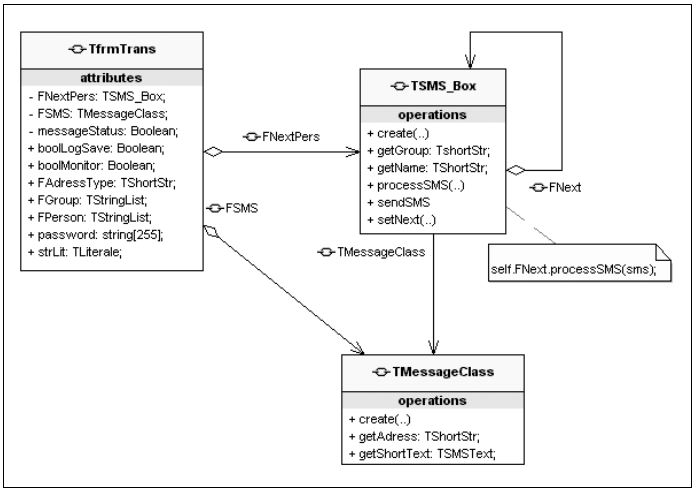


Abb. 2.48: Selbstreferenz beim Chain Pattern

Die Stringlisten dienen nur der grafischen Aufbereitung mit der zugehörigen Auswahlmöglichkeit, wer ein SMS erhält. Mit der Funktion `createPers()` erfolgt nun die einzelne Verknüpfung, indem ich jede einzelne Person und die zugehörige Gruppe dem Objekt `mybox` übergebe.

Zugleich wird jeweils die Referenz des vorherigen Konstruktors als `Next` mitgeliefert. Man kann auch sagen, aus jeder Person und seiner Gruppenzugehörigkeit entsteht ein Objekt vom Typ der `SMS_Box`:

```
function TfrmTrans.createPers(uname, group: TShortStr;
                               Next: TSMS_Box): TSMS_Box;
var
  mybox: TSMS_Box;
begin
  mybox:= TSMS_Box.create(uname, group);
  mybox.setNext(next);
  result:= mybox
end;
```

C#-Code-Beispiel mit return-Funktion:

```
  public TSMS_Box createPers(string uname, string group,
                             TSMS_Box next) {
    TSMS_Box mybox;
    mybox = new TSMS_Box(uname, group);
```

```
    mybox.setNext(next);
    return mybox;
  }
  public string getAddressType() {
    return FAdressType;
  }
```

Somit ist die Verkettung abgeschlossen und jedes Objekt kennt seinen Nachfolger. Das sequenzielle Traversieren beginnt beim letzten Objekt und dauert so lange, bis NIL gefunden wird. Aus dieser Perspektive handelt es sich eigentlich um Vorgänger ;). Anders als die meisten Datenmengen kann eine Zuständigkeitskette den Zeiger nicht direkt auf einen bestimmten Inhalt in der Objektkette positionieren.

Es erfolgt der Aufruf des Clients, welcher die Struktur der Nachricht mit entsprechendem Text und den Adressaten festlegt. Sodann startet er das durchgängige Traversieren durch die zuvor angelegte Objektkette:

```
procedure TfrmTrans.btnSMSClick(Sender: TObject);
var
  aname: TShortStr;
begin
  aname:= lboxChain.Items.Strings[lboxchain.itemindex];
  FSMS:= TMessageClass.create(edtSend.text, aname);
  edtSend.text:=' ';
  FNextPers.processSMS(FSMS, trans);
  FSMS.Free;
end;
```

C#-Code-Beispiel in Konsolenanwendung:

```
  public void startMaxPattern() {
    AppMaxPatterns trans = new AppMaxPatterns();
    trans.getAllPersons();

    TMessageClass FSMS = new TMessageClass("Flügelheim Hans",
                                           "terminal");
    trans.FNextPers.processSMS(FSMS, trans);
    Console.WriteLine("Anzahl " + FSMS.ccount + " SMS
                                            gesendet");
  }
```

Anhand des `ItemIndex` lässt sich feststellen ob eine Einzelperson oder eine Gruppe gemeint ist.

> *Mit `ItemIndex` können Sie einen Eintrag aus der Liste zur Laufzeit auswählen. Man setzt dazu den Wert von `ItemIndex` auf den Index des auszuwählenden Eintrags. In meinem Fall ist der Index im Member `Strings[]` zu finden.*

Das eigentliche Abarbeiten und Suchen nach den zuständigen Objekten ist Aufgabe der Methode `ProcessSMS`, die sich mit dem aktuellen self-Parameter des Nachfolgers durch die Kette angelt. Aufgepasst, es handelt sich um keine Rekursion, da der Member `FNext` ein anderes Objekt mit der Methode `processSMS()` aufruft. Bei einer Rekursion würde sich die Methode ohne `FNext` effektiv selbst aufrufen.

```
procedure TSMS_Box.processSMS(sms: TMessageClass;
                              vtrans: TFrmTrans);
begin
  if vtrans.FAdressType = 'person' then
  if sms.getAdress = self.FUsername then
    vtrans.edtSend.text:=sms.getshortText +' '+ self.FUsername;
  if vtrans.FAdressType = 'group' then
  if sms.getAdress = self.FGroup then begin
    vtrans.edtSend.text:=sms.getshortText +' '+ self.FGroup;
    sms.ccount:= sms.ccount + 1;
  end;
  if FNext <> NIL then
    self.FNext.processSMS(sms, vtrans);
end;
```

C#-Code-Beispiel mit this-Aufruf (self in Delphi)

```
  public void processSMS(TMessageClass sms,
                         AppMaxPatterns trans) {
    if (trans.getAddressType().Equals("person")) {
      if (sms.getAdress().Equals(this.FUsername)) {
        Console.WriteLine(sms.getShortText() + " " +
                          this.FUsername);
        sms.ccount = sms.ccount + 1;
      }
    }
    if (trans.getAddressType().Equals("group")) {
      if (sms.getAdress().Equals(this.FGroup)) {
        Console.WriteLine(sms.getShortText() + " " +
                          this.FGroup);
        sms.ccount = sms.ccount + 1;
      }
    }
    if (FNext != null) {

      this.FNext.processSMS(sms, trans);
    }
  }
```

Das Prüfen der Zuständigkeit ist mit einer zweifachen if-Abfrage geregelt, da bei einer Gruppe mehrere Objekte (Personen) betroffen sind, die auf eine Anfrage reagieren müssen und das Senden des SMS einleiten.

Schön ist auch zu sehen, dass nur reagiert, wer sich angesprochen fühlt, die restlichen Objekte in der Kette werden schlicht und ergreifend übersprungen. Auch im täglichen Leben funktioniert ein Teil des Chain dergestalt, dass ich beim Rufen von „Klaus“ nicht reagiere, erst bei Max zeigt sich dann der Reflex. Indem ich dann in die Runde frage, ob jemand Klaus heißt, kommt man dem Muster mit seiner Taktik schon ziemlich nahe.

Abschließend ein Blick auf das Sequenzdiagramm, welches das zweistufige Vorgehen mit dem Aufbau der Kette und dem nachfolgenden Traversieren nochmals verdeutlicht:

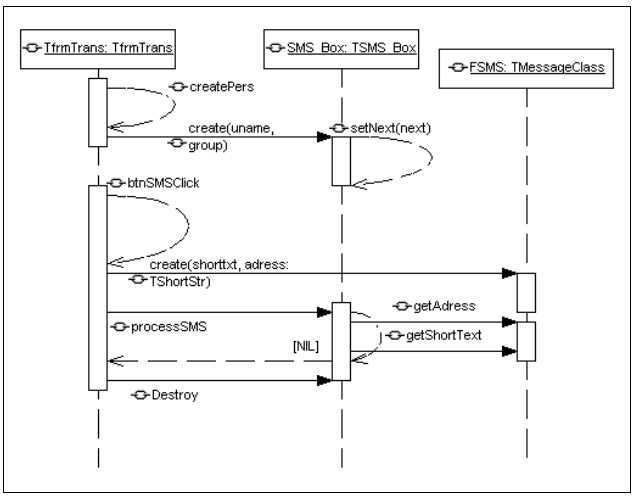


Abb. 2.49: Zweistufiges Chain Pattern

Verwendung

Erwähnt habe ich noch nicht die Möglichkeit, ein SMS durch einen Webservice zu versenden, und dies möchte ich nun nachholen. Dieser Service ist nach einer bestimmten Probezeit kostenpflichtig und er funktioniert aus eigener Erfahrung, je nach Gateway und Land, recht gut, mangelt aber noch an Zuverlässigkeit.

```
procedure TSMS_Box.sendSMS;
var
  soapClient: OLEVariant;
  res: boolean;
begin
  soapClient:= createOleObject('MSSOAP.soapClient');
  soapClient.MSSoapInit('http://sal006.salNetwork.com:83/lucin/
                    SMSmessaging/process.xml');
```

```
  res:=soapClient.sendMessage('4471277',cleanup
               ('test over the air'),'max@kleiner.com','pass');
  if res then
    showmessage('send successfully')
  else
    showmessage('sending failed');
end;
```

Nun, als konkrete Verwendung des Chain Patterns ist sicher das Funktionieren von Exceptions zu erwähnen. Eine Exception löst die Anwendung aus, wenn die normale Programmausführung durch einen Fehler oder ein anderes Ereignis unterbrochen wird.

> *Die Steuerung übergibt die Laufzeitbibliothek an einen Exceptionhandler. Dadurch ist die normale Programmlogik von der Fehlerbehandlung getrennt.*

Da Exceptions Objekte sind, sind sie durch Vererbung in einer Hierarchie organisiert und kennen die entsprechenden Vorgänger. Sie bringen bestimmte Informationen (z. B. eine Fehlermeldung) von der Stelle im Programm, an der man sie auslöst, zu dem Punkt, an dem sie behandelt werden.

Wenn in einer DLL eine Exception erzeugt wird, aber keine Behandlung erfolgt, gibt die Umgebung sie nach außen an den Aufrufer weiter. Wenn man die aufrufende Anwendung oder Bibliothek in Delphi geschrieben hat, ist die Exception in einer normalen `try...except`-Anweisung behandelbar.

Wenn die Anwendung in einer Bibliothek die Unit `SysUtils` nicht verwendet, ist die Unterstützung von Exceptions deaktiviert. Tritt in diesem Fall in der Bibliothek ein Laufzeitfehler auf, wird die aufrufende Anwendung beendet. Folgender Code zeigt die Taktik, wenn derVerantwortliche nicht auffindbar ist, die Routine mit `Exit` zu verlassen:

```
function ForceDirectories(Dir: string): Boolean;
begin
  Result:= True;
  if Length(Dir) = 0 then
    raise Exception.CreateRes(@SCannotCreateDir);
  Dir:= ExcludeTrailingPathDelimiter(Dir);
{$IFDEF MSWINDOWS}
  if (Length(Dir) < 3) or DirectoryExists(Dir)
    or (ExtractFilePath(Dir) = Dir) then Exit;//avoid 'xyz:\
{$ENDIF}
{$IFDEF LINUX}
  if (Length(Dir) = 0) or DirectoryExists(Dir) then Exit;
{$ENDIF}
  Result:= ForceDirectories(ExtractFilePath(Dir))
                              and CreateDir(Dir);
end;
```

Wie weit in einem Chain die Weitergabe einer Exception reicht, ist abhängig davon, wo in der Kette die Exception behandelt wird. Konkret wirft die Laufzeitumgebung eine

Exception, wenn man in der Funktion `Mean()` aus der Bibliothek `math` ein leeres Array übergibt.

Die geworfene Ausnahme `EAccessViolation` ist aber nicht aussagekräftig genug, sie ist zu ungenau, weil in der Kette zu spät, d. h. zu generell, geantwortet wird. Nur wer weiß, dass ein leeres Array mit einem NIL-Pointer repräsentiert wird, kann sich einen Reim darauf machen. Fazit: Je länger eine Exception durch die Kette wandert, desto allgemeiner und ungenauer die Fehleraussage.

2.5.2 Command

Ein objektbasiertes Verhaltensmuster (bestimmt Zeitpunkt und Art der Ausführung).

Zweck

> *Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.*

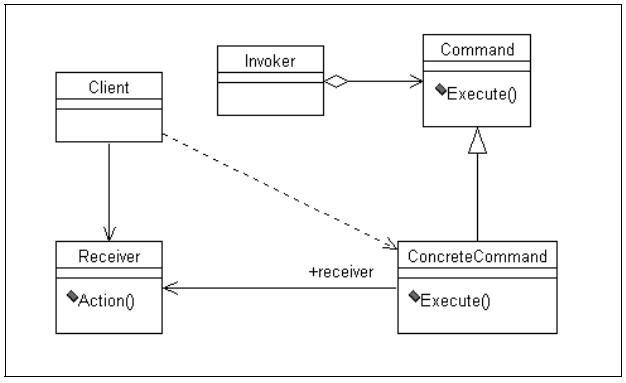


Abb. 2.50: Entkopple Befehl vom Empfänger im Command

Motivation

Es kann nützlich oder mitunter sogar nötig sein, Befehle an Objekte zu stellen, ohne etwas über die auszuführenden Operationen zu wissen.

Aussichtsreiche Kandidaten sind Stratege und Befehl (Command), denn sie kapseln beide die nötigen Verarbeitungen in eigene Klassen. Der Command scheint mir besser zu passen, auch wenn es bei mir nicht um eine GUI mit verschiedenen visuellen Elementen wie Menüs, Icons und Schaltflächen geht, die alle dieselben Befehle aufrufen.

In einer rein prozeduralen Klasse ist die Parametrisierung einer Aktion mit einer Callback-Funktion möglich. Das sind Routinen, die an eine Prozedur oder Funktion überge-

ben und dann von diesem Funktionsrumpf aus aufgerufen werden. `EnumFonts` ist z. B. eine Windows-Routine, welche Callback für jeden im System installierten Font aufruft.

Oder das Streamingsystem einer Komponente ruft `GetChildren` auf, um die Ausführung eines Callback für jede untergeordnete Komponente im Formular zu registrieren. Die Callback-Aufrufe müssen in der Reihenfolge der Erstellung (Reihenfolge der untergeordneten Komponenten in der Formulardatei) erfolgen.

Das Command Pattern lässt sich demzufolge als objektorientierter Ersatz für Callbacks einsetzen. Auch Befehle, die zu unterschiedlichen Zeitpunkten stattfinden, lassen sich mit dem Command spezifizieren und sogar protokollieren.

> *Der konkrete Befehl definiert dann die Anbindung eines Empfängers an eine Aktion, welche ein Aufrufer mit einem vorherigen Befehl auslöst.*

Implementierung

Mithilfe eines Interfaces soll nun eine wieder verwendbare Klasse entstehen, mit der sich eine beliebige Verarbeitung an Dateien und Unterverzeichnissen durchführen lässt, wie ein Durchsuchen nach Mustern in einem Text-File.

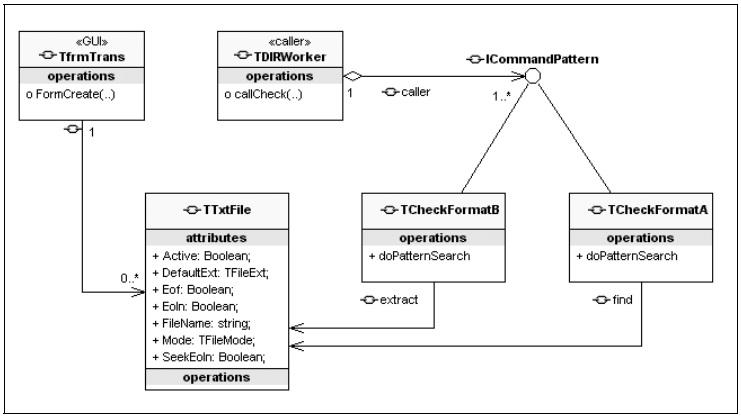


Abb. 2.51: Der Aufrufer übergibt zwei Befehle

Für den Start soll ein kleines Konsolenprogramm entstehen, das Dateinamen inklusive Pfad und Erstellungsdatum aller Dateien in einem Verzeichnis und seinen Unterverzeichnissen ausgibt. Die symbolische Funktion heißt `doPatternSearch`.

Diese wieder verwendbare Klasse des Aufrufers nenne ich `TDirWorker`. In der Minimalvariante besteht sie aus dem Konstruktor, in den das Startverzeichnis und ein Objekt des Typs `ICommandPattern` übergeben werden.

```
procedure TDirWorker.callCheck(myInst: ICommandPattern);
begin
  myInst.doPatternSearch;
```

```
  messageDlg('some_otherWork', mtInformation,[mbok],0);
end;
```

`ICommandPattern` ist nun das Interface, das die Verarbeitung der einzelnen Dateien und Verzeichnisse des Verzeichnisbaums kapselt. Das Interface enthält eine einzige Methode namens `doPatternSearch`. Sobald man allerdings etwas kompliziertere Anwendungen als die vorliegende Miniatur bauen möchte, empfiehlt es sich, noch eine zweite Methode `endFileOperation` zum Interface hinzuzufügen, mit der nach dem Traversieren der Verzeichnisse gewisse Abschlussarbeiten erledigt werden, z. B. das Schließen einer Datenbankverbindung.

```
ICommandPattern = Interface
  procedure doPatternSearch;
end;
```

Als Weiteres benötigen wir eine Klasse, die das Interface implementiert und in der Methode `doPatternSearch` mit Dateien und Verzeichnissen eine Verarbeitung vornimmt. Ich nehme gleich zwei Klassen als Implementierer des konkreten Befehls. Diese Klassen lassen sich später für andere Verarbeitungen mit weiteren Klassen erweitern, die ebenfalls `ICommandPattern` implementieren.

```
procedure TCheckFormatA.doPatternSearch;
var ldbPath: string;
begin
if FileExists(extractFileDir
                (application.exeName)+'\'+ DBNAME) then
   ldbPath:=extractFileDir(application.exeName)+'\'+DBNAME;
   messageDlg('ASearch doing', mtInformation,[mbok],0);
end;
```

Da diese Klasse im Moment nur Dateien und Verzeichnisse nach Mustern durchsuchen kann, nenne ich sie `TCheckFormat`. Die Methode `doPattern()` wiederum ruft den Empfänger der Aktion auf, hier verkürzt als `FileExists`. Damit ist unsere Miniatur fast vollständig, es fehlt einzig noch der Client selbst:

```
procedure TMainFrm.Button1Click(Sender: TObject);
begin
  with(TDirWorker.Create) do begin
    callCheck(TCheckFormatA.create);
    callCheck(TCheckFormatB.create);
    Free;
  end;
end;
```

> *Das Command Pattern entkoppelt also das Objekt, das die Anfrage auslöst, vom Objekt, das die Umsetzung vollbringt.*

Hier gilt die Regel Objekt(Objekt.Aktion), d. h. ein Befehl `TCheckFormat` wird dem Aufrufer `TDirWorker` als Objekt übergeben.

Flexibel zeigt sich das Pattern auch mit der Methode `doPatternSearch`, da beliebige und offene Operationen auf den immer gleichen Dateien oder Verzeichnissen ausführbar sind. Möglich ist dies durch die Parametrisierung mit einem Befehls-Objekt.

C#-Code-Beispiel mit Namespace und Interface (InterSpace ;):

```
using System;
namespace ch.ecotronics.maxbuch.Command {
  //Interface Befehl
  public interface ICommandPattern {
  string doPatternSearch();
  }
//Konkreter Befehl A
  public class TCheckFormatA : ICommandPattern{
    public string doPatternSearch() {
      return "Wir sind in Klasse TCheckFormatA";
    }
  }
  //Konkreter Befehl B
  public class TCheckFormatB : ICommandPattern{
    public string doPatternSearch() {
      return "Wir sind in Klasse TCheckFormatB";
    }
  }

  //Aufrufer
  public class TDirWorker {
    public string callCheck(ICommandPattern iinst) {
      ICommandPattern myint = iinst;
      return myint.doPatternSearch();
    }
  }
}
```

Verwendung

Die fraktale Geometrie erlaubt es, natürliche Formen mathematisch zu beschreiben und chaotische Systeme bildlich zu veranschaulichen. Ein bekanntes Fraktal ist jede Küstenlinie. Betrachtet man einen immer kleineren Ausschnitt der Küste, zeigen sich mit jedem Schritt mehr Buchten und Halbinseln.

Mit jedem Fels oder Sandkorn wiederholen sich dieselben, selbstähnlichen Muster, und die Gesamtlänge wächst gegen Unendlich. Neben der Selbstähnlichkeit haben Fraktale die Eigenschaft der Nichtlinearität: Eine Küstenlinie ist nicht mit den Elementen der

euklidischen Geometrie, der eindimensionalen Geraden oder der zweidimensionalen Fläche, beschreibbar.

Aus der Erfahrung eigener Forschung [chaosmax, c't, 1995] entstand die folgende Anwendung der Berechnung und grafischen Darstellung von Fraktalen, realisiert durch ein Command Pattern. Eine eigentliche `FraktalUnit` ist seit Jahren als Open Source in Umlauf.

> *Die Dimension einer Küstenlinie ist größer als eins, aber kleiner als zwei. Sie entspricht damit einer gebrochenen Zahl zwischen eins und zwei, daher der Name Fraktal.*

Fraktale sind durch die fraktale Dimension vergleichbar. So unterscheiden sich etwa Bäume durch ihre Dimension zwischen zwei und drei. Man kann Fraktale als Untermenge (Subset) der Formengrammatik betrachten, denn es genügt eine rekursiv angewendete Produktregel, um komplexe Gebilde zu generieren. Ein Grundsatz dazu lautet:

> *„Complexity is repeated Application of simple Rules.“* [vi]

Ein bekannter Ansatz ist Mandelbrots Arbeit über die fraktale Geometrie der Natur, in der die Relevanz rekursiv definierter Formen und Kurven für viele Wissenschaftsbereiche deutlich wird (Mandelbrot, 1982). Wie Mandelbrot mit der Beschreibung von Wolken, Bäumen, Küstenlinien und anderen Formen nachweist, sind fast alle natürlichen Formen durch fraktale Algorithmen darstellbar.

Das Klassendiagramm in Abb. 2.52 zeigt die Struktur der Verwendung.

Auch hier nimmt die Klasse `TImageWorker` eine Schlüsselrolle ein, welche den Befehl entgegennimmt und mit weiteren Grundfunktionen umhüllt. Variante `mandel3Click` zeigt den direkten Aufruf ohne die „zwischengeschaltete“ Klasse des Command:

```
procedure Tfrmchaos.mandel3Click(Sender: TObject);
begin
  with IChaosBase(TModelMandelbrot.create) do begin
    setup(frmChaos);
  end
end;
```

Nachteil ist bei erhöhter Grundfunktionalität, wie die Methode `setup` andeutet, keine Wiederverwendbarkeit zu erhalten. Zudem besteht die Gefahr, gemeinsame Logik direkt auf dem Form hinter den Button zu platzieren, also ist folgende Variante die Verbesserung. Wenn der „Worker“ als eigene Klasse mehrere zusätzliche Aufgaben übernimmt, besteht eine Kapselung, die übersichtlicher und besser kontrollierbar ist:

```
procedure Tfrmchaos.totalFractal;
begin
  with (TImageWorker.Create) do begin
    callFractal(TModelLogistic.create, frmChaos);
    callFractal(TModelMandelbrot.create, frmChaos);
```

```
    callFractal(TModelLorenz.create, frmChaos);
  end;
end;
```

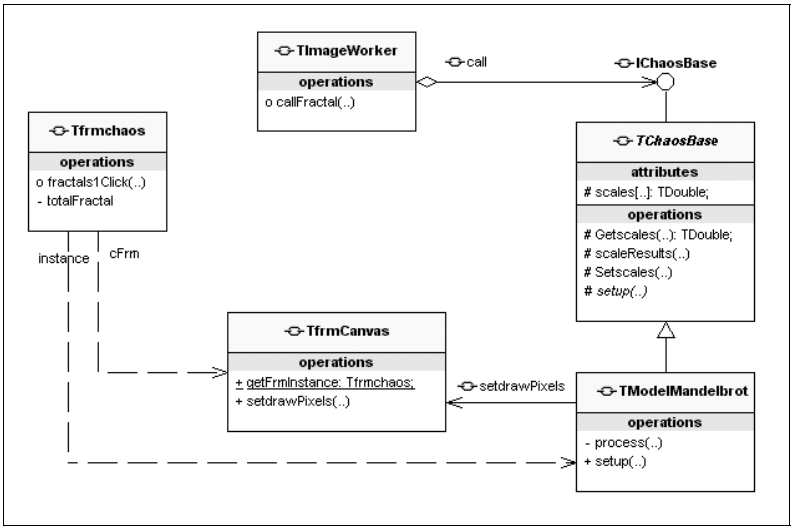


Abb. 2.52: Der Command berechnet und erzeugt Fraktale

Interessant ist der zweite Parameter im Befehlsempfänger von `TImageWorker`, nämlich die Referenz auf ein Form. Diese Referenz dient später dazu, sozusagen als Callback, die Aktion des Zeichnens im Empfänger zu aktivieren.

Der Empfänger ist das Form, welches die Aktion `setDrawPixel()` über den Befehl `setup` erhält. `Setup` wiederum wird vom Aufrufer an das Objekt weitergeleitet. Man darf nicht vergessen, dass im Command Pattern der eigentliche Befehl aus einem erzeugten Objekt besteht! Dieses erzeugte Objekt wird dem Aufrufer `TImageWorker` durch die Methode `callFractal()` übergeben:

```
procedure TImageWorker.callFractal(intf: IChaosBase; frm:TForm);
begin
  intf.setup(frm);
end;
```

Das Spezielle am Befehlsmuster sieht man erst im Sequenzdiagramm: Der Client, hier `TFrmChaos`, erzeugt einen neuen konkreten Befehl. Eigentlich ist es ein Befehls-Objekt, wie ich schon zeigte. Anschließend erzeugt die Applikation den Aufrufer, unser `TimageWorker`, und konfiguriert ihn mit dem Befehl und der Formreferenz, indem das Befehls-Objekt an `callFractal` übergeben wird.

Der Aufrufer `TImageWorker` weiß aber nicht, dass es sich um den konkreten Befehl handelt, für ihn spielt nur eine Rolle, dass er ein Objekt erhält, welches das Interface `IChaosBase` implementiert und deshalb die Methode `setup` zur Verfügung steht.

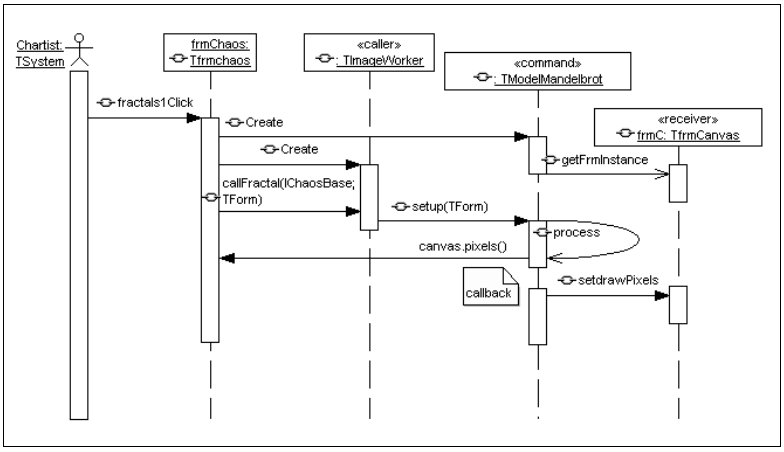


Abb. 2.53: Der Command mit zwei Empfängern

Die Empfänger-Objekte (wie `TfrmCanvas`), die im Originaldiagramm bei GoF noch vorkommen, existieren in anderen Beispielen nicht als unabhängige Klassen. Diese temporären Objekte im konkreten Befehls-Objekt lassen sich dann aber als Empfänger deuten. Es ist auch einfach, neue Befehls-Objekte mit `callFractal()` hinzuzufügen, weil man keine existierenden Klassen ändern muss.

Die abstrakte Klasse `TChaosBase`, die das Interface `IChaosBase` realisiert, wurde nötig, weil die Berechnung der grafischen Skalierung ein gemeinsames Grundverhalten darstellt, welches weder das Interface noch die konkreten Klassen realisieren können.

```
TModelMandelbrot = class (TChaosBase)
private
  procedure process(X,Y, au,bu: double; X2,Y2: integer);
public
  procedure setup(vForm: TForm); override;
end;
```

Als Abschluss einen Eindruck in ein generiertes Fraktal, sozusagen vom Modell zum Code, der mit dem mathematischen Modell dann wieder zu konkreter Grafik wird (siehe Abb. 2.54).

2.5.3 Interpreter

Ein klassenbasiertes Verhaltensmuster (Grammatik und Interpretation einer Sprache)

Zweck

> *Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze und Regeln in der Sprache zu interpretieren.*

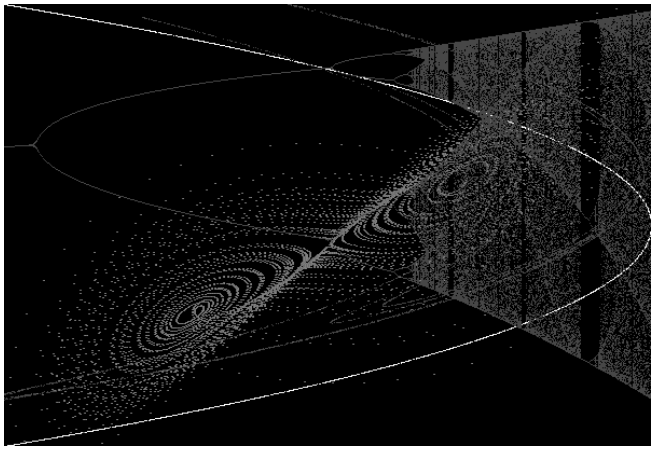


Abb. 2.54: Unendliche Weiten mit der Sternzeit des Commander

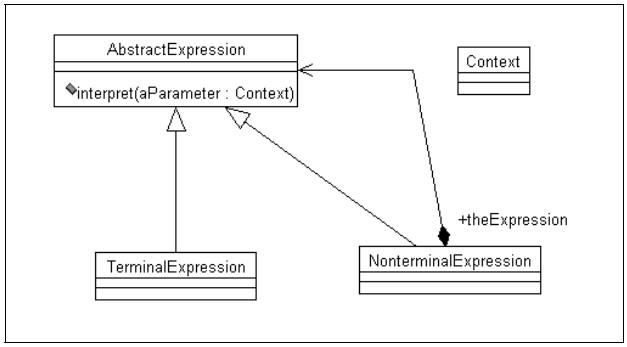


Abb. 2.55: Der Interpreter entschlüsselt den Kontext

Motivation

Wir alle kennen Interpreter aus der täglichen Erfahrung. Dieses Entwicklungssystem arbeitet mit einem Zwischencode, ähnlich dem Bytecode von Java, für den es für alle Computertypen und Rechnerarchitekturen Interpreter geben sollte.

Hilfreich sind Interpreter auch bei der Fehlersuche; Syntax-Fehler sind meistens Schreibfehler auf der Stufe Codieren und Programmieren, z. B. ein "=" anstelle eines ":=". Diese lassen sich schnell mit den Meldungen eines Interpreters korrigieren.

Wenn eine bestimmte Art von Problemen oft genug auftaucht, ist es nahe liegend, das Problem als Regelwerk einer einfachen Sprache zu formulieren (Compilerbau).[vii] Das Interpreter-Muster beschreibt, wie man eine Grammatik für einfache Sprachen definiert und die entsprechende Syntax interpretiert.

Bestehende Werkzeuge wie ein Parser sind eine Alternative zum Interpreter, da ein Parsergenerator ohne den Aufbau eines abstrakten Syntaxbaums auskommen kann und Ausdrücke so interpretiert.

Jeder Interpreter benötigt als Abstrakten Ausdruck einen Syntaxbaum. Kombinationen, die sich aus den grundlegenden syntaktischen Elementen (den so genannten Token) zusammensetzen, ergeben Ausdrücke, Deklarationen und Anweisungen.

Eine Anweisung beschreibt eine algorithmische Aktion, die sich innerhalb eines Programms ausführen lässt. Ein Ausdruck ist eine syntaktische Einheit, die in einer Anweisung enthalten ist und einen Wert beschreibt.

Eine Deklaration definiert einen Bezeichner (z. B. den Namen einer Funktion oder Variablen), der in Ausdrücken und Anweisungen verwendet wird. Sie weist dem Bezeichner bei Bedarf auch Speicherplatz zu.

Das Interpreter-Muster verwendet eine Klasse zur Darstellung der Regeln einer Grammatik. Die **Grammatik** wiederum besteht aus **Ausdrücken** (Expressions), die wiederum aus den kleinsten Elementen, den **Token,** bestehen. Sie sehen, die Geschichte eines Syntaxbaumes ist streng hierarchisch und wird in der Regel durch einen Parser mit einem so genannten rekursiven Abstieg durchschritten.

Im Prinzip ist ein Programm auch eine Folge von Token, die durch Trennzeichen voneinander getrennt sind. Token sind die kleinsten Texteinheiten in einem Programm. Ein Trennzeichen kann entweder ein Leerzeichen oder ein Kommentar sein. Ein Token ist also ein Symbol, ein Bezeichner, ein reserviertes Wort oder eine Direktive.

Ein Ausdruck ist dann eine Konstruktion, die einen Wert zurückliefert. Beispiele:

```
X  { Variable }
@X  { Adresse einer Variable }
15  { Integer-Konstante }
InterestRate  { Variable }
Calc(X,Y)  { Funktionsaufruf }
X * Y  { Produkt von X und Y }
Z/(1 - Z)  { Quotient von Z und (1 - Z) }
C in Range1  { Boolescher Ausdruck }
['a','b','c']  { Menge }
```

Die Grammatik definiert dann die gültige Kombination von regulären oder neuen Ausdrücken, im Folgenden lassen sich `Unit`, `Package` und `Library` definieren:

```
Ziel ->(Programm|Unit|Package|Bibliothek)
Programm ->[PROGRAM Bezeichner ['('Bezeichnerliste')'] ';']
        Programmblock '.'
Unit ->UNIT Bezeichner [Portabilitäts-Direktive] ';'
       interface
       implementation
       initialization'.'
Package ->PACKAGE Bezeichner ';'
```

```
       [requires-Klausel]
       [contains-Klausel]
       END '.'
Bibliothek ->LIBRARY Bezeichner ';'
       Programmblock '.'
```

Im Interpreter sind auch die Begriffe Terminal und nicht Terminal von Bedeutung:

Terminal-Symbole sind nicht weiter durch Wiederholung auflösbar. Die nicht Terminalen lassen sich dem Interpreter weiter übergeben, bis sich ein Terminal finden lässt. Ein Terminal-Symbol ist z. B. true oder false, also eine boolesche Variable. Nicht Terminalsymbole sind Ausdrücke, welche die Operatoren `OR`, `AND` und `NOT` enthalten, so z. B.:

```
b OR land AND High NOT land
```

Ein Parser würde aus dem Symbol BORLAND die folgenden Token herausfinden:

OR, LAN, LAND und AND!

Wenn wir gerade dabei sind, folgt Wettbewerb 2: Wie viele Substrings befinden sich im Wort RESTAURANT; es sind ganze 15, aber welche?

Implementierung

Mit der Implementierung ist das so eine Sache, wenn der letzte Interpreter gerade mal sieben Jahre alt ist, den wir für eine eigene Makrosprache in CALWIN II benötigt haben. Die Applikation hilft durch umfangreiche, flexible Berechnungsmöglichkeiten den Verkehrswert eines Immobilien-Objektes zu finden, indem sich mit verschiedenen „Schätzungsmodellen“ (Templates) die Werte ermitteln und in einem integrierten Dokument schlussendlich darstellen und drucken lassen.

Der Assistent hatte zum Ziel, eine komplette Anweisungsliste (Makroliste) einem Interpreter zu übergeben. Während der Laufzeit war aber unklar, welche Fragen er dem Anwender in welcher Reihenfolge stellen musste. Sein sich ständig modifizierender Weg resultiert aus einer Baumstruktur, die einer Grammatik ähnlich ist.

Viele so genannte Assistenten oder Wizards erstellen eine Makroliste, die sie einem Interpreter zur Ausführung übergeben. Verwenden Sie also den Interpreter, wenn eine Sprache oder eine Makroliste zu interpretieren ist und man diese Ausdrücke als Syntaxbaum oder Knotenstruktur darstellen kann.

Die Knotenstruktur als nicht Terminal-Symbole befinden sich in der Klasse `TAssiTree-Node`, die eine reine Member-Klasse ist. Im Gegensatz zum Original Pattern (siehe Abb. 2.55) befindet sich der Interpreter im Kontext, d. h. in der Klasse `TWizard`. Da der Kontext die für den Interpreter globalen Informationen enthält, wurde auch gleich der Interpreter dorthin implementiert.

Die Liste `LifeWay` ist eine dynamisch aufgebaute Stringliste, welche die jeweils aktuelle Kaskade im Dialog in der richtigen Reihenfolge darstellt und durch eine Variable `Assistep` gesteuert wird. `AssiTree` bildet schlussendlich einen Baum, auch als Stringliste implementiert, der sich aufgrund der Benutzereingaben durch den Assistenten ergibt und dessen Traversieren eben die Liste `LifeWay` darstellt.

```
procedure TAssiFrm.InnerBuild(AssiTreeNode :TAssiTreeNode);
var t:LongInt;
begin
  LifeWay.AddObject
         (Format('%10d',[AssiTreeNode.ID]),AssiTreeNode);
  if AssiTreeNode.Childs<> nil then
  for t:=0 to AssiTreeNode.Childs.Count-1 do
    InnerBuild(TAssiTreeNode(AssiTreeNode.Childs.Items[t]));
end;
```

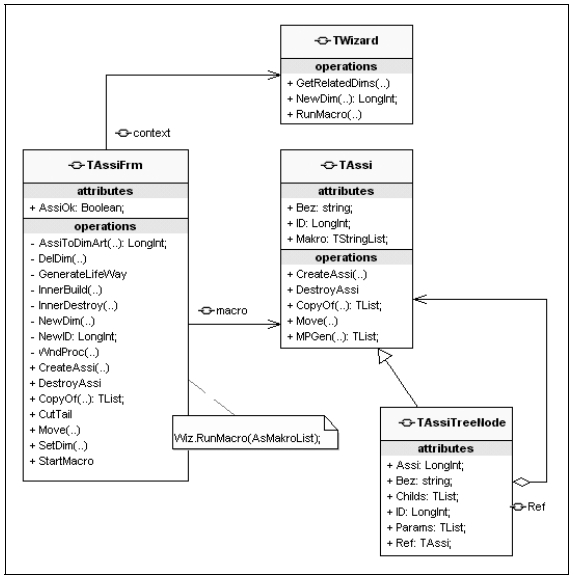


Abb. 2.56: Der Interpreter als Assistent

Der Interpreter und das Kompositions-Muster zeigen viele gemeinsame Implementierungen, wie die Methode `InnerBuild` verdeutlicht. Die Stringliste `LifeWay` lässt sich rekursiv mit den nötigen Kontextinformationen und Befehlen füllen, ähnlich einem Composite, das auch ganze Strukturen in Listen halten kann.

Ich komme zum eigentlichen Interpreter, respektive einem Ausschnitt davon. Die Makroliste ist nun reif, den Interpreter zu füttern. Sie beinhaltet im Stringteil Befehle wie z. B. "`GOTODIM`" und im Objektteil einen Zeiger auf eine Parameterstruktur, die wiederum eine `TList` mit Parametern des Typs `TAssiParams` enthält:

```
.....
if aMacro.Strings[i]= 'GOTOOBJ' then begin
  TempList:=Tlist(aMacro.Objects[i]);
  Seq:=TAssiParams(Templist.items[0]).Seq;
  UCWObj:=GetObjByObjArtAndDimNr(Seq,CurrentDimNr);
```

```
  CurrentObjNr:=UcwObj^.ObjNr;
end else
if aMacro.Strings[i]= 'DELSHEET' then begin
  DelSheet(CurrentObjNr);
  ReLoad3;
end else
if aMacro.Strings[i] = 'INSSHEET' then begin
  TempList:=Tlist(aMacro.Objects[i]);
  Seq:=TAssiParams(Templist.items[0]).Seq;
  Inssheet(CurrentDimNr,Seq);
  Reload3;
.....
```

Zugegeben, der Interpreter ist ähnlich dem Fliegengewicht kein einfacher Geselle, sodass ich keinen einfacheren Interpreter gefunden habe.

> *Die Eigenschaft `Objects` einer Stringliste enthält eine Menge mit Objekten, von denen jedes mit einem String in der Eigenschaft `Strings` verknüpft ist. Mit `Objects` kann man bspw. Bitmap-Objekte mit den Strings der Liste verbinden.*

Das Objekt `TStringList` ist aber nicht der Eigentümer der Objekte des Arrays `Objects`. Objekte, die man dem Array `Objects` hinzufügt, sind auch dann noch vorhanden, wenn das Objekt `TStringList` freigegeben wird. Eine Routine muss die Objekte explizit freigeben.

Bei umfangreichen Syntaxbäumen oder Knotentypen ist es meist hilfreich, eine Strukturliste temporär in eine andere neue Liste zu kopieren, d. h. Zeiger für Zeiger als Typen (die Methode `Assign` kopiert mit `Add` nur bestehende Elemente aus einer vorhandenen Liste in eine andere):

```
function TAssi.CopyOf(L:TList):TList;
var i:LongInt;
    p:TAssiParams;
begin
 Result:=TList.Create;
  for i:=0 to L.Count-1 do begin
    p:=TAssiParams.Create;
    p.DimArt:=TAssiParams(L.Items[i]).DimArt;
    p.Seq:=TAssiParams(L.Items[i]).Seq;
    p.Bz:=TAssiParams(L.Items[i]).Bz;
    Result.Add(TAssiParams(p));
  end;
end;
```

Von CALWIN II zum TRadeRoboter. Im TRadeRoboter selbst kommt ein kleiner Parser zum Einsatz, der ein File so auseinander nimmt (in seine Token zerlegt), dass die einzelnen Zeilen des Files in einer Listview-Komponente einen gezielten Eintrag erhalten.

Als Erstes erfolgt die Differenzierung zwischen einem File oder einem Stream, sodass beim Token des Zeilenvorschubs `#13` eine erste Zeile vorhanden ist. Sukzessive schneidet dann die Routine den String um die Länge einer Zeile ab, sodass jeweils ein weiterer Eintrag im `ListItem` des `ListView` erfolgt:

```
procedure TTRoboParse.scanFile_orStream(myFile: string);
var
  Tempstr, cont: String;
  position: integer;
begin
  with view do begin
   if fileexists(myFile) then
   cont:= openRead(myFile) else
   cont:= myfile;
   position:= pos(chr(13), cont);
   if (position= 0) and (length(trim(cont))> 0) then begin
     ListItem:= Items.Add;
     ListItem.Caption := cont;
     exit
   end;
   while position > 0 do begin
     TempStr:= cont;
     delete(TempStr,position, length(TempStr)-position + 1);
     delete(cont, 1, position);
     ListItem:= Items.Add;
     ListItem.Caption := TempStr;
     position:= pos(chr(13), cont);
   end;
  end;
end;
```

Mit `TListItem` werden die Darstellung und die Datenzuordnungen eines Eintrags in einer Listenansicht festgelegt. Alle Listeneinträge der Listenansicht werden von einem `TListItems`-Objekt gesammelt und an die Eigenschaft `Items` des `TListView`-Steuerelements übergeben. Die geparsten Zeilen sehen nun so aus (siehe Abb. 2.57).

Verwendung

Die meisten Sprachen verwenden mit einem Interpreter den ASCII-Zeichensatz mit den Buchstaben A bis Z und a bis z, den Ziffern 0 bis 9 und weiteren Standardzeichen. Nicht alle Sprachen unterscheiden zwischen Groß- und Kleinschreibung.

> *Das Leerzeichen (ASCII 32) und die Steuerzeichen (ASCII 0 bis 31 einschließlich ASCII 13 für den Zeilenvorschub) werden im Interpreter als Blanks bezeichnet.*

Auch XML darf im Kontext eines Interpreters nicht unerwähnt bleiben. Nachdem ein XML-Dokument von einer DOM-Implementierung ausgewertet oder eben interpretiert

wurde, stehen die von ihm repräsentierten Daten als Hierarchie von Knoten zur Verfügung. Jeder Knoten entspricht einem geparsten Tag-Element im Dokument. Insbesondere behandelt ein DOM-Parser alle Elemente als interne Knoten. Diese Aussage will ich nun anhand des Klassendiagramms in Abb. 2.58 überprüfen.

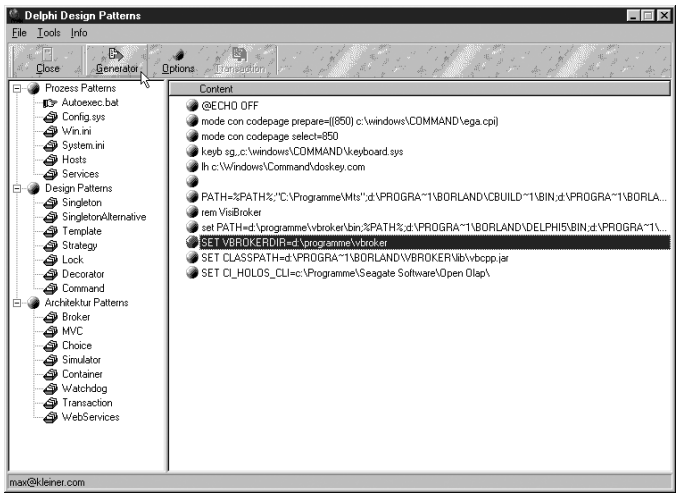


Abb. 2.57: Hier war ein Parser im Einsatz

Das Dokumenten-Objekt-Modell (DOM) ist ein Satz von Standardschnittstellen zur Darstellung eines ausgewerteten XML-Dokuments. Diese Schnittstellen werden von verschiedenen Drittherstellern implementiert. Wenn Sie den Standardhersteller nicht verwenden möchten, können Sie über eine Registrierung weitere DOM-Implementierungen anderer Anbieter in das XML-Grundgerüst integrieren. Die Namen dieser Units enden dann auf "**.xmldom*".

Jeder Knoten als `TMSDOMNode` entspricht also tatsächlich gemäß Abb. 2.58 einem Tag-Element im Dokument. Als Ausgangspunkt dient das folgende XML-Beispiel:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE TRadeRoboter_Stock SYSTEM "sth.dtd">
<TRadeRobo_Stock>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>19.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="ORION">
      <name>Patternskonkret</name>
      <price>99.95</price>
      <symbol>PAK</symbol>
```

```
      <shares type="preferred">25</shares>
   </Stock>
</TRadeRobo_Stock>
```

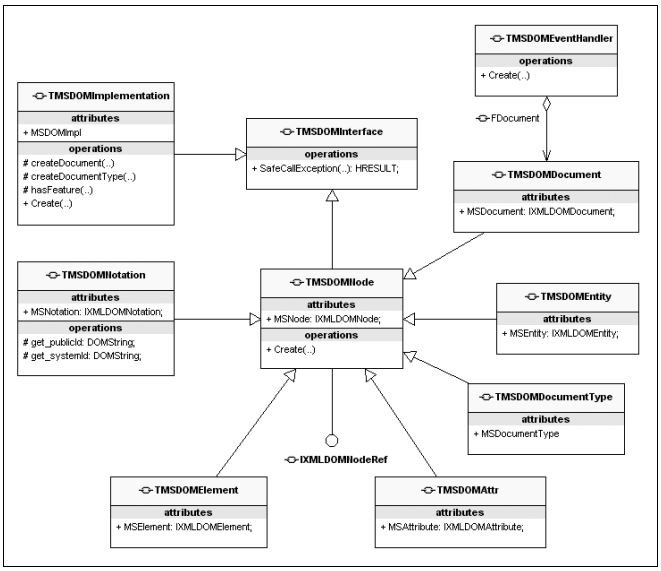


Abb. 2.58: Die Knotenstruktur im Mittelpunkt

Das `TXMLDocument` würde die Hierarchie der Knoten folgendermaßen generieren:

Der Stamm der Hierarchie ist der Knoten `TRadeRobo_Stock`. Dieser Knoten (Node) verfügt über zwei untergeordnete Knoten, die den beiden Stock-Tags entsprechen.

Jeder dieser beiden untergeordneten Knoten verfügt über vier eigene untergeordnete Knoten (<name, price, symbol, shares>). Diese vier Knoten fungieren als Endknoten. Der enthaltene Text erscheint dann als Wert der einzelnen Endknoten.

Mit dem gezeigten XML-Dokument lässt sich dann beispielsweise der Wert der Borland-Aktie folgendermaßen lesen:

```
BorlandStock:= XMLDocument1.DocumentElement.ChildNodes[0];
Price:= BorlandStock.ChildNodes['price'].Text;
```

Ein weiterer „nicht einfacher“ Interpreter in der VCL der Unit *SConnect* ist der `TDataBlockInterpreter`, der vom Design her dem Muster recht nahe kommt. Viele Stream-Verbindungskomponenten benötigen den Interpreter für das Marshaling von Botschaften, die an einen Anwendungsserver gesendet oder von diesem empfangen werden.

Objekte, welche die `IDataBlock`-Schnittstelle unterstützen, speichern die Nachrichten als Kontext selbst. `IDataBlock` ist dann die Schnittstelle, über welche die Objekte zur Interpretation auf den Speicherpuffer zugreifen.

Was interpretiert nun die Methode `InterpretData(data: IDataBlock)` und wozu? Diese Methode interpretiert einen Aufruf, der von einem Anwendungsserver empfangen wird.

> *`TDataBlockInterpreter` übernimmt das Marshaling der COM-Schnittstellenaufrufe für nicht COM-basierte Verbindungskomponenten.*

Der Parameter `Data` bezeichnet eine Schnittstelle, die sich zum Lesen der Botschaft verwenden lässt. `InterpretData` liest den Aufruf, interpretiert ihn als einen `IDispatch`-Aufruf und führt die nötigen Aktionen aus, wie etwa den Aufruf einer lokalen COM-Schnittstelle.

Es muss sich also nicht immer um einen Zeichen-Interpreter handeln, der die nötigen Befehle generiert. Der abstrakte Ausdruck entspricht hier dem `DataBlock`, der mit dem Aufruf zusammen die weiteren Aktionen ausführt.

In Abb. 2.59 ist eine weitere Schnittstelle ersichtlich: Eine vom Typ `ISendDataBlock`, die der Interpreter verwenden kann, um Informationen in einem Datenblock an das Ziel-Objekt zu übermitteln (normalerweise an das entfernte Datenmodul eines Anwendungsservers).

In einen Stream gestellte Verbindungskomponenten senden die von `IDataBlock` erhaltenen Daten an einen Anwendungsserver, nachdem das Objekt zur Datenblockinterpretation seinen Speicher durch das Marshaling gefüllt hat.

Am Schluss dieses Musters, welches tief in ein System eindringen kann und für viele von akademischem Wert sein mag, denke ich doch, dass Interpreter in allgemeiner Anwendung täglich im Hintergrund zum Laufen kommen.

Auch wir interpretieren ja so lange einen Text, bis die Terminal-Symbole erkannt und geordnet werden. Dann füttern wir den geparsten Satz unserem Kleinhirn, welches den intergalaktischen Bewusstseinsgenerator anwirft ;).

Zu empfehlen ist auch die Unit *WebScript.pas*, die einen vollständigen Parser implementiert (Auszug folgt), und wer immer schon mal wissen wollte, wie HTML durch den Parser als Anweisungsfolge geschrieben wird, der ist mit dieser Unit bestens bedient.

```
procedure TScriptFile.ParseStream(ASource: TStream;
                                        AOwned: Boolean);
 function NotBlank(P: PChar; L: Integer): Boolean;
 var I: Integer;
 begin
   for I := 0 to L - 1 do
    if not (P[I] in [' ', #13, #10]) then begin
      Result := True;
      Exit;
    end;
   result:= False;
 end;
```

```
procedure WriteHTML(ASourceStartLine,
                    ASourceEndLine: Integer);
var Item: THTMLItem;
begin
  if NotBlank(PChar(OutStream.Memory),OutStream.Size)
  then begin
   Item := THTMLItem.Create(FProducer.FHTMLItems);
   Item.AssignString(0,PChar(OutStream.Memory),utStream.Size);
   Item.TagToken:= TagToken;
   ScriptSection:= ScriptSection +
     Format('%s.WriteItem(%d)'#13#10,
     [GlobalScriptVariableNames[svHTMLItems],Item.FIndex]);
   OutStream.Size:= 0;
.....
```

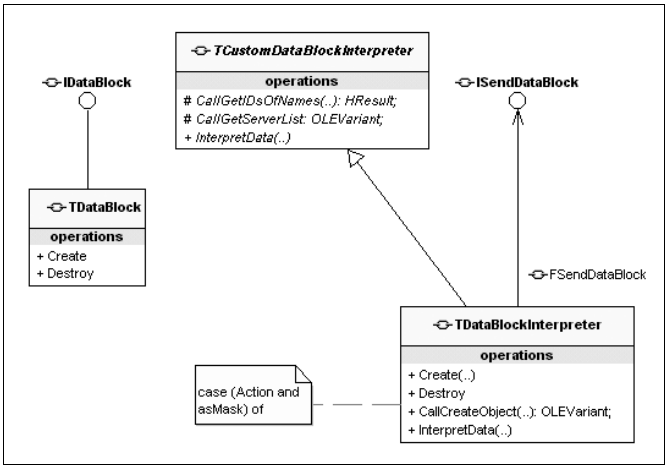


Abb. 2.59: Datenblöcke als Symbole

2.5.4 Iterator

Ein objektbasiertes Verhaltensmuster (Traversieren der Elemente eines Container)

Zweck

Biete eine Möglichkeit, um auf die Elemente eines zusammengesetzten Objekts (Aggregat) sequenziell zugreifen zu können, ohne die zugrunde liegende Repräsentation offen zu legen.

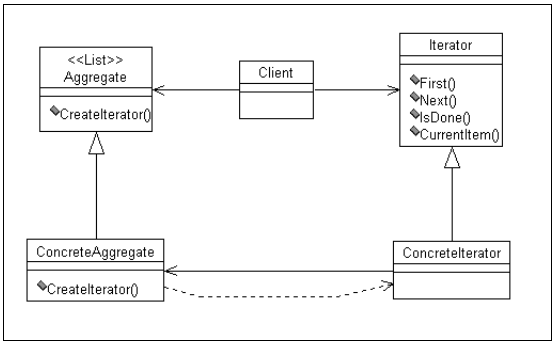


Abb. 2.60: Der Iterator bietet einen Service zum Traversieren

Motivation

Iteratoren sind häufig am Laufen, jedoch ohne eigenen Klassenmechanismus. Die Grundidee des Musters ist, die Verantwortung für den Zuriff und die Funktionalität zum Durchlaufen der Liste aus dem Listen-Objekt zu extrahieren, um ein eigenes Iterator-Objekt einzusetzen.

Meistens will man über eine standardisierte Schnittstelle irgendwelche Container-Objekte (Collections) abarbeiten, ohne die interne Struktur kennen zu müssen. Den Benutzer darf die interne Datenhaltung wie Liste, Tabelle oder Baum nicht interessieren. Soll der Typ der Liste geändert werden (man spricht auch von polymorphen Iteratoren), dann wird ein Iterator mit dem Fabrikmethoden-Muster kombiniert.

Da eine Collection meist einen Iterator benützt, kann die Collection einer Iterator-Factory durchaus entsprechen. Zusätzlich beinhaltet eine Iterator-Instanz, im Gegensatz zum Factory Pattern, noch eine Referenz auf die Datenstruktur, damit die Daten navigierbar bleiben.

> *Iteratoren kommen häufig bei rekursiven Strukturen in einem Composite zum Vorschein, wobei der Schein trügt, da Liste und Iterator eben in derselben Klasse zu Hause sind!*

Implementierung

Darf ich kurz an den eingebauten Iterator beim Composite erinnern. Die Aufgabe besteht darin, die Objekte mit ihren Listen in einen Baum zu transformieren:

```
procedure TDims.FillTree(aOl: TTreeview; xnode: TTreenode);
var
  i: Integer;
  dbcontent: string[255];
begin
  dbcontent:= dimart +' '+ dimartbez;
```

```
  xnode:= aOl.items.addchild(xnode, dbcontent);
  for i:= 0 to treechilds.Count -1 do
     TDims(treechilds.items[i]).FillTree(aOl, xnode);
end;
```

Auch beim Interpreter besteht mit der rekursiven Methode `InnerBuild()`

```
procedure TAssiFrm.InnerBuild(AssiTreeNode: TAssiTreeNode);
```

eine ähnliche Vorgabe, eine Liste in eine nicht binäre Struktur zu zerlegen. Was jedoch beiden fehlt, ist eine eigene Klasse mit Operatoren, um den Zugriff von außen zu steuern. Solche Mechanismen erlauben z. B., die zu traversierende Menge mit Meldungen auszustatten, wie z. B. die Methode `InIterator`, die angibt, ob das Objekt über seine Elemente iteriert.

Im folgenden Beispiel arbeite ich mit drei Klassen. Eine erste Klasse beinhaltet die in einer Liste gehaltenen ersten Nutzdaten der Charts im TRadeRoboter:

```
TMyChart = class(TContextChart)
private
  FPrice: float;
  FName: string;
public
  property Name: string read FName write FName;
  property Price: double read FValue write FValue;
end;
```

Das folgende Klassendiagramm zeigt die Struktur mit dem ausgelagerten Iterator, genannt `TMyListIterator`, der die Liste wie auch die Nutzdaten kennt (siehe Abb. 2.61).

Die beiden anderen Klassen sind die Liste und der Iterator. Die Klasse `TMyList` ist ein typensicherer Wrapper von `TList` und sie veröffentlicht vor allem die Methode `Iterator`, die der Client direkt benutzen kann. Zudem erzeugt `TMyList` den Iterator `TMyListIterator` mit derselben Methode und gibt die Referenz wie folgt zurück:

```
function TMyList.Iterator: TMyListIterator;
begin
  result:= TMyListIterator.Create(self);
end;
```

C#-Code-Beispiel mit this:

```
  public TMyListIterator Iterator() {
    return new TMyListIterator(this);
  }
```

Beim Erzeugen des Iterator zur Laufzeit wird gleichzeitig das Listen-Objekt dem Iterator bekannt gemacht, d. h. mithilfe des Parameters `self/this` übergeben (eine Technik, die auch beim Observer oder Strategy bekannt ist):

```
constructor TMyListIterator.Create(List: TMyList);
begin
  inherited Create;
  FList:= List;
  FIndex:= 0;
end;
```

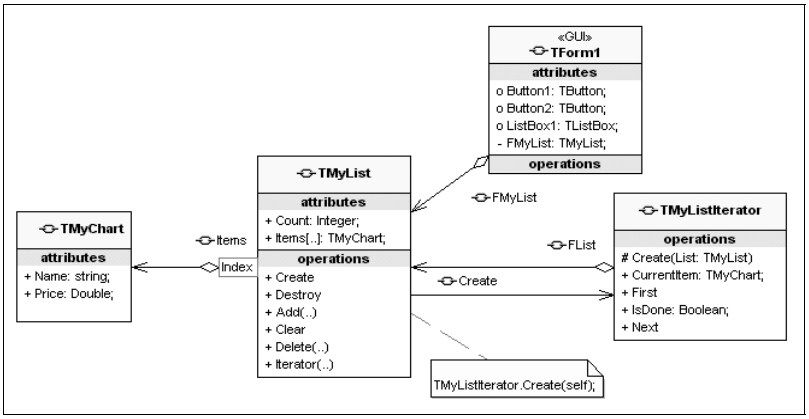


Abb. 2.61: Der Iterator kennt die Liste wie seine Items

C#-Code-Beispiel:

```
  protected internal TMyListIterator(TMyList List) {
    FList = List;
    FIndex = 0;
  }
```

Die Liste selbst ist wie gesagt ein Wrapper um die Klasse `TList`, sodass der Designentscheid Vererbung versus Komposition zugunsten der Objektkomposition ausgefallen ist. Die Konsequenz ist, dass man jede Methode delegieren muss (siehe Teil 1 des Buches), wie das Hinzufügen mit `Add` exemplarisch aufzeigt:

```
procedure TMyList.Add(Item: TMyChart);
begin
  FList.Add(Item);
end;
```

C#-Code-Beispiel:

```
    public void Add(TMyChart Item) {
      FList.Add(Item);
    }
```

Wie aber wird der Iterator aus der Sicht des Clients gebraucht und worin besteht der Unterschied zum herkömmlichen Gebrauch? Ein herkömmlicher Zugriff sieht so aus:

```
begin
  for i:= 0 to item.count-1 do
  item:= FMyList[i];
  Listbox1.Items.Add(item.price);
end;
```

Als problematisch erweist sich, dass wir nicht richtig wissen, was passiert und wo etwas in der Liste passiert, fast wie im täglichen Leben (Leben rekursiv ergibt übrigens Nebel ;). Ein for-loop hat einige Bedeutungen, die sich bei einem direkten Aufruf einer Standardschleife so nicht überwachen lassen.

Fehlermeldungen sind dann wenig aussagekräftig, wenn z. B. die Steuervariable einer for-Anweisung keine einfache Variable ist (sondern beispielsweise eine Komponente eines Datensatzes oder einer Liste) und wenn sie nicht lokal zur Prozedur ist, welche die for-Anweisung enthält.

Mit dem Iterator sind wir auf der richtigen Spur, da eine Kontrolle durch Entkopplung möglich ist und zudem der Iterator ausbaufähig wird:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  iter: TMyListIterator;
  item: TMyChart;
begin
  iter:= FMyList.Iterator;
  while not Iter.IsDone do begin
    item:= iter.CurrentItem;
    Listbox1.Items.Add(item.price);
    Iter.Next;
  end;
end;
```

C#-Code-Beispiel mit Liste:

```
 //Mit Iterator Liste abarbeiten
   iter = FMyList.Iterator();
   while (!iter.IsDone()) {
     item = iter.CurrentItem();
     rtxtbAusgabe.Text = rtxtbAusgabe.Text + item.Name + "\n";
     iter.Next();
   }
   rtxtbAusgabe.Visible = true;
 }
```

Der Client konstruiert zuvor die Liste, die dann durch den Aufruf von `FMyList.Iterator` noch die Instanz des Iterators zurückgibt, d. h., das Objekt `iter` wird in der Methode `Iterator`, die als Funktion daherkommt, selbst erzeugt!

```
  result:= TMyListIterator.Create(self);
```

Interessant ist die eigentliche Methode `CurrentItem` des Iterator, welche den momentanen Eintrag durch das private Feld `FIndex` ermitteln kann und mit `Next` fortlaufend den nächsten Eintrag erhält:

```
function TMyListIterator.CurrentItem: TMyChart;
begin
  Result:= NIL;
  if FIndex < FList.Count then
    Result:= FList[FIndex];
end;
```

C#-Code-Beispiel:

```
  public TMyChart CurrentItem() {
    if (FIndex < FList.Count) {
      return FList.GetItem(FIndex);
    } else {
      return null;
    }
  }
```

So sind Iteratoren einfacher zu gebrauchen, zumal sie für weitere Bedürfnisse vorbereitet sind. Bspw. muss ein Traversieren rückwärts möglich sein oder eine Zählstatisik zum Tragen kommen. Auf den Aufruf des Clients hat dies keinen Einfluss, für ihn ist immer maßgebend, das Ende der Liste mit der Funktion `IsDone` zu erreichen.

Verwendung

Der Iterator wird so oft gebraucht, dass er in allen modernen objektorientierten Sprachen bereits eingebaut ist. Das Muster ermöglicht den sequenziellen Zugriff auf die Elemente einer Liste, wobei der Nutznießer als Client nicht wissen muss, ob sich diese Element innerhalb eines Sets, einer Liste, eines Vektors oder gar einer Map befinden.

Ein Beispiel ist die Implementierung einer Suchmaschine mit der Navigation durch einen Iterator. Eine Suchmaschine muss sich die Ergebnismenge sowie den aktuellen Suchindex mit `CurrentItem` des Benutzers merken.

> *Enumeratoren sind eine Art Iteratoren, die eine Aufzählung (ein Enum-Objekt) verwalten, z. B. ermöglicht das Objekt `Modules` den Zugriff auf alle Module, die instanziert oder aktiviert sind.*

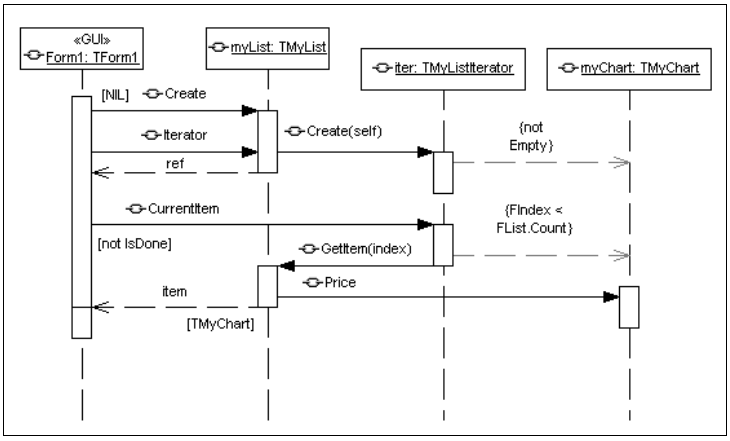


Abb. 2.62: Der Iterator übernimmt die Kontrolle

Im Bereich des WebSnap und der Scriptverwaltung sind einige Iteratoren als eigene Objekte anzutreffen. ServerScripts verwenden die Schnittstelle `IIteratorIndex`, um zu ermitteln, ob sich ein Objekt in einem Enumerationsprozess befindet. Ist das der Fall, wird das Element ermittelt, das derzeit das im Objekt aktive ist. `IIteratorIndex` ist die Schnittstelle, welche die Position eines Objekts angibt, wenn über dessen untergeordnete Komponenten iteriert wird.

Die WebSnap-Anwendung ruft zunächst die Methode `StartIterator` auf, um das Objekt auf das erste Element zu positionieren. Um zum nächsten Element zu gelangen, ruft die Anwendung die Methode `NextIteration` auf. Abschließend ruft die Anwendung die Methode `EndIterator` auf, um das Ende der Iteration zu melden.

Das implementierende Objekt bestimmt die Elemente, über die iteriert wird. Handelt es sich beispielsweise um einen Adapter, können dies Datensätze oder die Seiten sein, mit deren Hilfe der Adapter Informationen gruppiert. Es kann sich auch um Namens-/Wertepaare handeln, die der Adapter repräsentiert. Ist das Objekt eine Liste mit Feldern oder Aktionen, so kann es sich um die Felder oder Aktionen in der Liste handeln.

Es gibt aber auch einfache „prozedurale Monster“ in der Kategorie der Dateiverwaltung, welche die Grundidee des Iterator-Musters aufgreifen.

> *Die beiden Funktionen `FindFirst` mit `FindNext` nämlich. `FindNext` liefert den nächsten Eintrag, der mit dem Dateinamen und den Attributen übereinstimmt, die man zuvor im Aufruf von `FindFirst` angegeben hat.*

Der gesuchte Eintrag muss einer der an `FindFirst` übergebenen Einträge sein.

Ich meine, Iteratoren müssen nicht immer auf Klassenstrukturen basieren.

2.5.5 Lock

Ein objektbasiertes Verhaltensmuster (kontrolliert eine Methode zur Laufzeit).

Zweck

Ermögliche einen Mechanismus, der kurzfristig einige Aspekte des Verhaltens eines Objektes deaktiviert. Blende kontrolliert eine Methode ein oder aus.

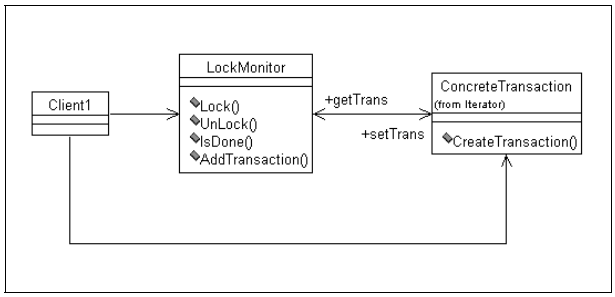


Abb. 2.63: Der Lock kann aktivieren oder deaktivieren

Motivation

Es gibt Situationen, in denen beim Eintreten eines Ereignisses eine bestimmte Methode warten muss, bis das nächste Ereignis eine Entwarnung gibt und die Methode wieder freigibt. In der Regel sind es Methoden, die einen kontinuierlichen Datenstrom wie Messdaten, Transaktionsdaten verarbeiten, oder Ereignisbehandler, die man kurzfristig durch den Lock deaktivieren muss.

Dieses Sperrverhalten ist aber nicht mit der Synchronisation von Threads zu verwechseln, die einen gemeinsamen kritischen Abschnitt betrifft. Wenn man auf Objekte zurückgreift, ist nicht garantiert, dass deren Eigenschaften und Methoden thread-sicher (threadsafe) sind.

Das bedeutet, dass beim Zugriff von außen auf Eigenschaften oder beim Ausführen von Methoden bestimmte Aktionen stattfinden, die auf Speicherbereiche, globale Daten oder auch externe Ressourcen zugreifen, die in dem Moment nicht vor dem Zugriff anderer Threads geschützt sind.

Der Lock-Mechanismus hat eher mit den kritischen Abschnitten zu tun, die mit einem Signal oder einem Zähler sozusagen als Torwächter den Abschnitt schützen.

> *Kritische Abschnitte funktionieren wie Schranken, die immer nur einen einzigen Thread passieren lassen. Für den praktischen Einsatz muss man eine globale Instanz erzeugen.*

Als Konsequenz kann immer nur eine Methode den Abschnitt bearbeiten und vor allem gibt die Methode selbst den Abschnitt wieder frei. Meistens sind Schreibvorgänge in einen Speicherbereich solch kritische Abschnitte.

Da das Lock Pattern keine globalen Variablen im Sinne von Synchronisationsvariablen kennt, kann es auch nicht mit einer anderen Methode kommunizieren, sodass die Hauptaufgabe im Kontrollieren der anderen Methode besteht.

Der Lock-Mechanismus ist in seiner Implementierung einfach, hat aber weit reichende Konsequenzen auf das Zeitverhalten und birgt dann bei wirklich Echtzeit-nahen Systemen so seine Geheimnisse. Das Pattern ist auch unter dem Namen Reference Counter im Einsatz, das eine Variante als Code Pattern darstellt.

Implementierung

Im Falle TRadeRoboter wird der Lock zum temporären Deaktivieren der globalen Transaktionsdaten benötigt, damit die lokalen Transaktionsdaten in der Anzeige des Transaktionsmonitors nicht verloren gehen. Hier die Schritte:

1. Globale Daten erscheinen auf dem Monitor.
2. Ein Ereignis unterbricht den globalen Datenstrom.
3. Globaler Datenstrom wird gesperrt.
4. Lokale Daten erscheinen auf dem Monitor.
5. Globaler Datenstrom wird entsperrt.
6. Globale Daten erscheinen wieder auf dem Monitor.

Die folgende Funktion ist eher als Simulation gedacht und soll nur die Technik der Transaktionsdarstellung mit einem Lock aufzeigen. Ansonsten hätte das Buffern des globalen Datenstroms während des Sperrens sowie die vorgängige Datenkanalisierung das Beispiel verkompliziert.

Beim Eintreffen einer lokalen Buchung wird der globale Datenstrom kurz unterbunden, indem die Methode `setLocking` die Ereignisbehandlung im Sinne des kurzfristigen Sperrens auf NIL setzt:

```
procedure TTransMonitor.setLocking(signal: boolean);
begin
  if signal then application.onMessage:= NIL else
    application.onMessage:= showTransMessage;
end;
```

Lock selber kennt einen Zähler, der wie eine Semaphore den Zustand kontrolliert. Ein Lockmanager kann auch Sperren oder Semaphoren als Variablen implementieren, um einen gleichzeitigen Zugriff auf gemeinsam genutzte Variablen zu verhindern.

> *Ein gleichzeitiger Zugriff auf gemeinsam genutzte Variablen kann u. a. zu Aktualisierungsverlusten oder Dateninkonsistenzen führen.*

Wenn mehrere Methoden dieselben Ressourcen aktualisieren, ist unbedingt ein Synchronisieren erforderlich, das die Konflikte unterbindet. Mit der Methode `setLocking` hat man die Garantie, dass kurzfristig Methoden deaktivierbar sind, die sich sonst gegenseitig in die Ressourcen geschrieben hätten.

```
procedure TTransMonitor.Lock; //locked > 0 , unlock =0
begin
  inc(STLock);
```

```
  if STLock = 1 then setLocking(true);
end;
```

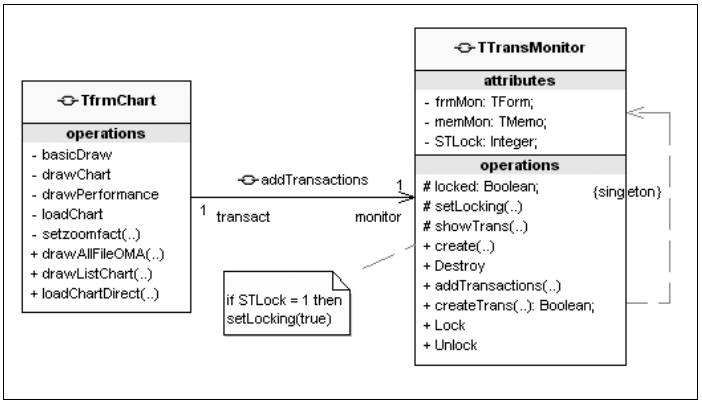


Abb. 2.64: Lock und Unlock in einer Klasse

In der Zwischenzeit kann die Buchung im Transaktionsmonitor zur Anzeige erfolgen und wird somit als Transaktion in den Datenbereich geschrieben:

```
procedure TTransMonitor.addTransactions(amount: string);
begin
 STLock:=0;//unlock zum anfang
 lock;
 try
   with memMon.lines do begin
     add('TR-Buchung: ' + amount);
     add(timeToStr(time)+' / '+ dateToStr(date));
   end;
 finally //garantierte freigabe(deadlock verhindern)
   unlock
 end;
end;
```

Lock Patterns sind auch bei Ressourcenspendern beliebt. Nein, hier werden keine Geldmittel oder Personen freigesetzt. Ein Ressourcenspender verwaltet den nicht dauerhaft gemeinsam benutzten Status bezüglich der Anwendungskomponenten innerhalb eines Prozesses. Der gelegentlich eingesetzte MTS (MS Transaction Server) benötigt so einen Spender. Zum dynamischen Schluss noch das zugehörige SEQ (siehe Abb. 2.65).

Verwendung

Den Lock in Reinkultur, im Sinne der kurzfristigen Deaktivierung einer Methode oder eines Methodenzeigers, habe ich in den Jahren selten entdeckt. Einige Objekte der CLX

verfügen aber über einen integrierten Sperrmechanismus, der die Ausführung anderer Threads oder Methoden verhindert und dem Lock ähnlich ist. Als Verwendung wird demzufolge der TRadeRoboter mit einer Threadsafe-Liste ausgestattet, sodass während dem Schreiben der Transaktion in die Liste kein anderer Thread in die Quere kommt. Bei der Erweiterung sind drei Gedanken eingeflossen:

1. Die Liste ist als Singleton gesetzt.
2. Gleichzeitiges Lesen bietet keine Probleme.
3. Die Liste bietet einen Rollback-Mechanismus.

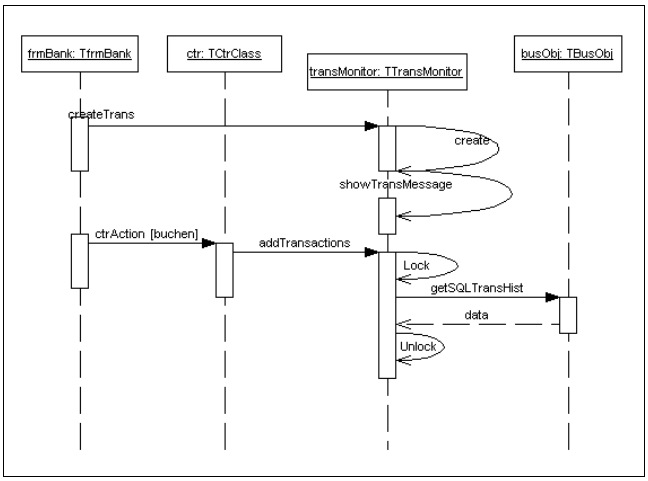


Abb. 2.65: Kontrollierte Transaktion mit dem Lock Pattern

Kollisionen sind also nur möglich, wenn ein Thread einen Schreibzugriff ausführt.

Beispielsweise besitzen auch Zeichenflächen-Objekte (`TCanvas` und seine Nachkommen) eine Methode namens `Lock`, die den Zugriff anderer Threads auf die Zeichenfläche erst dann gestattet, wenn die Methode `Unlock` aufgerufen wird.

Der Sperrmechanismus in der Methode `LockObject` sieht häufig so aus:

```
if Singleton then begin
  if FList.Count < 1 then begin
    Lock;
    try
      if FList.Count < 1 then
        CreateInfo;
    finally
      Unlock;
    end;
  end;
```

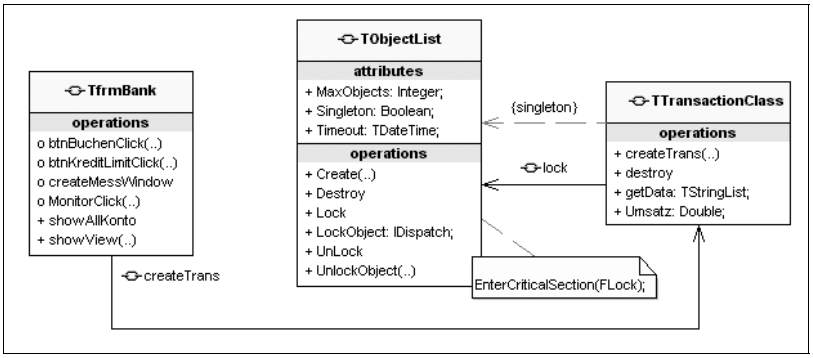


Abb. 2.66: Die Transaktion wird als Objekt geschrieben

Typisch ist auch hier das `Lock` vor dem `Try` und das `Unlock` zur garantierten Freigabe nach dem `finally`. Hier ist `CreateInfo` der kritische Abschnitt.

Sowohl die Objekthierarchie der VCL als auch der CLX enthalten beide das Threadsafe-Listen-Objekt `TThreadList`. Ein Aufruf von `TThreadList.LockList` gibt einerseits das Listen-Objekt zurück und hindert andererseits weitere Threads am Zugriff auf die Liste, bis die Methode `UnlockList` aufgerufen wird.

> *Unter Windows ruft `LockList` die unter Thread-Experten bekannte Funktion `EnterCriticalSection` zum Sperren der Threadsafe-Liste auf. Die Methode gibt das `TList`-Objekt zurück, in dem sich die Liste befindet.*

```
with PObjectInfo(FList[i])^ do
  if not Locked then begin
  Locked:= True;
  LastAccessed:= Now;
  Result := Obj;
  Exit;
end;
```

Unter Linux gibt `LockList` das TList-Objekt, das die Liste enthält, erst dann zurück, nachdem der Mechanismus alle anderen Threads gesperrt hat. Eine letzte Routine zeigt anhand eines Destruktor in Delphi.NET, dass mit dem so genannten „dispose pattern“ auch nicht verwaltete Ressourcen oder Code freigegeben werden können. Man spricht in .NET auch von Ressourcen-Wrappern.

Anstelle des Lock-Mechanismus tritt ein Monitor in Kraft, der auch hier wieder die Routine threadsafe macht. Ich hoffe, Sie haben Nachsicht mit der Häufigkeit von englischen Fachbegriffen, wie eben threadsafe, denn „Programmfaden-sicher“ wäre auch nicht die wahre Übersetzung ;).

```
Destructor BaseResource.destroy
begin
  monitor.Enter(self) //make it thread safe
```

```
  try
    if not disposed then begin
      fileStream.close;
      disposed:= true;
    end;
  finally
    monitor.Exit(self);
  end;
end;
```

In der CLX oder der JVCL (JediVCL)[viii] hat es zwei waschechte Lock-Mechanismen, die mit `TBag` (einer Art `TCollection`) und `TStrings` der Idee sehr nahe kommen.

Beim Hinzufügen von mehreren Elementen will man aus Performancegründen vermeiden, dass jedes Mal ein `Change`-Ereignis erfolgt. In `TBag` sollen Änderungen nur dann kommunizieren, wenn das Hinzufügen von **allen** neuen Elementen abgeschlossen ist, d. h., vor dem Hinzufügen von Elementen wird das Benachrichtigen durch `Change` mit einem Lock gesperrt:

```
procedure TBag.AddItems(Items: TList);
begin
  Lock;
  try
    for I:= 0 to Items.Count - 1 do
      Add(Items[I]); //calls Change
  finally
    Unlock;
  end;
end;

procedure TBag.Change;
begin
  if not Locked then
    if Assigned(FOnChange) then FOnChange(Self);
end;

procedure TBag.SetLocking(Updated: Boolean);
begin
  if Updated then Change //Bag has become unlocked
end;
```

Das Boolean Flag `Updated` bezieht sich auf das **mehrfache** Aktualisieren durch `Change`, nachdem die Liste geändert wurde. Durch die Methode `Lock` lässt sich das Flag am Anfang auf false setzen. Bei `Unlock` wird das Flag wieder true, sodass man jetzt mit `Change` eine Änderung, d. h. ein Update, kommuniziert.

Hingegen beim Hinzufügen eines **einzelnen** Elements muss `Change` jedes Mal feuern. Das Geniale am Lock ist die Möglichkeit einer Verschachtelung, die mit einfachen Flags nicht möglich wäre:

```
procedure TBag.Add(Item: Pointer);
begin
  ...{Add Item to internal structure}
  Change;
end
```

Eine weitere und letzte Anwendung oder Entdeckung ist in der Klasse `TStrings` anzutreffen, die auch zwischen dem Beginn und Ende einer Änderung in der Liste die Ereignisbehandlung deaktiviert und somit Konsistenz und Performance optimiert:

```
procedure TStrings.BeginUpdate;
begin
  if FUpdateCount = 0 then SetUpdateState(True);
  Inc(FUpdateCount);
end;

procedure TStrings.EndUpdate;
begin
  Dec(FUpdateCount);
  if FUpdateCount = 0 then SetUpdateState(False);
end;

procedure TStringList.SetUpdateState(Updating: Boolean);
begin
  if Updating then Changing else Changed;
end;
```

2.5.6 Mediator

Ein objektbasiertes Verhaltensmuster (Zusammenspiel von Objekten sowie die Art der Zusammenarbeit).

Zweck

> *Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt und steuert. Mediatoren fördern lose Koppelung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen, das Zusammenspiel der Objekte an einer zentralen Stelle zu variieren.*

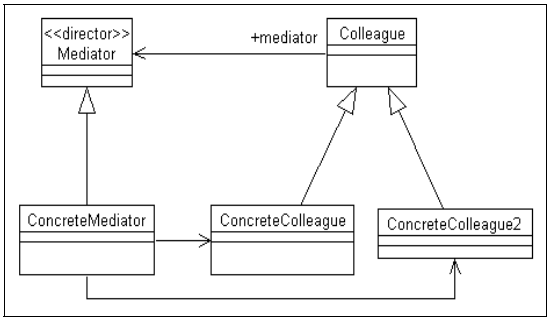


Abb. 2.67: Der Mediator vermittelt zwischen Kollegen

Motivation

Vermittler haben auch im täglichen Leben immer eine Berechtigung. Nur wird im täglichen Leben nicht immer zwischen Kollegen vermittelt, wie ein Anwalt zeigt. Eigentlich ist jeder „Object Broker“ ein Vermittler, auch eine Anwendung wie der MTS (Microsoft Transaction Server). Auch er vermittelt Anfragen zwischen Kollegen.

Wenn Sie den MTS einsetzen, kann man eigene MTS-Transaktionen in den Anwendungsserver integrieren, um die Transaktionsunterstützung zu erweitern. MTS-Transaktionen können sich über mehrere Datenbanken erstrecken oder Funktionen enthalten, die sich nicht auf Datenbanken beziehen.

Wenn das Verhalten eines Subsystems über viele Objekte verstreut ist, ist es schwierig, Funktionalität zu ändern oder gezielt anzupassen.

> *Es gibt einen Zielkonflikt, der besagt, das Verteilen von Funktionen zwischen Objekten zu fördern, mit dem Nachteil, dass dann zu viele Objekte die anderen kennen müssen.*

Für die Wiederverwendbarkeit ist Unterteilung (Dekomposition) einer Anwendung in Objekte ein Vorteil, bei zu vielen Verbindungen zwischen den Objekten wird das Wieder verwenden zum Problem. Man kann dem Widersprechen und meinen: „Das ist kein Zielkonflikt: Die Kunst des Design besteht darin, möglichst schlanke Objekte zu machen und deren gegenseitige Referenzen trotzdem zu minimieren!“ Angenommen man hat in sieben Objekte unterteilt und jedes kennt im Extremfall das andere, ergibt $((n^2 - n) / 2)$ Verbindungen, d. h. 21.

> *Der Mediator bietet eine Lösung gegen diesen Wildwuchs an, indem zwar die lose Kopplung möglich ist, aber die Objekte einem Vermittler (oft auch Direktor genannt) unterstellt sind, der zwischen Kollegen vermittelt.*

Als Direktor ist er Vermittler zwischen der Anfrage des Clients und den Kollegen, sodass die Kollegen nur indirekt zusammenarbeiten können. Die Kollegen untereinander müssen sich nicht kennen. Manstelle sich eine Telefonzentrale ohne Punkt-zu-Punkt-Verbindung vor, die ein Konferenzgespräch zwischen Fremden vermittelt.

Eine gewisse Ähnlichkeit besteht zum Facade Pattern, wobei der Mediator ein multidirektionales Protokoll hat, im Gegensatz zum Facade, das die Anfragen nur unidirektional an sein Subsystem richtet. Facade und Observer lassen sich zudem kombinieren, indem die Kollegen denVermittler mithilfe des Observer benachrichtigen.

Implementierung

Im Falle TRadeRoboter ist es der `TSMSController`, der die Koordinationsarbeit übernimmt. Er kapselt nicht nur die drei Kollegen Port, Gate und Status, er verbindet auch die Ereignisse mit den entsprechenden Eventhandlern. Eine ähnliche Konstellation ist beim Bridge Pattern im Einsatz, wobei Struktur und Verhalten unterschiedlich sind. Im Original der GoF sind es visuelle Controls, die das Muster erzeugt und anschließend untereinander koordiniert.

In meinem Beispiel erzeugt der Mediator die zu koordinierenden Kollegen als nicht visuelle Klassen, die Interaktion erfolgt dann mithilfe von Ereignissen, die von Änderungen ausgelöst werden. Die Grundidee ist folgende:

Beim Versenden eines SMS entsteht ein Aufruf an den Controller, der, sobald Daten am SMSPort zum Senden bereitstehen, mithilfe von asynchronen Ereignissen die restlichen Klassen inklusive SMSPort koordiniert. Der Controller als Mediator vermittelt zwischen dem Port (Objektinstanz `FSMSPort`) und dem Gate (Objektinstanz `gate`), er initialisiert Datenstrukturen und informiert die Statusklasse nach erfolgtem Versenden des SMS.

> *Somit müssen Port, Gate oder Status unter sich keine Nachrichten mit gegenseitigen Referenzen austauschen, die ganze Interaktivität steuert der Controller.*

Dies kann er nur, weil jedes Ereignis eines Kollegen (Port, Gate etc.) auf eine Methode im Controller zeigt. Ereignisse sind eben Methodenzeiger. Bevor man Ereignisse behandeln kann, gilt es, Ereignisempfänger zu definieren. Als Erstes verbindet der Controller mit der Methode `WireSMSPort` die jeweiligen Ereignisse der Kollegen mit seinen Methoden. Innerhalb der Methoden erfolgt dann die eigentliche Koordinationsarbeit.

```
constructor TSMSController.create;
begin
  inherited Create;
  FSMSPort:= TSMSPort.Create;
  WireSMSPort;
end;
```

C#-Code-Beispiel:

```
    public TSMSController() {
      FSMSPort = new TSMSPort();
      this.wireSMSPort();
    }
```

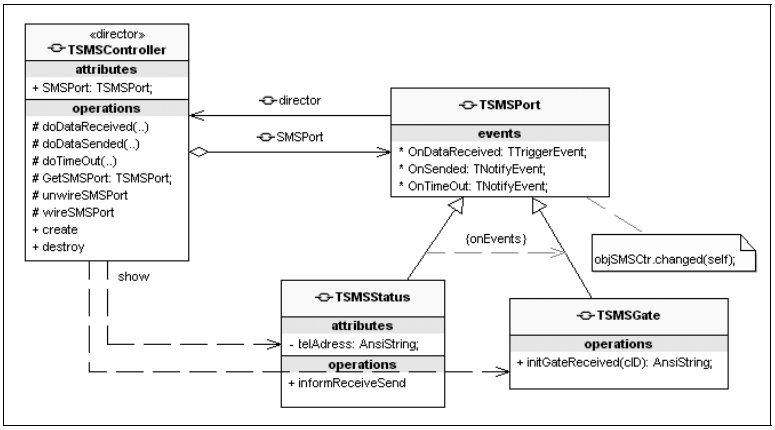


Abb. 2.68: Mediator koordiniert zwischen Port, Status und Gate

Bei dieser interaktiven Schwerarbeit stellt sich die Frage, ob die Klasse nicht überfrachtet ist. Eigentlich nicht, da noch keine zeitkritischen Elemente vorhanden sind, zumal die Ereignisse teilweise asynchron sind. Ich komme zur Verbindung:

```
procedure TSMSController.wireSMSPort;
begin
  FSMSPort.OnDataReceived:= doDataReceived;
  FSMSPort.OnTimeOut:= doTimeOut;
  FSMSPort.OnSended:= doDataSended;
end;
```

C#-Code-Beispiel mit Eventhandlern (verbindet die Ereignisse der Kollegen mit den Eventhandlern des Controllers):

```
  protected internal void wireSMSPort() {
    FSMSPort.OnDataReceived += new
                        dataReceivedDelegate(doDataReceived);
    FSMSPort.OnSended += new sendedDelegate(doDataSended);
    FSMSPort.OnTimeOut += new timeOutDelegate(doTimeOut);
  }
```

Vermittler fördern grundsätzlich die lose Kopplung, indem sie die Objekte abhalten, sich direkt miteinander zu verbinden oder aufeinander Bezug zu nehmen. Ein Vermittler kann sich auch auf fremde Dienste innerhalb seiner Methoden berufen, um an Ergebnisse zu gelangen, die für die Koordination wichtig sind.

> *Anstelle eines Ereignisempfängers könnte man, wie beim Observer, die Kollegen auch als Abonnenten im Controller eintragen, wie ein Subscriber-Objekt. Bei Änderungen ist es dann der Controller, der sie koordiniert benachrichtigt.*

Wenn nun ein `OnDataReceived` Ereignis z. B. durch einen Trigger verursacht eintritt, wird geprüft, ob die verbundene Methode existiert und diese dann entsprechend aufgerufen. Das Ereignis kann über den Controller zum Port, aber auch direkt beim Port eintreffen:

```
procedure TSMSPort.doDataReceived(Sender: TObject;
                                  trigger: TTrigger);
begin
  if Assigned(FOnDataReceived) then
                    FOnDataReceived(Sender, trigger);
end;
```

C#-Code-Beispiel:

```
  protected internal void doDataReceived(object Sender) {
    this.OnDataReceived(Sender);
  }
```

Es folgt mit dem Methodenzeiger die eigentliche Koordinationsarbeit. Der Aufruf `doDataReceived` aktiviert die Zusammenarbeit der Kollegen in effizienter Weise.

Falls Sie einen neuen Dienst wie SMS für eine Anwendung bereitstellen wollen, die mit anderen Systemen kommuniziert, besteht der erste Arbeitsschritt in der Entwicklung eines Kommunikationsprotokolls, das die Server und Clients des Dienstes benutzen. Folgende Fragen sind dabei relevant:

- Welche Botschaften oder Ereignisse lassen sich senden?
- Auf welche Weise müssen diese Botschaften koordiniert werden?
- Wie ist die Information kodiert?

Hierzu wurde eine DLL in Delphi gebaut,[ix] die einen Zugangspunkt am Festnetz für die Aufgabe von SMS-Meldungen zur Verfügung stellt. Viele GSM-Betreiber bieten via Telefonleitung diesen Dienst an, wobei es noch zu viele unterschiedliche Zugangsprotokolle gibt. Die größte Verbreitung hat das TAP-Protokoll gefunden.

Auf der anderen Seite lässt sich ein SMS auch über ein Web-Interface, wie in meinem Fall ein Webservice (siehe auch Chain Pattern), versenden.

```
procedure TSMSController.doDataReceived(Sender: TObject;
                                        trigger: TTrigger);
begin
  gate.initGateReceived(actcID);
  gate.FFilePath.Assign(trigger);
  smsState.informReceiveSend(strlit[23]
                       +' '+format('%p',[pointer(sender)]));
end;
```

C#-Code-Beispiel:

```
protected internal void doDataReceived(object Sender) {
  //Dummy-Methoden
  strDummy = gate.initGateReceived(Sender.ToString()) + \n\n";
  strDummy = strDummy +
     smsState.informReceiveSend(Sender.ToString());
}
```

Das kleine Protokoll vom Client (`TSMS_Box`) zum Server (`TSMSPort`) erweitert die Signatur des normalen `TNotifyEvent` mit einem neuen Ereignis:

```
TTriggerEvent= procedure (Sender:TObject; Trigger:TTrigger)
                                                  of object;
```

Sobald man über eine Instanz des Ereignisempfängers verfügt, in meinem Fall ist es der Methodenzeiger, der auf die Behandlungsroutine `doDataReceived` zeigt, lässt sich auch jederzeit die Verbindung der Ereignisse mit dem Empfänger wieder unterbrechen.

```
procedure TSMSController.unwireSMSPort;
begin
  FSMSPort.OnDataReceived:= NIL;
  FSMSPort.OnTimeOut:= NIL;
  FSMSPort.OnSended:= NIL;
end;
```

C#-Code-Beispiel:

```
  protected internal void unwireSMSPort() {
    FSMSPort.OnDataReceived -= new
                        dataReceivedDelegate(doDataReceived);
    FSMSPort.OnSended -= new sendedDelegate(doDataSended);
    FSMSPort.OnTimeOut -= new timeOutDelegate(doTimeOut);
  }
```

Lassen wir das Ganze mit dem Sequenzdiagramm nochmals Revue passieren.

1. Der Direktor als Vermittler erzeugt den SMSPort und verbindet die Ereignisse.
2. Ein Ereignis, ausgelöst durch erhaltene Textdaten, tritt beim Port ein.
3. Der Port meldet die Änderung seines Zustands dem Controller.
4. Der Controller koordiniert das Initialisieren und Senden zwischen Gate und Status.
5. Der Controller deaktiviert Verbindungen mit NIL und zerstört den Port.

Innerhalb der Klasse `TSMSPort` existieren exakt zwei Zustände, die mit zwei Ereignissen in eine Wechselbeziehung treten. Der Port ist entweder bereit, auf Ereignisse zu reagieren (<ready switched>) oder hat ein Ereignis behandelt, d. h. weitergeleitet (<handled>). Die Zustandsübergänge sind durch die jeweiligen Ereignisse markiert (siehe Abb. 2.70).

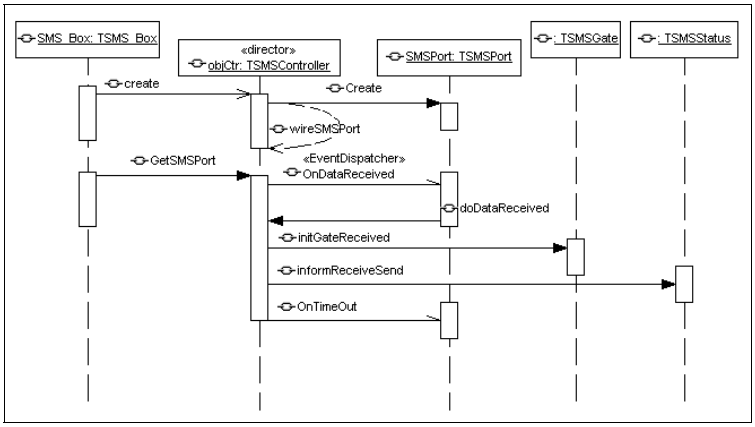


Abb. 2.69: In der Sequenz zeigt sich die Koordination

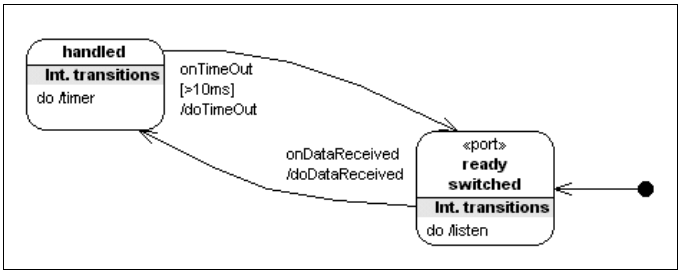


Abb. 2.70: Im SE wird das Event an den Eventhandler weitergeleitet

Es gibt drei grundlegende Vorteile des Mediator:

- Der Mediator vermittelt das Gesamtverhalten durch Weiterleiten von Anfragen, die mit der Methode `WireComPort` verbunden wurden.
- Der Mediator (meist als Singleton) zentralisiert die Steuerung und lässt sich als ideales Kapseln von Protokollen einsetzen.
- Dadurch dass die einzelnen Kollegen-Objekte keine direkte Beziehung untereinander aufweisen, ist Wiederverwendung einfacher.

Verwendung

Fast jede Komponentenarchitektur benutzt das Mediator-Muster, indem die Komponente das Ereignis an einen Vermittler weitergibt. Der Vermittler ist konkret in Form einer Formklasse anzutreffen. Die Klasse `TForm` übernimmt das Verbinden von Events zu den Eventhandlern, weil andernfalls die Kontrollelemente wie `TButton`, `TEdit` oder `Tcaption` einen Wildwuchs von direkten Verbindungen untereinander erzeugen müssten.

Bei der Komponentenentwicklung können Sie den Methodenzeiger als Platzhalter verwenden: Sobald der Quelltext feststellt, dass ein Ereignis eintritt, wird die Methode aufgerufen (falls vorhanden), die vom Benutzer für dieses Ereignis vorgesehen wurde.

Man spricht in .NET auch von Delegation, was exakt einem Methodenzeiger entspricht:

> *„MS.NETTechNet: Delegation between different classes is done by method pointers, variables that point to methods.“*

Ein Komponentenereignis weist man einer Behandlungsroutine zu, so erfolgt die Zuweisung nicht an eine Methode mit einem bestimmten Namen, sondern an eine Methode in einer bestimmten Klasseninstanz. Dieses Objekt ist normalerweise das Formular, das die Komponente enthält (was aber nicht der Fall sein muss).

Diese Zuweisung geschieht ausschließlich zur Laufzeit.

```
TControl = class(TComponent)
private
  FOnClick: TNotifyEvent; //method pointer
  ...
protected
  property OnClick; TNotifyEvent read FOnClick write FOnClick;
end;
```

C#-Code-Beispiel mit der Deklaration `delegate`:

```
  public delegate void dataReceivedDelegate(object sender);
  public delegate void timeOutDelegate(object sender);
  public delegate void sendedDelegate(object sender);
```

Der Begriff OLE DB kennzeichnet im Weiteren als Verwendung z. B. eine OLE-Schnittstelle, die verschiedenen Anwendungen einen standardisierten Zugriff auf Daten aus unterschiedlichen Quellen bietet. Genau dies kann die Aufgabe eines Mediator sein.

> *Über eine einheitliche Schnittstelle soll der Entwickler Zugriff auf Datenquellen aller Art erhalten. OLE DB orientiert sich an den elementaren Funktionen einer Datenquelle. Weil die Komponenten nicht direkt an die Datenquelle gekoppelt sind, sollen sie sich über verschiedene Plattformen und Anwendungen einheitlich verteilen lassen.*

Anwendungen (wie dbGo) verwenden Microsoft ActiveX Data Objects (ADO) 2.1 zur Interaktion mit einem OLE DB-Provider, der eine Verbindung zu einem Datenspeicher einrichtet und auf die darin gespeicherten Daten zugreift. Der momentane Zustand ist, die diversen Schnittstellen (HTTP, ODBC, OLE, MAPI etc.) einzeln kennen und unterstützen zu müssen. So greift man auf eine genormte Stelle zu, von der aus OLE DB die weiteren Knoten ansteuert und koordiniert.

> *Stichwort Adressen: Wie oft hat man schon irgendwelche Adressen wieder neu angelegt oder unterschiedlich mutiert, weil die Interoperabilität und Austauschbarkeit im Programm fehlte.*

COM Server Exceptions lassen sich auch als Mediator realisieren, indem der Mediator die Exceptions in einer Systemereignis-Logdatei protokollieren und in weiteren Aufrufen koordinieren muss. Die Kollegen, die man koordinieren könnte:

- Exceptions in eine Textdatei speichern
- E-Mails an einen Administrator senden
- Technische Mitarbeiter über einen Pager informieren

Die Eigenschaft `ServerExceptionHandler` als Mediator eignet sich zur Verwaltung von Remote-Servern, insbesondere dann, wenn ein Server physikalisch nicht erreichbar ist. Wenn beispielsweise in einem COM-Objekt eine Exception ausgelöst wird und die Serveranwendung eine formularbasierte Anwendung (nicht einfach eine DLL) ist, kann die Standard-Ereignisbehandlungsroutine ein Meldungsfenster auf dem Server anzeigen.

Bevor dieses Meldungsfenster nicht geschlossen wird, können keine Client-Anforderungen auf dieser COM-Objektinstanz bedient werden! Indem die Exception in einem Ereignisprotokoll festgehalten wird, ist diese Information von allen Remote-Anwendern einsehbar, sofern diese Zugriffsrechte für den Server besitzen.

2.5.7 Memento

Ein objektbasiertes Verhaltensmuster (Zeitpunkt des Speicherns außerhalb des Objektes).

Zweck

> *Erfasse und externalisiere den internen Zustand eines Objekts, ohne sein Kapseln zu verletzen, sodass das Objekt später in diesen Zustand zurückversetzt werden kann.*

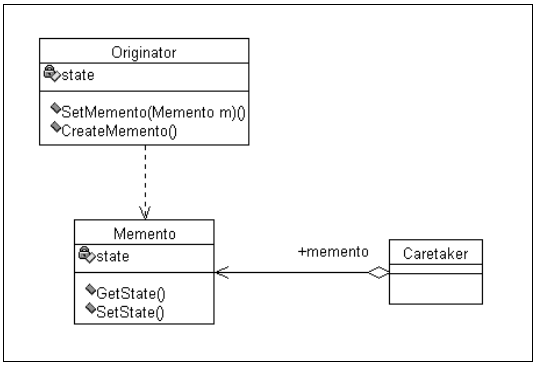


Abb. 2.71: Das Memento speichert Zustände

Motivation

Sicher hatten Sie auch schon den Wunsch, Zustände rückgängig zu machen. Mit `Undo` können Sie bekanntlich die letzte Änderung einer Eigenschaft verwerfen. Die Anzahl der Änderungen, die man rückgängig machen kann, lässt sich durch die Eigenschaft `UndoLevels` ideal festlegen.

Mit einem `CanUndo` sollte man feststellen, ob der Benutzer Änderungen vorgenommen hat, die das Memento durch einen Aufruf der Methode `Undo` rückgängig machen kann.

Auch in der Datenbanktechnik ist Undo ein Muss. Mit einem so genannten `SavePoint` ist ein Bearbeitungsstatus speicherbar und lässt sich später wiederherstellen. `SavePoint` ist ein Integer, der den aktuellen Status des Änderungsprotokolls angibt. Sollen die Änderungen im Änderungsprotokoll auf einen früheren Status zurückgesetzt werden, weist man `SavePoint` wieder den Wert zu, der galt, bevor die Änderungen vorgenommen wurden.

> *Nun hat man es in der OO-Welt mit Objekten zu tun, die ihren Zustand selten nach außen transportieren, sodass sich mehrstufiges Speichern in einem externen Objekt als Problem erweist. Die Kapselung darf nicht gefährdet sein.*

Die Lösung ist eine Trennung zwischen dem Urheber (Originator) und der Zustandsspeicherung (Memento). Das Memento ist ein Objekt, das einen Snapshot des internen Zustands eines anderen Objekts (Originator) speichert. Zudem benötigen beide Objekte eine Hilfsklasse zur temporären Speicherung des Zustands, Caretaker (Aufbewahrer) genannt.

Das Memento-Objekt besitzt effektiv zwei Schnittstellen, es kennt die Daten des Originators und kennt den Caretaker zwecks Speicherung der Daten. Achtung! Hier ist Wettbewerb 3: Was hat „myosotis“ mit Memento zu tun?

Implementierung

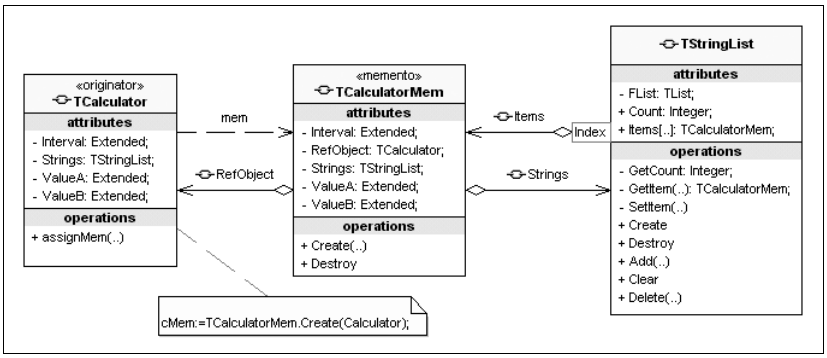


Abb. 2.72: Das Memento als Erinnerungshilfe

Die Idee stammt von Jochen Fromm, der mit der nötigen Hilfsklasse `TStrings` im Grunde ein Kopieren des Zustands von A nach B und umgekehrt implementiert. Das Kopieren lässt sich konkret mit `Assign` realisieren. Das Erfassen und externalisieren des

internen Zustands sowie das Zurücksetzen erfolgt im Beispiel direkt über ein Konstruktor/Destruktor-Paar, im Original der GoF als `SetState` und `GetState` notiert.

Ich beginne mit dem Speichern des Zustandes, der drei einfache Werte beinhaltet:

```
  cMem:= TCalculatorMem.Create(Calculator);
```

Nachdem das Memento erzeugt ist, kann man den internen Zustand des `TCalculator` mit `Assign`, von einer zur anderen String-Klasse, zuweisen. Originator und Memento bedingen, den Zugriff auf die privaten Felder als „friend“ offen zu legen:

```
constructor TCalculatorMem.Create(Calculator: TCalculator);
begin
  inherited Create;
  refObject:= Calculator;
  ValueA:= refObject.ValueA;
  ValueB:= refObject.ValueB;
  Interval:= refObject.Interval;
  Strings:= TStringList.Create;
  Strings.Assign(refObject.Strings);
end;
```

Die einzelnen Werte sind im Memento gespeichert sowie als Gesamtes mit `Assign` abgebildet. Dies ermöglicht, auch nur inkrementelle Änderungen zu speichern. Dies bedingt eine strikte Reihenfolge der Undo-Schritte. Es ist prinzipiell ein Unterschied, grafische Objekte mit Positionen oder interne Werte ohne Visualisierung zu speichern.

Im nächsten Schritt wird eine in der Zwischenzeit erfolgte Änderung wieder rückgängig gemacht, indem der Destruktor zum Zuge kommt. Die Routine gibt also das Memento an den Urheber `TCalculator` zurück:

```
destructor TCalculatorMem.Destroy;
begin
  refObject.ValueA:= ValueA;
  refObject.ValueB:= ValueB;
  refObject.Interval:= Interval;
  refObject.Strings.Assign(Strings); //go back
  Strings.Free;
  inherited Destroy;
end;
```

Im folgenden Sequenzdiagramm zeigt sich, dass der Caretaker als String-Klasse entweder als eigene Klasse typisiert oder als Hilfsklasse im Sinne einer Abhängigkeit modelliert ist. Da im Original der Caretaker das Memento erzeugt und das Memento wiederum einen Caretaker als Container benötigt, empfiehlt sich eine eigene Typisierung des Caretakers. Im Beispiel fehlt diese Typisierung, die Klasse `TStrings` ist demzufolge in mehreren Exemplaren als Hilfsklasse im Einsatz:

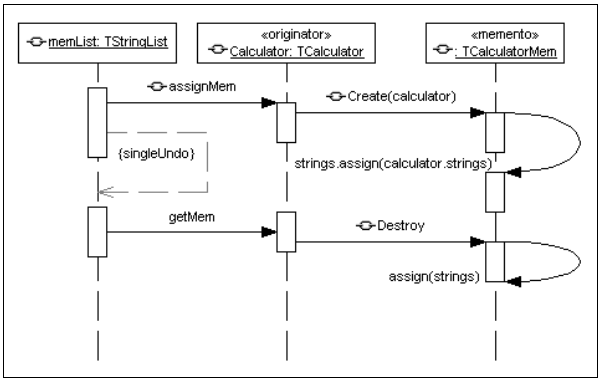


Abb. 2.73: Zustand setzen und zurücknehmen im Memento

Verwendung

Als Verwendung in der CLX ist die Klasse `TRecall` als Memento im Einsatz, die intensiv von der Technik von `TPersistent` und natürlich von `Assign` Gebrauch macht:

```
TRecall = class (TObject)
  private
    FReference: TPersistent;
    FStorage: TPersistent;
  public
    constructor Create(AStorage, AReference: TPersistent);
    destructor Destroy; override;
    procedure Forget;
    procedure Store;
    property Reference: TPersistent read FReference;
  end;
```

`TPersistent` kapselt das fundamentale Verhalten, das allen Objekten gemeinsam ist, die anderen Objekten zugewiesen werden können und die Eigenschaftswerte in eine Formulardatei (.xfm- oder .dfm-Datei) schreiben bzw. daraus lesen können.

> *`TPersistent` ist also der Vorfahr aller Objekte, die über Zuweisungs- und Stream-Funktionen verfügen.*

Es ist nun interessant herauszufinden, wie sich der `TRecall` einsetzen lässt. Diese Klassen als Originator (`TFont`, `TBrush` und `TPen`) gibt es tatsächlich, zudem ist `TRecall` auch mit neuen Klassen, die eine Art `Undo` benötigen, der erste Kandidat.

Das Speichern des Zustandes erfolgt mit der Methode `Store` (siehe Abb. 2.74) und hat folgende effiziente Implementierung. (Die Funktion `Undo` wie ein `Recall` vermisse ich seit meiner Geburt im täglichen Leben ;)

```
procedure TRecall.Store;
begin
  if Assigned(FReference) then
    FStorage.Assign(FReference);
end;
```

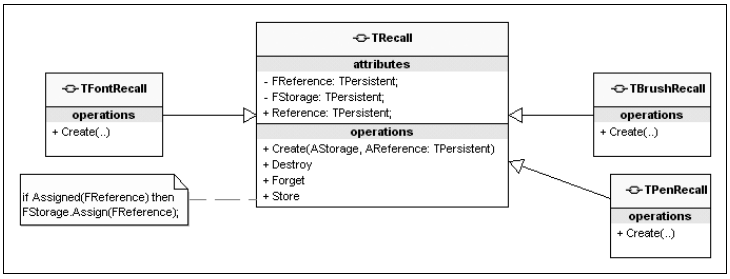


Abb. 2.74: Das Memento im richtigen Leben der CLX

Mit `Assign` kopiere ich die Eigenschaften und andere Attribute eines Objekts aus einem anderen. Der Zuweisungsoperator bewirkt, dass Destination dasselbe Objekt wie Source referenziert, wobei die Methode den Inhalt des in Source referenzierten Objekts in das von Destination referenzierte Objekt kopiert. Da `FStorage` die Destination darstellt, entspricht dies exakt dem Caretaker als Container.

In den meisten Anwendungen wird `Assign` überschrieben, um die Zuweisung von Eigenschaften aus ähnlichen Objekten vorzunehmen. So auch im Beispiel, das zum Schließen des Memento-Kreises noch den Aufruf in der konkreten Verwendung zeigt:

```
constructor TCalculatorRecall.Create(ACalculator: TCalculator);
begin
  inherited Create(TCalculator.Create, ACalculator);
end;
```

2.5.8 Observer

Ein objektbasiertes Verhaltensmuster (Objekte registrieren und flexibel aktualisieren).

Zweck

Definiere eine Abhängigkeit von 1:n zwischen Objekten, sodass bei einer Zustandsänderung alle registrierten Objekte benachrichtigt werden und sich dann automatisch oder zu gegebener Zeit aktualisieren können.

Motivation

Mit den Patterns bricht der Observer in eine neue Dimension auf, da er unter den Pattern-Anhängern als Star und Liebling zugleich gilt. Seine Mechanismen sind raffiniert und besitzen einen hohen Stellenwert, auch bei IT-Architektur und Telematik.

Wiederverwendbarkeit stellt sich leider nicht von selbst ein. Klassen sollten demzufolge eine maximale Unabhängigkeit aufweisen. Vor allem unter GUI-Klassen oder Controls sind zwei Designgrundsätze bekannt:

- möglichst dezentrale Ereignisbehandlung
- möglichst lokale Datenhaltung

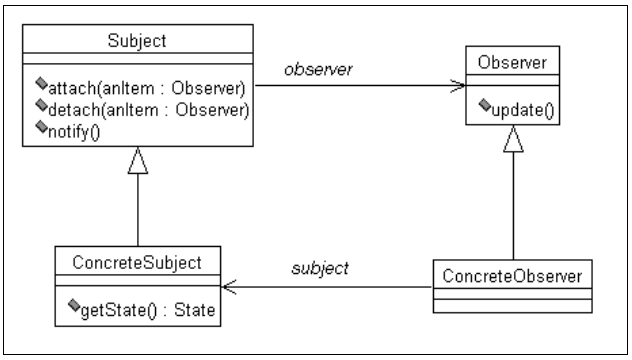


Abb. 2.75: Der Observer als Pattern-König

In der Literatur findet man meist nur den zentralen Ansatz, die Ereignisse in einer großen Case-Schleife als Fallunterscheidung in einem Container zu verarbeiten. Demgegenüber steht die dezentrale Verarbeitung der Ereignisse in den Komponenten.

Stellen Sie sich das Klassendiagramm der Kleinbank „KleinerBank“ in Abb. 2.76 vor, die ständig damit konfrontiert ist, eine Änderung der Konten in vielen abhängigen GUI-Controls aktualisieren zu müssen. Folgende Änderungen des Kontostandes sollten quasi gleichzeitig grafisch und protokollarisch zum Vorschein kommen:

- Barbezug beim Automaten innerhalb der `TCashMachine`
- Buchen eines Betrages über ein zugehöriges Konto
- Eröffnen eines Kontos mit zugehörigem Kunden `TCustomer`

GUI-Objekte, die also die ankommenden Ereignisse autonom verarbeiten, müssen somit auf die Daten, die sie grafisch anzeigen, auch lokal zugreifen können. Die Idee, diese Daten in den entsprechenden lokalen Instanzenvariablen zu verwalten, scheitert an der gegenseitigen Abhängigkeit bei der Aktualisierung zu anderen Objekten!

> *Wenn also eine Liste anhand eines selektierten Elements andere Objekte benachrichtigen muss, ist die Liste selbst nicht mehr autonom, da sie die anderen Objekte zwangsläufig kennen und einbinden muss.*

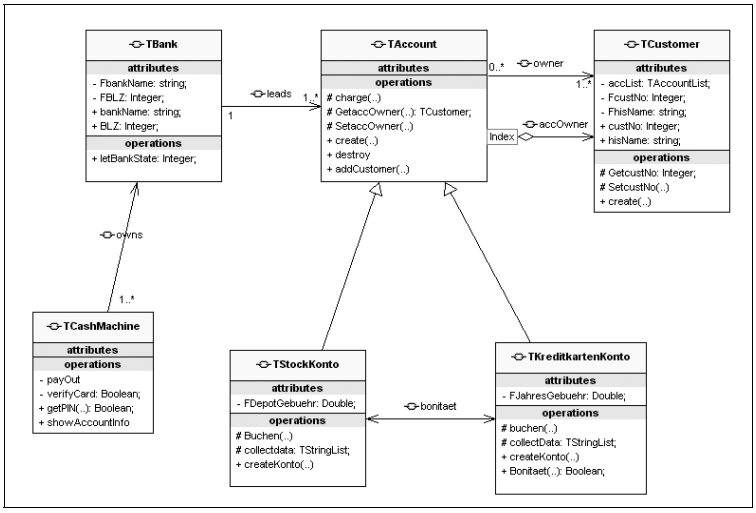


Abb. 2.76: Das Problem der gemeinsamen Aktualisierung

Und autonom heißt auch wieder verwendbar. Dieses Dilemma zwischen der möglichst lokalen Datenhaltung und dem Bestreben, Objekte autonom zu entwerfen, lässt sich durch das faszinierende Observer Pattern lösen.

Das Observer-Muster beschreibt eine 1:n Abhängigkeit zwischen einem Subjekt (Observable) und den vielen Observern, die man bei einer Daten- oder Zustandsänderung durch das Subjekt (auch Model) benachrichtigen und aktualisieren muss.

Im Zusammenhang mit der MVC-Architektur spielt der Observer nochmals eine zentrale Rolle. Diese Art der Interaktion ist auch als Publish Subscribe (publiziere und abonniere) bekannt, indem das Subjekt die Nachrichten publiziert, nachdem die Beobachter sie abonniert haben. Konkret will ich nun Schritt für Schritt dieses zentrale Muster aufbauen.

Implementierung

Zuerst wird eine abgeleitete Klasse gebaut, die als Subjekt die einzelnen Objekte, die benachrichtigt werden wollen, registrieren kann. Jedes Objekt (View) lässt sich also in dieser Basisklasse `TSubjectCenter` einzeln registrieren:

```
TSubjectCenter = class(TObservable)
  private
    FObservers: TList;
    FAmount: double;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Add(Observer: TObserver); override;
    procedure Remove(Observer: TObserver); override;
    procedure Notify; override;
```

```
    procedure allReset; override;
    //Daten als Modell
    function getAmount: double;
    procedure changeAmount(value: double);
  end;
```

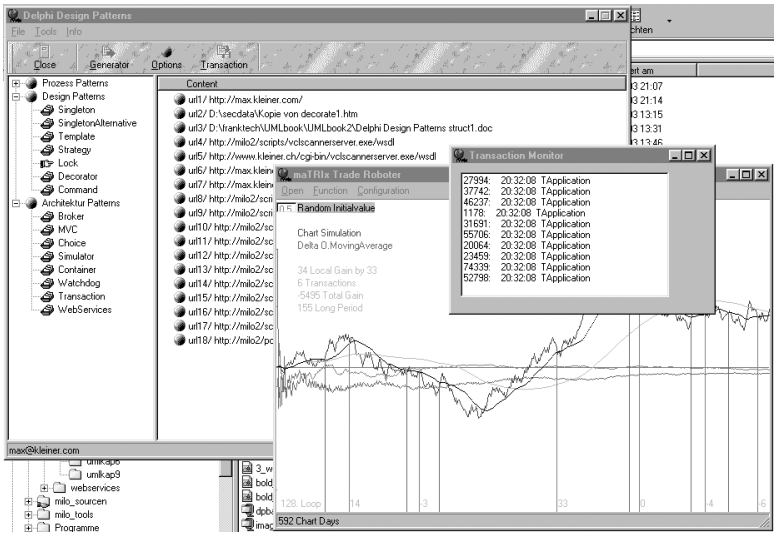


Abb. 2.77: Der Observer aktualisiert Chart und Transaktion

Die Objekte werden am Anfang von einer Formklasse konstruiert und dann wie gesagt registriert, d. h. als Referenzen in eine Liste aufgenommen, die das Subjekt (`TSubject-Center`) verwaltet. Das Subjekt kennt nur die Liste von registrierten Objekten (Beobachter), die das Subjekt anhand der abstrakten Schnittstelle benachrichtigen soll. Das Subjekt kennt keine der konkreten Klassen.

> *Demzufolge ist die Kopplung zwischen Subjekt und Beobachtern auf ein optimales Minimum reduziert, da aus der Logikschicht kein direkter Zugriff auf die Viewschicht nötig ist.*

Im TRadeRoboter existieren als Beispiel zwei Objekte als Observer (`obBars` und `obMemo`) und je nach Bedarf noch ein Adapter, die man in der Liste registriert:

```
obBars:= TProgBars.createBars(grpBox);
obMemo:= TBMonitor.create(NIL);
with ctr.subjectCtr do begin  //registrieren im Subject
  add(obBars);
  add(obMemo);
end;
```

C#-Code-Beispiel mit ArrayList:

```
  private void FormCreate(object Sender, System.EventArgs e) {
    FSubject = new TMySubject();
    FMyBars = new TMyBarsMaker(pnlContainer);
    FMyTrackBars = new TMyTrackBars(pnlContainer, FSubject);
    FFormAdapter = new TFormAdapter(this);
    startState = false;
    FSubject.Add(FMyBars);
    FSubject.Add(FMyTrackBars);
    FSubject.Add(FFormAdapter);
  }
```

Bei einer Zustandsänderung, in meinem Fall eine Transaktion, möchten die Objekte benachrichtigt werden, um ihre Darstellung aktualisieren zu können. Eine Änderung muss der Client immer zentral dem Subjekt melden. Dies geschieht über die Methode `changeAmount`, die dann wiederum mit `Notify` die Objekte einzeln benachrichtigt:

```
procedure TSubjectCenter.changeAmount(value: double);
begin
  if value <> 0  then begin
    FAmount := value;
    Notify; //state change notification
  end;
end;
```

C#-Code-Beispiel mit integer statt double:

```
    public void setX(int value) {
      if (value != Fx) {
        Fx = value;
        this.Notify();
      }
    }
```

Die Methode `Notify` ist nun angewiesen, dass alle Objekte die Methode `Update` besitzen (design by contract), damit die Nachricht ankommen kann:

```
procedure TSubjectCenter.Notify;
var  i: integer;
begin
  for i := 0 to pred(FObservers.Count) do
    TObserver(FObservers.Items[i]).Update(Self);;
end;
```

C#-Code-Beispiel mit Enumerator:

```
  public override void Notify(){
    IEnumerator objEnumerator = FObservers.GetEnumerator();
    while (objEnumerator.MoveNext()) {
      ((TObserver) objEnumerator.Current).Update(this);
    }
  }
```

Bis zum jetzigen Zeitpunkt hat noch keine Aktualisierung stattgefunden, man beachte auch, dass die Zustandsänderung der Transaktion direkt in das Feld `FAmount` im Subjekt eingeschrieben wurde. Nun kommt das Geniale am Observer.

Jedes Objekt ist nun nach der Benachrichtigung durch `Notify` selbst verantwortlich, die aktuellen Daten aus dem Subjekt zu holen (d. h. aus dem Feld `FAmount` zu lesen) und sich je nach Kontext und Zeitpunkt zu aktualisieren.

Wie das Objekt überhaupt weiß, wo sich die Daten befinden, dazu dient der self-Parameter, der wie beim Strategie-Muster jedem Objekt übergeben wird. Die Objekte kennen nun die aktuelle Adresse des Subjekts, um sich die Daten anschließend mit der Methode `getAmount` zu holen:

```
procedure TBMonitor.Update(subjectCtr: TObservable);
begin
  with memMon.lines do begin
    add(intToStr(ctr.KundenNo));
    add(floaTtoStr((subjectCtr as TSubjectCenter).getAmount));
    add(timeToStr(time)+' / '+ dateToStr(date));
  end;
end;
```

C#-Code-Beispiel mit Adapter (`getX` steht für `getAmount`):

```
  public override void Update(TObservable ChangedSubject) {
    FAdaptee.UpdateView();
  }
  public void UpdateView() {
    Xed.Text = System.Convert.ToString(FSubject.getX());
  }
```

So steht jedem Objekt frei, sich wie und wann zu aktualisieren, und die Methode `Update` lässt sich mit `override` sehr spezifisch gestalten. Mit dem self-Parameter erreichen wir sogar die **Unabhängigkeit** vom Subjekt, d. h., die Daten lassen sich auch in einem anderen Subjekt prozessweit verwalten, das Objekt ist ja bei der Benachrichtigung immer im Besitz der aktuellen Adresse. Nun werfen wir einen Blick auf das Klassendiagramm:

Beide Oberklassen sind abstrakt deklariert, eine davon hat ein Grundverhalten und ist als Singleton mit Klassenmethoden implementiert. So kann ich sowohl weitere Objekte hin-

zufügen als auch ein zusätzliches Model (`TSubjectCenter`) einrichten. Noch ein Wort zur Standardisierung. Jedes Objekt muss sich an die `Update`-Methode halten.

Wenn sich eine neue Methode, wie im Falle TRadeRoboter die Methode `reset`, aufdrängt, müssen die Objekte diese Operation nachträglich als „design by contract“ (siehe Teil 1 über OCL) implementieren. Dazu gibt es Ansätze, wie ein Objekt während der Laufzeit zu neuer Funktionalität mithilfe von Schnittstellen kommt.

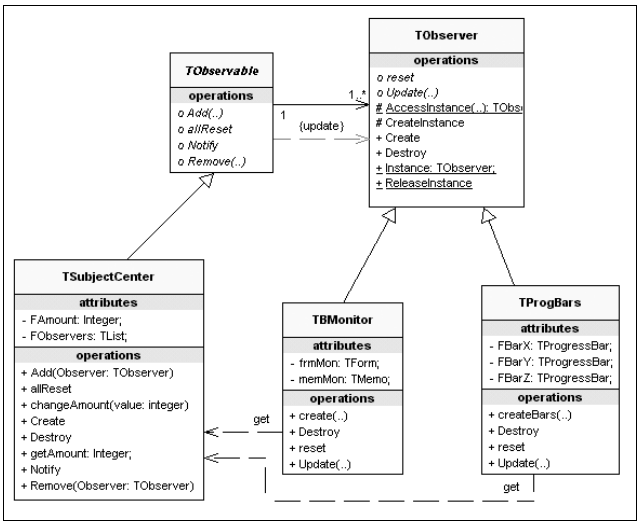


Abb. 2.78: Der Observer mit seinen zwei Beobachtern

Als Krönung sei noch der dynamische Blick auf das SEQ gezeigt:

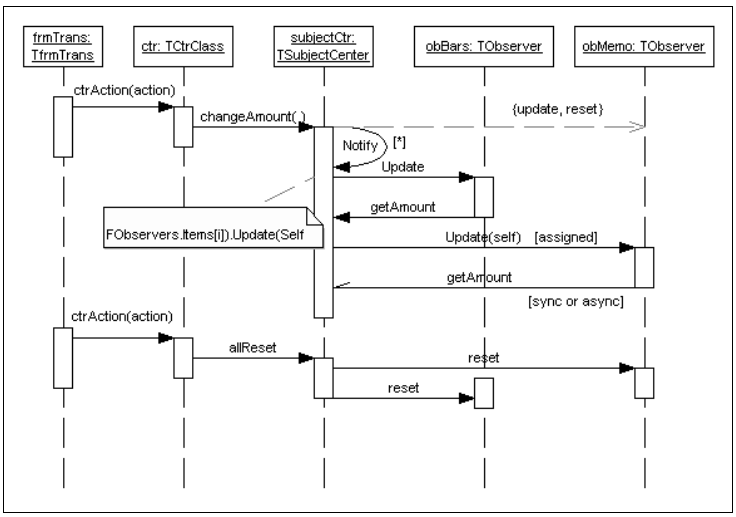


Abb. 2.79: Der Observer im Nachrichtendienst

Verwendung

Wenn man in den Sourcen von Borland nach Update sucht, werden 199 Dateien als gefunden angezeigt. Auch mit dem Suchwort register kommt man noch auf 186 Einträge. Die Verwendung des Patterns ist teilweise auch in der IDE von Delphi vorhanden.

Eine echte Verwendung ist hingegen der SQL-Monitor von InterBase. Diese Komponente fängt die zwischen einer SQL-Verbindungskomponente und einem Datenbankserver gesendeten Informationen ab und speichert diese in einer Stringliste. Zuerst werden die Monitore einzeln registriert:

```
procedure TIBSQLMonitorHook.RegisterMonitor(SQLMonitor:
                                        TIBCustomSQLMonitor);
begin
  if not FEventsCreated then
  try
    CreateEvents;
  except
    SQLMonitor.Enabled := false;
  end;
  if not Assigned(FReaderThread) then
    FReaderThread := TReaderThread.Create;
  FReaderThread.AddMonitor(SQLMonitor);
end;

procedure TReaderThread.AddMonitor(Arg: TIBCustomSQLMonitor);
begin
  EnterCriticalSection(CS);
  if FMonitors.IndexOf(Arg) < 0 then
    FMonitors.Add(Arg);
  LeaveCriticalSection(CS);
end;
```

Da die Implementation mit einem Thread funktioniert (eine Art Subjekt), ist die Benachrichtigung mit `Execute` durch ein asynchrones `PostMessage()` an die registrierten Monitore realisiert.

```
procedure TReaderThread.Execute;
var
  i: Integer;
  FTemp : TTraceObject;
begin
{$IFDEF MSWINDOWS}
  while (not Terminated) and (not bDone) do begin
    ReadSQLData;
    if (st.FMsg <> '') and not ((st.FMsg = ' ') and
                    (st.FDataType = tfMisc)) then begin
```

```
      for i:= 0 to FMonitors.Count - 1 do begin
        FTemp:= TTraceObject.Create(st);
        PostMessage(TIBCustomSQLMonitor(FMonitors[i]).Handle,
                    WM_IBSQL_SQL_EVENT, 0, LPARAM(FTemp));
      end;
    end;
  end;
{$ENDIF}
end;
```

Aus der Sicht des Clients gibt es noch eine kleine Warnung:

Rufen Sie `SetTraceCallbackEvent` nicht auf, wenn dem TSQLConnection-Objekt eine TSQLMonitor-Komponente zugeordnet ist. `TSQLMonitor` benötigt für seine Arbeit einen Callback-Mechanismus, und `TSQLConnection` kann jeweils nur einen Callback unterstützen.

Auch die Synchronisation zwischen den schreibenden Ereignissen und den lesenden Beobachtern dringt hart in die Technik der kritischen Abschnitte ein:

1. Die registrierten Leser müssen auf ein Write-Event warten.
2. Blockieren, damit die Schreiber fertig werden.
3. Blockieren, damit die Leser fertig werden.
4. Deblockieren der Leser, die auf ein Write-Event warten.
5. Warten, bis alle Leser fertig gelesen haben.
6. Blockieren aller, die auf ein Write-Event warten.
7. Deblockieren aller Leser, die auf ein Abschließen des Schreiben warten.
8. Deblockieren des Mutex.

2.5.9 State

Ein objektbasiertes Verhaltensmuster (Verhaltensänderung durch Zustandswechsel).

Zweck

> *Ermögliche es einem Objekt, sein Verhalten so zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine zugehörige Klasse gewechselt hat.*

Motivation

Es gibt Objekte, die sich je nach Zustand anders verhalten müssen. Zustandsdiagramme beschreiben die Sicht auf solch dynamisches Verhalten der im Klassendiagramm definierten statischen Objekte. Jedes Zustandsdiagramm ist **einer** Klasse zugeordnet und beschreibt deren Verhalten genauer.

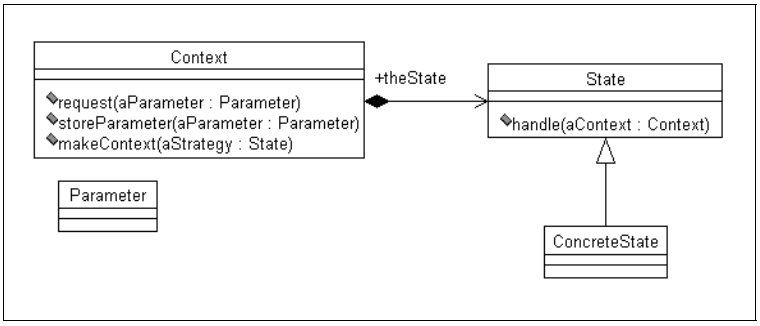


Abb. 2.80: Das State reagiert auf den Kontext

Das gesamte System und damit jede Instanz einer Klasse befindet sich zu jedem Zeitpunkt in einem wohldefinierten Zustand (jedenfalls tut ein artiges Objekt das). Durch eine äußere Einwirkung (Ereignis)[9] kann sich der Zustand eines Objektes derart ändern, dass andere Methoden (Verhalten) zum Tragen kommen.

Das State Pattern bietet eine Lösung, solches Verhalten technisch umzusetzen, ohne einen langen Case/Event-Loop von Zustands-Flags abfragen zu müssen, der auch nur mäßig wartbar ist. Dies setzt Techniken voraus, die zu den OO-Grundlagen gehören wie Delegation, Polymorphie und Parametrisierung.

> *Ein deutliches Beispiel für die Objekt-Komposition ist die Delegation. Dabei wird eine Instanz einer Klasse in einem Objekt erzeugt, die dann spezifische Aufgaben übernimmt.*

Wenn sich nun während der Laufzeit diese Aufgabe ändert, wird einfach diese Instanz durch eine andere des gleichen Typs ersetzt. Beim State Pattern sieht das so aus:

Ein Objekt delegiert eine Anfrage an ein State-Objekt, das den aktuellen Zustand des Objekts repräsentiert, worauf sich das Verhalten mit den zugehörigen Methoden verändern kann. Andere Design Patterns benutzen die Delegation auch, die einen mehr, die anderen weniger (siehe Pattern-Technik-Übersicht im Anhang).

Stellen Sie sich folgendes State-Event-Diagramm eines Dokumentensystems vor, das man in Code umsetzen müsste. Vom Modell zum Code also. Die Zustandsübergänge sind definiert, sodass ähnlich einer „Finite State Machine“ (FSM)[x] nicht beliebige Übergänge zwischen den Zuständen möglich sind (siehe Abb. 2.81).

Ein Dokument hat die fünf gezeichneten Status-Ausprägungen. Genau genommen sind es die Einträge aus einer Mängelliste, welche den jeweiligen Status besitzen. Ein Mangel wird auf eine Anfrage hin eröffnet und in den Status <opened> gesetzt. Der Mangel wird behoben und erhält den Status <ready>, was meint, er ist für den Test bereit.

Das bedeutet also auch für die Idee des State Patterns: Wenn das Objekt den Zustand von <opened> auf <ready> ändert, wechselt auch das Verhalten von <SetDate> auf die Methode <proofFunction> mit.

[9] „An Event is an important Change in State“.

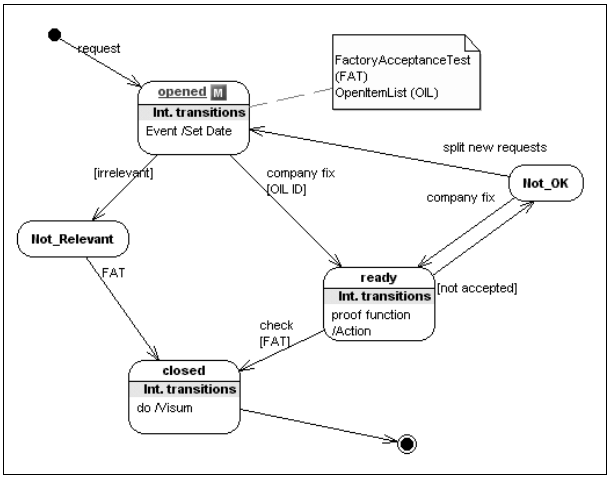


Abb. 2.81: Vom Diagramm zum State Code

Nun kommen wir zur Auflösung der erwähnten Aufgabe. Wenn es etwas wie einen State-Generator gibt, würde der Generator folgenden simplen Coderumpf, ohne zugehöriges Verhalten, aus dem Diagramm produzieren:

```
[OILStates]: procedure checkStates(event: TOILEvents);
case State of
  inherited checkStates(event)

  Init:   if event= request then state:= opened
  opened: if event= company_fix then state:= ready
          if guard= irrelevant then state:= Not_R
  ready:  if guard= not_accepted then state:= Not_OK
          if event= check(FAT) then state:= closed
  Not_OK: if event= company_fix then state:= ready
          if event= split_new_requests then state:= opened
  Not_R:  if event= FAT then state:= closed
  closed: NIL
end
```

Diese Technik zeigt, wie zumindest ein exakter Coderumpf inklusive Setzen der richtigen Zustände entstehen kann. Als nächsten Schritt würde man das Verhalten einrichten, indem zu den gültigen Zuständen die Methoden aufgerufen werden:

```
case State of
  opened: if event= company_fix then do begin
            state:= ready
            proofFunction()
          end.....
```

Diese Motivation soll zeigen, dass es ohne aufwändige Zustandsabfrage nicht geht, wenn die Zustände an eine exakte Ereignisfolge gebunden sind. Wenn diese Ereignisfolge nicht zwingend ist, bietet das State Pattern einen echt objektorientierten Ansatz, welcher auch für das soeben erstellte Beispiel seine Gültigkeit hat.

Das State Pattern vermeidet jedoch eine zentrale Zustandsabfrage mittels Case-Fallunterscheidungen, indem der Objektzustand als **Eigenschaft** in der Klasse vorhanden ist.

Implementierung

Das Beispiel stammt aus einem Zeicheneditor-Ansatz und wurde auf Delphi3000.com diskutiert. Angenommen Sie bieten ein Zeichen-Tool mit den Funktionen Selektieren, Zoomen und natürlich Zeichnen mit den jeweils gleichen Maus-Ereignissen wie `MouseDown` oder `MouseMove` in der Menüliste, Symbolleiste etc. an. `MouseDown` muss aber bspw. je nach Funktion ein anderes Verhalten zeigen. Im Normalfall, d. h. ohne Musterentwurf, würde daraus folgende, erwähnte Fallunterscheidung nach Zuständen resultieren:

```
procedure TForm.FormMouseDown/FormMouseMove/FormMouseUp
begin
  ..
  if sSelectionFlag then begin
    ...
  end else
  if sPaintFlag then begin
    ...
  end;
  ..
end;
```

Mit der Zeit müssten Sie bspw. mehr und mehr Flags verwalten wie:

`sPenFlag`, `sDrawRectFlag`, `sConvertFlag`, `sEraserFlag` etc.

Diese zusätzliche Zentralisierung führt zu schlecht verständlichem Code und einzelne Methoden lassen sich kaum wieder verwenden. Auch das Testen erweist sich dann als unübersichtlich.

Mit dem State wird nun Hilfe geboten. Von einer Basisklasse aus lässt sich das Verhalten in den abgeleiteten Klassen kapseln, somit wird mit einer Dekomposition die Zentralisierung aufgebrochen. Jede Klasse überschreibt die Methoden mit eigenem Verhalten.

Im diesem Fall reicht die Klasse `TConForm` die Anfrage an das aktuelle Tool-Objekt weiter. Ich zeige gleich, wie das funktioniert. Voraussetzung ist, eine Klasse (`TConForm`) zu bauen, welche die Zustandsänderungen entgegennimmt. Im Weiteren müssen in einer abstrakten Klasse (`TTool`) die Eventhandler angelegt werden, die man in den abgeleiteten Klassen mit `override` überschreibt, da sie polymorph sind. Hier die erste Struktur mit den jeweiligen Eventhandlern:

```
TTool = class
  private
    fForm : TConForm;
  public
    constructor Create(Form : TConForm);
    destructor Destroy; override;
    procedure SetCursor; virtual;
    procedure HandleMouseDown(x,y :integer); virtual;
    procedure HandleMouseMove(x,y :integer); virtual;
    procedure HandleMouseUp(x,y :integer); virtual;
    procedure StopAction; virtual;
  end;

TSelectionTool = class(TTool)
  public
    procedure SetCursor; override;
    procedure HandleMouseDown(x,y: integer); override;
    procedure HandleMouseMove(x,y: integer); override;
    procedure HandleMouseUp(x,y: integer); override;
  end;

TPaintTool = class(TTool)
  public
    procedure SetCursor; override;
    procedure HandleMouseDown(x,y: integer); override;
    procedure HandleMouseMove(x,y: integer); override;
    procedure HandleMouseUp(x,y: integer); override;
  end;
```

Wenn man den Originaltext analysiert (ermögliche es einem Objekt, sein Verhalten so zu ändern, wenn sein interner Zustand sich ändert ...), kommt man auf folgenden Schluss:

Das Objekt, welches sein Verhalten ändert, ist nicht etwa eines der Zustands-Objekte wie `TSelectionTool` oder `TpaintTool`, sondern es ist die Klasse `TConForm` selbst, die ihr Verhalten je nach Zustand ändert. Diese Klasse wird im Muster auch Kontext genannt.

Dieser Kontext hat eine Kernmethode in mehrfacher Ausprägung, welche den Zustand effektiv ändert. Diese Kernmethode kann `SetSelectionState` heißen. Das Ändern des Zustands geschieht z. B. dadurch, dass der Anwender in der Werkzeugleiste auf den Zoom-Button drückt und intern einen Zustandswechsel wie folgt auslöst:

```
procedure TConForm.SetSelectionState;
begin
  fActiveTool:= mySelectionTool;
end;
```

```
procedure TConForm.SetPaintState;
begin
  fActiveTool:= myPaintTool;
end;
```

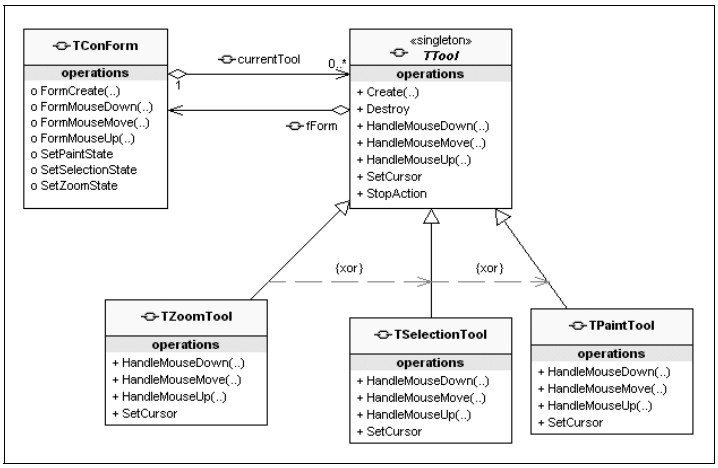


Abb. 2.82: Mit dem State hat jede Klasse ihr eigenes Verhalten

Anstelle von vielen Flags erhalte ich mit dem zur Klasse lokalen Parameter `fActiveTool` den jeweils aktuellen Zustand zugewiesen. Das Weiterleiten der Anfrage kann ich mit dem zugehörigen Event verarbeiten, welches den momentanen Status als Objekt beinhaltet, will heißen, das **Objekt ist** der Zustand:

```
procedure TForm.FormMouseDown(... X, Y: Integer);
begin
  fActiveTool.HandleMouseDown(x,y);
end;

procedure TForm.FormMouseMove(... X, Y: Integer);
begin
  fActiveTool.HandleMouseMove(x,y);
end;
```

Im Sequenzdiagramm sei das Verhalten mit dem Kontextwechsel dargestellt. Das Szenario beschreibt die Wahl des Anwenders, etwas zu selektieren, worauf die Routine durch das Ereignis des Anwenders das Zustands-Objekt wechselt und die Methode `MouseDown` somit ein anderes Verhalten an den Tag (sprich Oberfläche) legt (siehe Abb. 2.83).

Durch die Polymorphie der Eventhandler ist das Weiterleiten an z. B. `HandleMouseMove` auch mit dem richtigen Objekt `fActiveTool` assoziiert. Zum Schluss eine Vertiefung des Gelernten mit dem State-Event-Diagramm, welches der Klasse `TConForm` gewidmet ist,

wobei eben zu berücksichtigen sei, dass der Aufruf an ein Zustands-Objekt delegiert wird (mehr davon in der Verwendung, siehe Abb. 2.84).

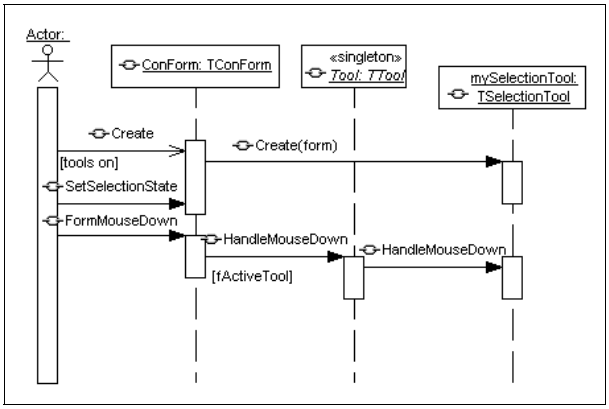


Abb. 2.83: Auf ein Ereignis folgt der Verhaltenswechsel

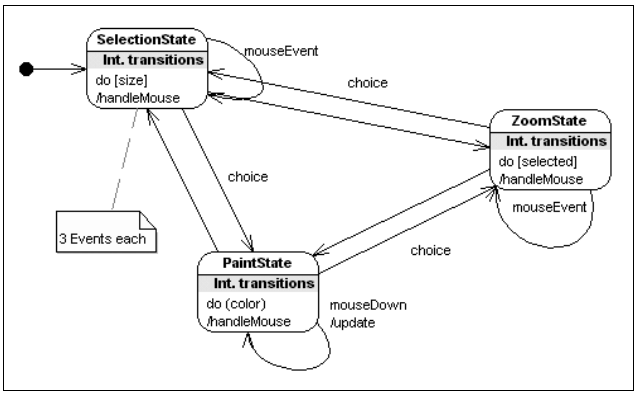


Abb. 2.84: Das Delegieren an ein Zustands-Objekt mit dem SE

Verwendung

Mit `TIconViewItem` können Sie auf die einzelnen Elemente einer Symbolansicht zugreifen. Die gesamten Einträge werden in der Eigenschaft `Items` der Symbolansicht mithilfe eines `TIconViewItems`-Objekts verwaltet.

Mit der ReadOnly-Eigenschaft `States` können Sie auf den Status des Elements zugreifen. Die Eigenschaft kann die Zustände als Werte `isNone` (Standardstatus), `isFocused` (Eingabefokus), `isSelected` (ausgewählt) und `isActivating` (aktiv) annehmen.

`TItemStates` besteht aber aus einer Menge (Set) von `TItemState`-Werten und nicht einzelnen Objekten, eine Erweiterung ist aber durchaus vorstellbar.

Als weitere Verwendung bietet sich eine Vertiefung in die Technik der Delegation an. Einige State Patterns benutzen für die Weiterleitung an das Zustands-Objekt diese Technik. Delegation lässt sich mit Polymorphie oder mit Methodenzeigern realisieren. Die reine Delphi- wie C#-Lehre besagt, dass Delegation am besten mit Methodenzeigern funktioniert. Die sind in der Unit System (früher in SysUtils) deklariert:

```
TMethod = record
  Code, Data: Pointer;
end;
```

Angenommen ich benötige die Fortschrittsanzeige einer Simulationsberechnung auf einem Form. Als ersten Ansatz kann ich eine herkömmliche Win-Message an mein Form mit dem zugehörigen Handle senden:

```
SendMessage(ProgressForm.Handle,WM_ProgressMsg,Progress,0);
```

Das Form empfängt die Nachricht und reagiert mit einer eigenen Prozedur:

```
procedure WMProgress(var msg: TMessage); message WM_Progress;
```

Das ist der normale Windows-Weg à la Petzold, bei dem früher galt: „Nur wer die Nacht durchdenkt, sieht die Morgenröte.“ Delphi benutzt diese Technik, um die Nachrichten in einen Methodenzeiger zu wandeln. Wenn eine Win-Message wie `WM_Activate` daherkommt, generiert das System das zugehörige Ereignis `OnActivate`.

> *Das Event kapselt also die Antwort für eine bestimmte empfangene Nachricht mithilfe eines Methodenzeigers.*

Anstelle von `SendMessage` deklariere ich nun einen Methodenzeiger als Delegate:

```
TProgressMsg= procedure(Msg: string;Progress: integer)of object;
```

Da ein Ereignis ein Zeiger auf eine Ereignisbehandlungsroutine ist, muss der Typ der Ereigniseigenschaft ein Methodenzeigertyp sein.

```
constructor TProgressForm.Create(LCID: integer;
                                 CompressClass: TCompressClass);
begin
  inherited create(NIL);
  fCompressClass:= CompressClass;
  fCompressClass.onProgressMsg:= ProgressMsg;
  ...
end;
```

Nun lässt sich das Darstellen des Fortschrittes auf dem `ProgressForm` an das Form selbst delegieren, da die Behandlungsroutine `ProgressMsg` jedes Mal startet, wenn das Event `onProgressMsg` feuert:

```
procedure TCompressClass.SendProgressMsg(Msg: string;
                                         Progress: integer);
begin
  if Assigned(onProgressMsg) then
    onProgressMsg(Msg,Progress);
end;
```

Die Aufrufkaskade zeigt folgendes Bild:

```
Message=SendProgressMsg -> Event=onProgressMsg ->
EventHandler=ProgressMsg
```

Der eigentliche Eventhandler `ProgressMsg` kann dann endlich auf die Nachricht, hoffentlich im positiven Sinne des Fortschrittes, reagieren ;):

```
procedure TProgressForm.ProgressMsg(Msg: string;
                                    Progress: integer);
var s : string;
    k : integer;
begin
  StepLabel.Caption:=Format(sProgress+
                            ' %s',[IntToStr(Progress)])+'%';
  StepLabel.Refresh;
end;
```

2.5.10 Strategy

Ein objektbasiertes Verhaltensmuster (Austauschen des Algorithmus).

Zweck

Bestimme eine Gruppe von Algorithmen, kapsle jeden einzelnen Algorithmus und ermögliche deren Austauschbarkeit während der Laufzeit. Das Strategie-Muster ermöglicht, den Algorithmus unabhängig vom Aufrufer zu variieren.

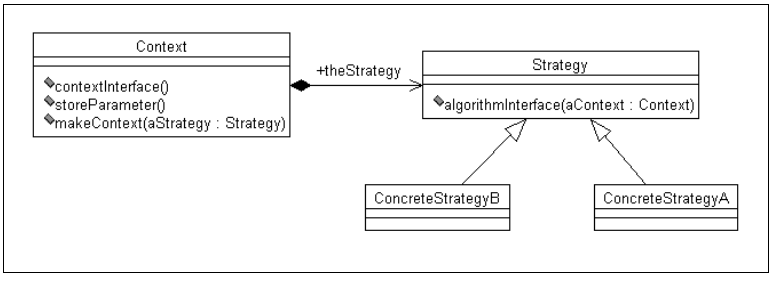


Abb. 2.85: Die Strategy lässt Algorithmen variieren

Motivation

Strategien sind immer gefragt, auch wenn die operative Hektik schon am überborden ist. Das Strategie-Muster legt den Fokus auf die Algorithmen und deren Aufgabe, etwas zu berechnen oder mit Geschäftslogik zu brillieren. Ein optischer Simulator oder eine starke Schachanwendung leben von Strategien.

> *Bei den meisten Brettspielen wie 4 gewinnt, Schach oder Mühle berechnet der Computer die möglichen Züge nach dem Minimax-Algorithmus. Dabei prüft er alle möglichen Zugkombinationen bis zu einer sinnvollen Tiefe und rechnet dabei anhand einer Bewertungsfunktion die eigenen und gegnerischen Gewinne zusammen.*

Jetzt kommt die eigentliche Idee des Minimax-Algorithmus:

Den Zug, der bei optimalem Spiel des Gegners den größten eigenen Gewinn zulässt, führt das Programm dann effektiv aus. Das optimale Spiel des Gegners ist eine Annahme, die aber meistens stimmt, da ja auch der Gegner gewinnen will.

Die ganze Bewertung erfolgt entlang eines Baumes und die Bewertung einer Stellung lässt sich mit ganzen Zahlen ausdrücken. Je größer der Absolutbetrag der Bewertung, desto größer ist derVor- bzw. Nachteil für die jeweilige Seite.

> *Die Routine muss jeweils nur die Endstellungen bewerten; die Übrigen lassen sich durch abwechselnde Maximum- bzw. Minimumsuche bestimmen, was einem Zug und einem Gegenzug entspricht. Daher der Name Minimax.*

Beim Strategie-Muster, auch unter dem Namen Policy bekannt, ist es nicht wünschenswert, alle möglichen Algorithmen in den Klassen fest zu codieren. Das Muster schlägt vor, Klassen zur Kapselung der unterschiedlichen Berechnungen zu deklarieren. Solch einen gekapselten Algorithmus nennt man Strategie.

Implementierung

Im Falle des TRadeRoboter kommt das Strategie-Muster bei der Berechnung der Gebühren beim exemplarischen Kreditkonto zum Einsatz. Exemplarisch meint, es kann auch ein anderes Konto sein, man wechselt dann die Referenz auf das Strategiekonto aus.

Auch die Kontextklasse ist auswechselbar, konkret kann ich die Berechnung der Monatsgebühr durch die Jahresgebühr zu den einzelnen Konten auswechseln.

Wenn der Bankangestellte die gewünschte Kontogebühr auf das Jahr oder den Monat wählt, lässt sich zur Laufzeit von der abstrakten Methode `ComputeCharges` die konkrete Methode in einer Controller-Klasse mit der entsprechenden Strategie aufrufen:

```
51: begin
     TKreditkonto(konto).Jahresgebuehr:=
        FRegularCharges.ComputeCharges(StrToFloat(amount));
    end;
52: begin
     TKreditkonto(konto).Jahresgebuehr:=
        FPreferred.ComputeCharges(StrToFloat(amount));
```

```
    end;
53: begin
     TKreditkonto(konto).Jahresgebuehr:=
        FTrial.ComputeCharges(StrToFloat(amount));
    end;
```

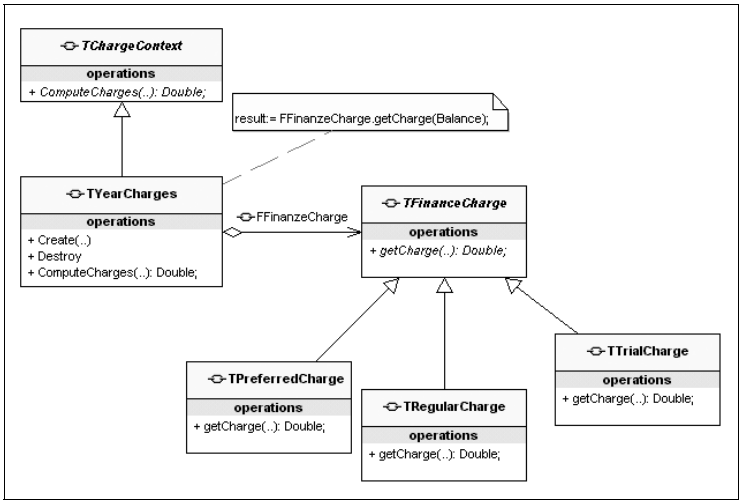


Abb. 2.86: Die Strategie hat 3 Algorithmen parat

Die Instanzen werden im Moment im Konstruktor des Controllers ins Leben gerufen, noch dynamischer wäre, die Instanzierung erst bei der Auswahl zu generieren:

```
51: begin
      FCharges:=
          TYearCharges.Create(TRegularCharge.Create);
      TKreditkonto(konto).Jahresgebuehr:=
          FCharges.ComputeCharges(StrToFloat(amount));
      FCharges.Free;
    end
//even shorter alternative
  with TYearCharges.Create(TRegularCharge.Create) do begin
    TKreditkonto(konto).Jahresgebuehr:=
                   ComputeCharges(StrToFloat(amount));
    Free;
  end;
```

C#-Code-Beispiel:

```
   FRegCharges = new uStrategy.TMonthlyCharges(
                    new uStrategy.TRegularCharge());
```

```
   FPrefCharges = new uStrategy.TMonthlyCharges(
     new uStrategy.TPreferredCharge());
   FTrialCharges = new uStrategy.TMonthlyCharges(
        new uStrategy.TTrialCharge());
```

Die Instanz lässt sich dann der Kontextklasse `TYearCharges` oder `TMonthlyCharges` übergeben, die durch den bekannten abstrakten Typ die konkrete Instanz einem Member zuweist. Die Klasse `TYearCharges` verwaltet also eine Referenz auf ein Strategie-Objekt. Das Strategie-Muster verdeutlicht ja mit der Übergabe einer **Objektreferenz**, dass hier maximale Flexibilität mit `self` als Objekt-Parameter erreicht wird.

```
TYearCharges = class(TChargeContext)
private
  FFinanceCharge: TFinanceCharge;
public
  constructor Create(aFinanceCharge: TFinanceCharge);virtual;
                         destructor Destroy; override;
  function ComputeCharges(const Balance:double):
                                     Double; override;
end;
```

Die Objektreferenz wird schlussendlich im Konstruktor an die Instanz `FFinanceCharge` gebunden, die eine Aggregation darstellt. Nun, die meisten Aggregationen erfolgen zur Designzeit, beim Strategie-Muster bestätigt sich, dass die Referenz bereits im Konstruktor erzeugt wird, bis der Destruktor sie scheidet:

```
constructor TYearCharges.Create(aFinanceCharge: TFinanceCharge);
begin
  inherited Create;
  if not assigned(aFinanceCharge) then
    raise Exception.Create('missing object');
  FFinanceCharge:= aFinanceCharge;
end;
```

C#-Code-Beispiel mit Exception Handling:

```
  public TMonthlyCharges(TFinanzeCharge aFinanzeCharge) {
    if (aFinanzeCharge == null) {
      throw new ApplicationException
        ("Missing FinanzeCharge object");
    }
    FFinanzeCharge = aFinanzeCharge;
  }
```

Das Zuweisen des Strategie-Objekts zur Referenz erfolgt mit `ComputeCharges`, das auch gleich die richtige Methode des Strategie-Objekts aufruft und als Resultat zurückgibt:

```
result:= FFinanceCharge.getCharge(Balance);
```

C#-Code-Beispiel:

```
public double CompCharges(double Balance) {
  return FFinanzeCharge.getCharge(Balance);
}
```

Sie können Objektreferenzen, etwa auch zur Verwendung als Callback, übergeben. Ich bin mir hier immer noch nicht sicher, aber das Kriterium des Rückrufs ist beim Strategie-Muster wie übrigens auch beim Observer gegeben.

> *Um Gebrauch von einem Callback zu machen, enthält die Schnittstelle des Anwendungsservers eine Methode, die eine Client-Schnittstelle als Parameter akzeptiert. Die Client-Anwendung ruft diese Methode auf und übergibt seine eigene Instanz. Der Anwendungsserver kann dann die Schnittstelle verwenden, um die Client-Anwendung aufzurufen.*

Beim Sequenzdiagramm erleben Sie nun eine Optimierung, welche die Interaktionen des eher schwierigen Patterns noch einmal verdeutlichen soll. Da die konkrete Objektreferenz nicht bekannt ist, sondern sich ja zur Laufzeit als Strategie ergibt, operiere ich hier im SEQ mit dem Stereotyp <Ref>.

Die eigentliche Berechnung der Gebühr ist dann in den einzelnen Klassen (in meinem Fall drei Klassen) des abstrakten Typs `TFinanceCharge` mit der Methode `getCharge` zu finden und ergibt jeweils das Resultat pro Strategie.

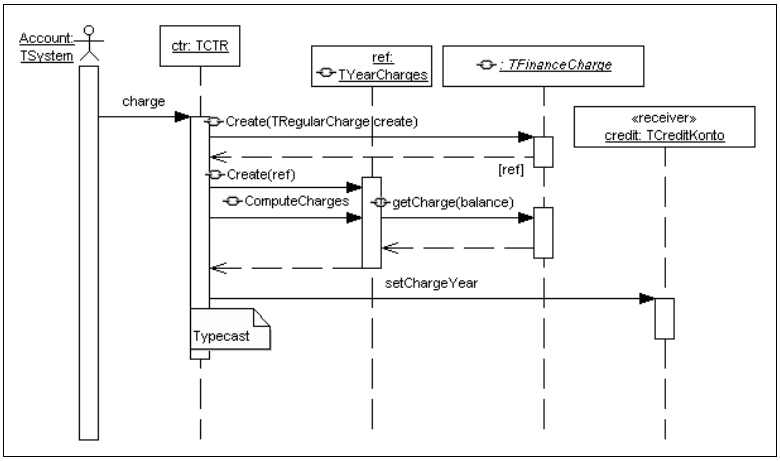


Abb. 2.87: Aus einer Strategie kann man wählen

Verwendung

Der Entscheidungswürfel `TDecisionCube` ist ein mehrdimensionaler Datenspeicher, der seine Daten aus einer Datenmenge bezieht. Bei dieser Datenmenge handelt es sich typi-

scherweise um eine speziell strukturierte SQL-Anweisung, die man über `TDecisionQuery` oder `TQuery` erstellt.

Die Daten sind dann so gespeichert, dass der Anwender sie auf einfache Weise reorganisieren und zusammenfassen kann, ohne dass er oder die Anwendung dazu die Abfrage erneut ausführen muss. Dies geschieht durch Strategien.

```
TQueryDims = class(TDimensionItems)
private
protected
  function GetQueryDim(Index: Integer): TQueryDim;
  procedure SetQueryDim(Index: Integer; Value: TQueryDim);
  constructor Create(Owner: TPersistent;
                     ItemClass: TQueryDimClass);
public
  function Add: TQueryDim;
  property Items[Index: Integer]: TQueryDim read GetQueryDim
                           write SetQueryDim; default;
end;
```

`TDimensionItem` dient als Basis zur Definition von Klassen, welche die Felder repräsentieren, die von einem Kreuztabellenspeicher verwendet werden. `TDimensionItem` beschreibt ein Feld der Datenmenge, die von einem Kreuztabellenspeicher verwendet wird, sowie die Rolle, die das Feld im Datenspeicher spielt.

Zwei Begriffe spielen hier eine Rolle:

- Dimensionselemente (Dimension)
- Zusammenfassungswerte (Zusammenfassung)

Kreuztabellenspeicher erzeugen automatisch Dimensionselemente, um die Felder zu repräsentieren, aus denen die Berechnung die mehrdimensionalen Arrays von Zusammenfassungswerten bildet.

In einem Entscheidungsgitter (`TDecisionGrid`) werden die Daten angezeigt, die über einen Entscheidungswürfel (`TDecisionCube`) von der Datenquelle bezogen werden:

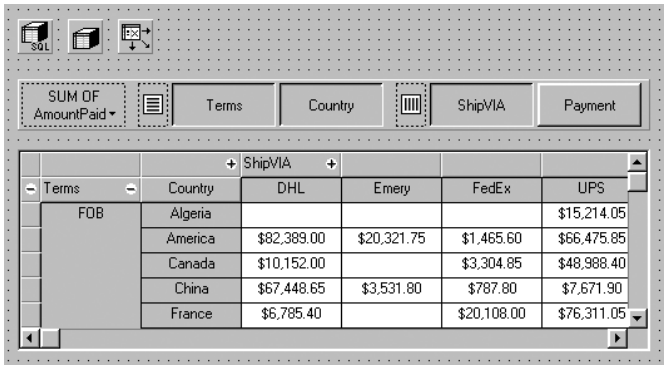
```
TDecisionGrid -> TDecisionCube -> TDecisionSource
```

Diese Umstrukturierung von Daten in einer Kreuztabelle (s. Abb. 2.88) wird als eigentliches Schwenken (pivoting) bezeichnet, daher der Begriff Pivot-Tabelle.

> *Verwenden Sie den Entscheidungspivot für Zusammenfassungen, die additiv sind, also SUM- und COUNT-Zusammenfassungen; AVERAGE, MAX und MIN sind nicht additiv.*

Kreuztabellen stellen also eine Teilmenge des Datenbestandes so dar, dass Abhängigkeitsbeziehungen und Trends deutlicher zu erkennen sind. Tabellenfelder werden zu den Dimensionen der Kreuztabelle, während die Feldwerte die Kategorien und Zusammenfassungen innerhalb einer Dimension darstellen.

Der Konstruktor von `TQueryDims` erfährt dann zur Laufzeit, welches Strategie-Objekt des Typs `TQueryDimClass` in der Collection zum Einsatz kommt.



		ShipVIA			
Terms	Country	DHL	Emery	FedEx	UPS
FOB	Algeria				$15,214.05
	America	$82,389.00	$20,321.75	$1,465.60	$66,475.85
	Canada	$10,152.00		$3,304.85	$48,988.40
	China	$67,448.65	$3,531.80	$787.80	$7,671.90
	France	$6,785.40		$20,108.00	$76,311.05

Abb. 2.88: Die Neuberechnung erfolgt ohne wiederholte Abfrage

2.5.11 Template

Ein objektbasiertes Verhaltensmuster (definiert die Schritte eines Algorithmus).

Zweck

> *Definiere das Skelett eines Algorithmus und delegiere zusätzliche Schritte an die Unterklassen. Die Verwendung eines Templates erlaubt den Unterklassen bestimmte Schritte eines Algorithmus zu überschreiben, ohne die Struktur des Templates zu verändern.*

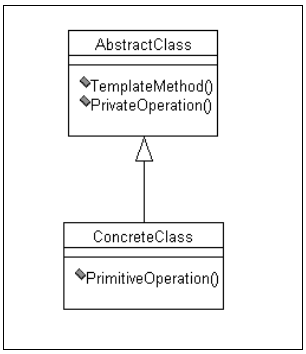


Abb. 2.89: Das Template unterteilt die Algorithmen in Schritte

Motivation

Angenommen man benötigt ein Verfahren, um einen manipulierten Text via Stream in eine Datei zu schreiben (siehe Abb. 2.90). Das Verfahren hat invariante Teile eines Algorithmus wie Datei öffnen oder Datei schreiben, die man genau einmal festlegen will. Die Unterklassen sollen dann das variierende Verhalten implementieren.

Gegenüber dem Strategie-Muster lässt sich das Template wie folgt abgrenzen:

- Eine Strategy verwendet Delegation, um den gesamten Algorithmus zu variieren.
- Ein Template verwendet Vererbung, um Teile eines Algorithmus zu variieren.

Es gibt vielfach eine spezielle Schablone (Template) für den Fall, dass nur ein einziger Algorithmus aktiv ist. In diesem Fall dient die Vererbung dazu, dass die Schablonenmethode die primitiven Operationen aufruft wie auch andere Klassen, weshalb hierzu eine eigene Schablone existiert.

> *Um die invarianten Operationen zu schützen, muss ein Client immer die Schablonenmethode aufrufen, und die Unterklassen überschreiben dann die primitiven Operationen.*

Delphi selbst macht regen Gebrauch von „allgemeinen" Templates, z. B. gibt es die Eigenschaft `Template`, welche eine benutzerdefinierte Vorlage für Dialogfelder referenziert. Diese Ressource wird von der Methode `Execute` verwendet, um die Bildschirmdarstellung der Dialogfenster festzulegen. Templates kommen auch in der CLX als Algorithmus zum Einsatz, wie unter der Implementierung zu sehen ist.

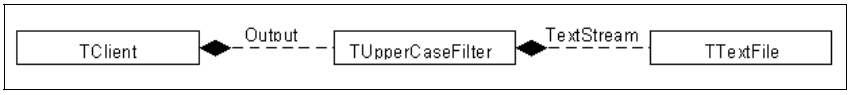


Abb. 2.90: Die Schritte vom Output zum Text-File

Implementierung

Im Falle TRadeRoboter benötige ich die vordefinierten Stream-Klassen `TMemoryStream` und `TStringStream`, um in einer Online-Version Finanz- und Börsendaten als Stream empfangen zu können.

> *Mit den spezialisierten Stream-Objekten lassen sich Informationen, die auf einem bestimmten Datenträger gespeichert sind, lesen, schreiben oder kopieren. Dazu sind in jeder von `TStream` abgeleiteten Klasse die entsprechenden Methoden implementiert.*

Außerdem können Sie mit diesen Objekten auf eine beliebige Position im Stream zugreifen. Mit den Eigenschaften von `TStream` kann man verschiedene Informationen über den Stream (z. B. Größe und aktuelle Position) abrufen. Das Template-Muster hat mit der Zeit auch die Bezeichnung Schablonenmethode erhalten.

Nun, wo ist das Template zu sehen? Die Klasse `TStream` hat eine Schablonenmethode `CopyFrom`, die wiederum die zwei Template-Methoden `ReadBuffer` und `WriteBuffer`

benötigen. Soweit ist das die erwähnte Struktur, die nicht verändert werden muss (siehe Zweck).

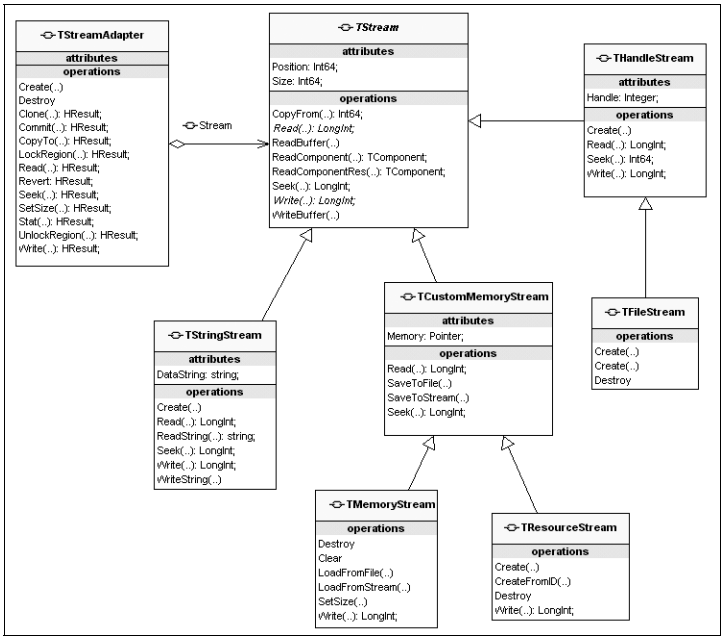


Abb. 2.91: Read und Write mit CopyFrom als Schablone

Einzig die Methoden `Read` und `Write`, die von `ReadBuffer` und `WriteBuffer` gekapselt sind, lassen sich konkret überschreiben und zeigen innerhalb des Template-Mechanismus ihre eigene Wirkung. So überschreibt oder erweitert auch bspw. die Klasse `TStringStream` die beiden Methoden.

- `CopyFrom` – die Schablonenmethode
- `Read` – primitive Operation
- `Write` – primitive Operation

Da `CopyFrom` die Methoden `Read` und `Write` mit `ReadBuffer` und `WriteBuffer` kapselt, ermöglicht das Verwenden von `CopyFrom` einen ständig strukturierten und sicheren Zugriff als Invariante von einem Client aus gesehen.

Hinter der Idee des Templates steckt ja das Verwenden einer definierten und standardisierten Vorlage, und die Vorlage ist sozusagen der Kopieralgorithmus `CopyFrom`, den man je nach gewünschtem Format wie z. B. Speicher- oder Ressourcenstreams mit `read` und `write` konkret anpassen, d. h. überschreiben, muss.

```
TStream = class(TObject)
.....
protected
  procedure SetSize(NewSize: Longint); virtual;
```

```
public //zu konkretisierende Methoden
  function Read(var Buffer; Count: Longint):
                   Longint; virtual; abstract;
  function Write(const Buffer; Count: Longint):
                   Longint; virtual; abstract;
  function Seek(Offset: Longint; Origin: Word):
                   Longint; virtual; abstract;
      //Template Methoden
  procedure ReadBuffer(var Buffer; Count: Longint);
  procedure WriteBuffer(const Buffer; Count: Longint);
  function CopyFrom(Source: TStream; Count: Longint):Longint;
.....
end;
```

Im TRadeRoboter mache ich nun mit der Funktion `getStreamTemplate` Gebrauch davon, d. h., die überschriebenen Methoden sind bereits durch die CLX in den Klassen `TMemoryStream` und `TStringStream` konkret vorhanden, die ich nach dem Konstruktor einfach nutzen kann:

```
function TBusObj.getStreamTemplate(rList: TStringList):
                                                TStringList;
var
  M: TMemoryStream;
  S: TStringStream;
begin
  M:= TMemoryStream.Create;
  S:= TStringStream.Create('');
  try
    M.LoadFromFile(ExtractFilePath(Application.ExeName)+SYSFILE);
    M.Seek(0,0);
    S.CopyFrom(M, M.Size);
    S.Seek(0,0);
    with rList do begin
      Clear;
      add('SYSTEM_FILES der Datenbank');
      addStrings(TStringList(frmTrans.getFileList
          (ExtractFilePath(Application.ExeName)+'T_BANK\*.*')));
      add(S.ReadString(S.Size));
    end;
    result:= rList;
  finally
    M.Free;
    S.Free;
  end;
end;
```

Verwendung

Obiges Beispiel zeigt, wie eng die Verzahnung einer Implementation mit einer Verwendung sein kann. Die Verwendung ist die Streamstruktur der CLX, die ich in einer Implementation direkt nutzen kann. Das heißt, ich habe kein eigenes Template entworfen, sondern nutze das Template aus der Klassenhierarchie, wie das z. B. auch mit dem Memento geschehen ist.

Eine echte Verwendung in der CLX ist auch die abstrakte Klasse `TGraphic`, welche mit dem Template-Muster die Klassen `TBitmap`, `TIcon` und `TMetafile` unterstützt. `TGraphic` definiert abstrakte Methoden wie `Draw` oder `LoadFromFile`, die mit den konkreten Methoden beliebige Formate wie PCX, GIF oder JPG bedienen können. Andere Objekte wie `TCanvas` profitieren dann vom Überschreiben der Methoden.

Eine weitere Verwendung mit Stream-Templates streift das Thema der „Baumschulen". Wie jede Baumstruktur setzt sich auch die Baumstruktur eines einfachen Dokumentensystems im TRadeRoboter aus Knoten und Kanten zusammen.

Ein Knoten ist in diesem Fall einfach eines der Textelemente oder der `TRText-Objekte` und wird somit durch die Klasse `TDocElement` repräsentiert. Eine Kante ist die Verbindung zwischen zwei Knoten. Für sie definiert die Unit eine neue Klasse:

```
TEdge = class
  StartNode, EndNode : TDocElement;
public
  procedure Store(Stream: TStream);
  constructor Load(Stream: TStream; List: TList);
  constructor StandardConnection(n1, n2: TDocElement);
  function GetEndNode: TDocElement;
end;
```

Die einzigen Daten einer Kante sind also die beiden Knoten, die verbunden werden sollen. Mit dieser allgemeinen Struktur ließen sich die verschiedensten in der Informatik verwendeten Typen von Grafen definieren, im TRadeRoboter sind diese theoretischen Möglichkeiten jedoch etwas eingeschränkt:

Zunächst handelt es sich um einen gerichteten Grafen – das heißt, in jeder Verbindung ist ein Knoten als Ausgangsknoten (StartNode) und der andere als Endknoten (EndNode) definiert. Bei der Darstellung bspw. einer Kontenhierarchie ist die Basisklasse jeweils der Anfangsknoten, die davon ausgehenden Verbindungen führen in Richtung der Wertpapiere, welche die Endknoten bilden.

> *Als zweite Beschränkung darf jeder Knoten nur einen Vorgängerknoten haben, so wie jede Klasse in Delphi nur eine Vorfahrklasse haben kann. Die Zahl der Nachfolger ist nur durch die Kapazität von TList beschränkt.*

Darüber hinaus sollte das Dokument keine Zyklen haben, d. h., dass kein Text-Element sein eigener Vorgänger sein darf. Da jedoch jeder Knoten nur einen Vorgänger haben kann, könnte in einem solchen Baum auch nur ein Zyklus entstehen.

Die wichtigsten Datenelemente von `TDocElement` sind die Liste der Nachfolgeknoten und der Zeiger auf den Vorgängerknoten (Property `PrevNode`). Die Liste `Edges` enthält Objekte der Klasse `TEdge`, um also an einen Nachfolgeknoten zu kommen, muss man aus einem dieser `TEdge`-Objekte das Element `EndNode` auslesen:

```
EndNode:= TEdge(Edges[i]).EndNode);
```

Diese scheinbar umständliche Speicherung der Verbindungsdaten ermöglicht es, später weitere Daten mit jeder Verbindung als Stream zu speichern.

2.5.12 Visitor

Ein objektbasiertes Verhaltensmuster (Operationen auf Objekte ohne Klassenänderung).

Zweck

> *Das Besucher-Muster erlaubt, neue Operationen zu definieren, ohne die Klassen der von ihm bearbeitenden Elemente zu verändern. Kapsle dazu die auf den Elementen auszuführenden Operationen als ein Objekt.*

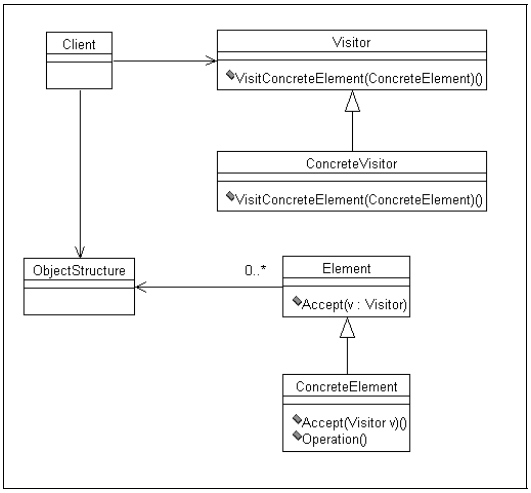


Abb. 2.92: Der Visitor besucht zuerst, bevor er operativ handelt

Motivation

Das Besucher-Muster kapselt verwandte Operationen einer jeden Klasse in einem eigenen Objekt, Besucher genannt. Die Elemente (Objekte) müssen nun den Besucher vorgängig akzeptieren, damit der Besucher die Operation ausführen kann. Somit muss diese Operation nicht mehr Teil des Elementes sein.

Angenommen es gibt verschiedene ausführbare SQL-Statements, die ich als Elemente definiert habe. Die daran beteiligten Operationen aus einer anderen Klasse ließen sich mit den Elementen neu kombinieren, was zu neuen Abfragen im Visitor führte.

Der Besucher führt die gemeinsamen Operationen für die Elemente aus. Die einzelnen Abfragen lassen sich als Gesamtstatistik aggregieren. Die Elemente allein wären nicht in der Lage gewesen, solch eine Statistik zu liefern.

Das Pattern hat sich bewährt. Wenn es sich nicht bewährt, würde man die Operationen wieder zurück in die Klassenelemente verschieben und somit die Elemente ändern.

> *Man sagt auch, dass ein Design Pattern, welches sich nach dem Einsatz nicht bewährt hat, ein Anti-Pattern ist.*

Das Besucher-Muster verfolgt wirklich eine gute Idee im Ansatz: Definieren Sie eine Methode, die auf den Elementen von anderen Klassen ausgeführt wird.

Implementierung

Ich baue nun an der Statistik weiter. Diese erwähnte Methode, die auf den anderen Klassen zum Arbeiten kommt, ist im TRadeRoboter die Gesamtstatistik, wie Varianz oder Durchschnitt eines Charts. Das Modell zeigt folgendes Bild:

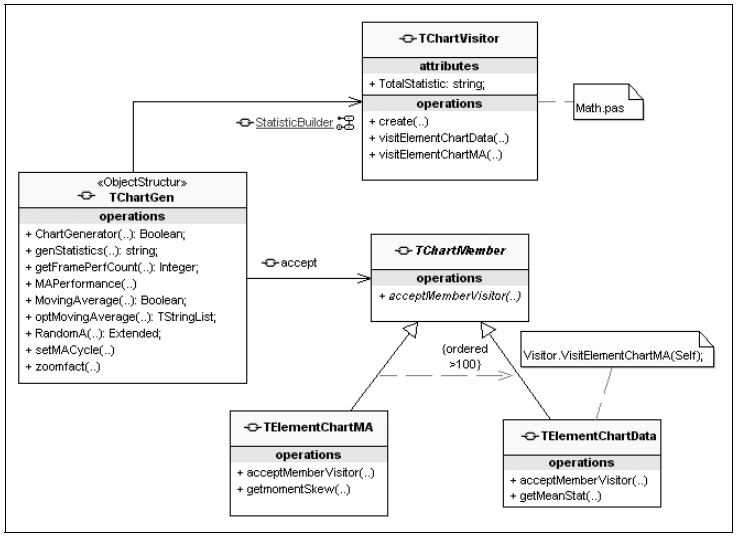


Abb. 2.93: Die Gesamtstatistik hat der Visitor im Auge

Diese Statistik setzt sich aus einzelnen Elementen der Klasse mit den zugehörigen Operationen zusammen, wie die Prozedur `MeanAndStdDev` aus der Math-Library zeigt:

```
procedure TElementChartData.getMeanStat(const data: TMemor;
                                        var mean, stddev: extended);
```

```
begin
  MeanAndStdDev(Data, mean, stddev);
end;
```

C#-Code-Beispiel mit eigener Berechnung des Mittelwerts aufgrund der Grundgesamtheit:

```
class Statistics {
  public static double getMean(ArrayList arrlNumbers) {
    double dblSum = 0;
    double dblMean = 0;
    //Console.WriteLine(arrlNumbers.Count);
    try  {
      for (int intCounter = 0; intCounter
             < arrlNumbers.Count; intCounter++) {
        dblSum = dblSum + System.Convert.
                    ToDouble(arrlNumbers[intCounter]);
      }
      dblMean = dblSum / arrlNumbers.Count;
    } catch (Exception objException) {
      Console.WriteLine(objException);
    }
    return dblMean;
  }
```

Die Prozedur `MeanAndStdDev` berechnet den Mittelwert und die Standardabweichung der Elemente eines Arrays. Da beides in einem Aufruf erfolgt, ist die Routine doppelt so schnell wie die getrennte Berechnung.

Die Statistik der einzelnen Elemente lässt sich nun durch den Visitor laufend aggregieren. Als besonderen Vorteil des Musters ist das Hinzufügen von neuen Operationen herauszustreichen, so z. B. das Vergleichen von mehreren Statistiken im Sinne einer historischen Auswertung oder im Falle des TRadeRoboter das iterative Finden des optimalen Moving Average (OMA) aus n-Aktien heraus.

```
procedure TChartVisitor.visitElementChartMA(inst:
                                          TElementChartMA);
var
  m1,m2,m3,m4, skew, kurtosis: Extended;
begin
  inst.getMomentSkew(fmaData, m1,m2,m3,m4, skew, kurtosis);
  fTotalStat:= fTotalStat +'  '+ floattoStr(m1) +
                                     '  '+floatToStr(m2)
  period:= format('%d %d %d %d',[cycle, perfamount, +
                                   framePCount, optMA]);
end;
```

C#-Code-Beispiel mit try-catch-Struktur:

```
public void visitElementChartMA(TElementChartMA inst) {
  double dblSkew = 0;
  double dblKurtosis = 0;
  try  {
    dblSkew = inst.getSkew(FmaData);
    dblKurtosis = inst.getKurtosis(FmaData);
  } catch (Exception objException) {
    Console.WriteLine(objException);
  }
  FTotalStatistic = FTotalStatistic + "Skew: "
     + dblSkew.ToString("F2") + "  " + "Kurtosis: " +
              dblKurtosis.ToString("F2") + "\n";
}
```

Die Elemente, die ich im Muster immer erwähne, sind konkret die wählbaren Kurven, aus denen sich ein Chart in der Objektstruktur `TChartGen` zusammensetzt:

- Kurs-Chartkurve
- Gleitender Durchschnitt (MA)
- Optimaler Moving Average (OMA)
- Buy-, Hold- oder Sell-Signale
- Bollinger Bänder, Momentum etc.

Der Aufruf der ganzen Mechanik erfolgt nun aus einem Guss aus der Objektstruktur, `TChartGen`, welche die Kontrolle der Aufrufkaskade bis am Schluss behält:

```
function TChartGen.genStatistics(const cData: TMemor;
                           const maData: TMAverage): string;
var
  visitor: TChartVisitor;
  objCData: TElementChartData;
  objCMA: TElementChartMA;
begin
  visitor:= TchartVisitor.create(cData, madata);
  objcData:= TElementChartData.create;
  objcMA:= TElementChartMA.create;
  try
    objCData.AcceptMemberVisitor(Visitor);
    objcMA.AcceptMemberVisitor(Visitor);
    result:= visitor.TotalStatistic;
  finally
    visitor.free;
    objCData.free;
```

```
    objcMA.free;
  end;
end;
```

C#-Code-Beispiel:

```
public static String genStatistics(TMemor cData,
                                   TMAverage maData) {
  TChartVisitor visitor;
  TElementChartData objCData;
  TElementChartMA objCMA;
  visitor = new TChartVisitor(cData, maData);
  objCData = new TElementChartData();
  objCMA = new TElementChartMA();
  try  {
    objCData.acceptMemberVisitor(visitor);
    objCMA.acceptMemberVisitor(visitor);
  } catch (Exception objException) {
    Console.WriteLine(objException);
  }
  return visitor.TotalStatistic;
}
```

In diesem Zusammenhang spricht man auch von der „Double-Dispatch"-Technik. Diese Technik besagt, dass die schlussendlich ausgeführte Operation von der Art der Anfrage und den Typen von zwei Empfängern abhängig ist.

Diese zwei Empfänger sind der Besucher und das Element. Werfen Sie nun einen Blick auf obigen Code und verfolgen gleichzeitig die Aufrufkaskade als Verdeutlichung des Visitors noch im Sequenzdiagramm (siehe Abb. 2.94).

Wenn also ein Element den Besucher akzeptiert (`acceptMemberVisitor`), ruft es eine entsprechende Operation im Visitor auf (`visitElementChartData`). Dabei übergibt sich das Element selbst als Parameter. Der Besucher ruft dann die zugehörige Operation für dieses Element auf (`getMeanStat`) und führt alle Operationen aus, die zu einem Gesamtergebnis beitragen (`totalStatistic`).

Verwendung

Als letztes Pattern soll mit dem Abschließen der Serie auch eine faszinierende, aber anspruchsvolle Verwendung die Krönung setzen. Stellen Sie sich die Entwicklung eines **CASE-Tool** mit Codegenerator vor, das verschiedene Operationen auf Klassenelementen (Member) ausführen muss:

- Variablen oder Elemente zeichnen
- Hilfe-Einträge der Variablen vornehmen
- Code der Variablen generieren
- Elemente in das Repository speichern

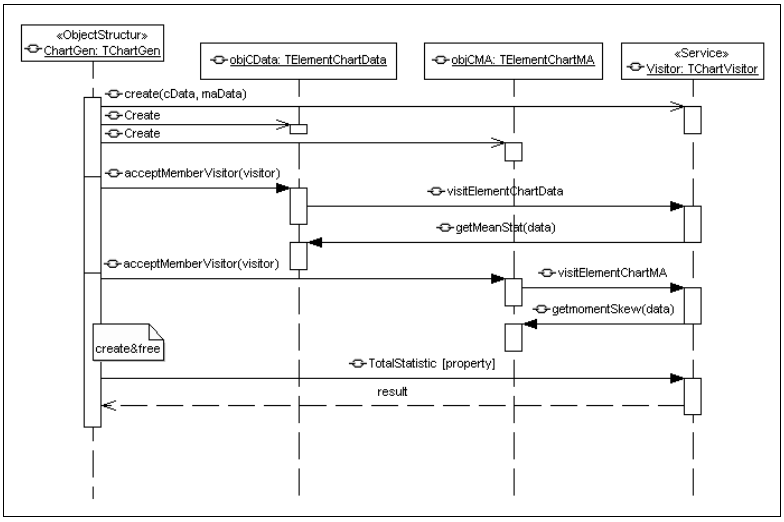


Abb. 2.94: Double-Dispatch-Technik im Visitor

Diese Struktur einer Klasse ist mit der Objektstruktur der Elemente gleichzusetzen. Das Problem ist, all diese oben erwähnten Operationen auf die einzelnen Elemente anzusetzen. Schnell wird es unübersichtlich, wenn das Zeichnen des Elementes mit der Codegenerierung oder dem Hilfetext-Eintrag vermischt ist.

Mehr noch, ein Hinzufügen einer neuen Operation muss man in jedem Element der Objektstruktur wieder implementieren, z. B. das Verschieben eines Elementes im CASE-Tool-Diagramm. Die Ausgangslage ist also, drei Elemente (siehe Abb. 2.95) zu haben, auf denen die folgenden (oder später mehr) Operationen wirken müssen (siehe Abb. 2.96).

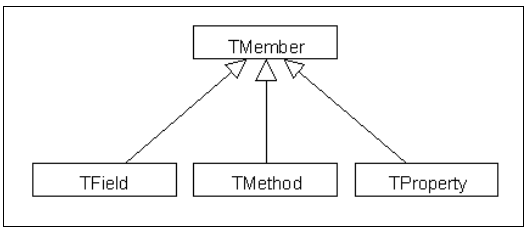


Abb. 2.95: Die drei Klassenelemente

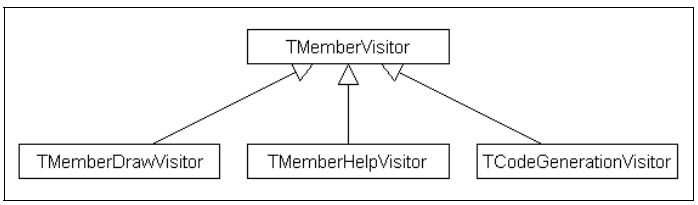


Abb. 2.96: Operationen auf den Elementen

Es ist also eindeutig besser, all die Operationen, vor allem die neuen, einer eigenen Visitor-Klasse hinzuzufügen und die Elemente dann nur noch anzufragen. Somit trenne ich die Operationen von den Elementen!

> *Zitat: „It would be better if each new operation could be added separately, and the member classes were independent of the operations that apply to them.“*[10]

Denn die meisten dieser Operationen behandeln die Elemente der Klasse sehr unterschiedlich, sei es das Zeichnen eines Propertys, einer Methode oder eines Events. Oder die Codegenerierung sieht für ein Property sicher anders aus als für ein Feld.

All diese differenten Methoden für die unterschiedlichen Elemente in einer einzigen Klasse zu implementieren, würde auch zu einer Überlastung der Klasse führen und sie schwer durchschaubar machen.

Was wir brauchen, ist jeweils **eine** Klasse **pro** Element, die sich von einer anderen Operation bearbeiten lässt, d. h., die jeweilige Operation besucht das Element der Klasse. Wir benötigen demzufolge drei Klassen (oder mehr), die zu den Elementen gehören:

- `TCodeGenerationVisitor`
- `TMemberHelpVisitor`
- `TMemberDrawVisitor`

Die vollständige Implementierung stammt von „White Ants“:

```
type
  TMember = class (TObject)
  public
    procedure AcceptMemberVisitor(Visitor: TMemberVisitor);
                                                        virtual;
  end;

  TField = class (TMember)
    procedure AcceptMemberVisitor(Visitor: TMemberVisitor);
                                                        override;
  end;
  TMethod = class (TMember)
    procedure AcceptMemberVisitor(Visitor: TMemberVisitor);
                                                        override;
  end;
  TProperty = class (TMember)
    procedure AcceptMemberVisitor(Visitor: TMemberVisitor);
                                                        override;
  end;
```

Die eigentlichen Operationen, welche die Elemente besuchen, sind von dem abstrakten Typ `TMemberVisitor` abgeleitet und lassen sich dann jeweils genau spezifizieren:

[10] ModelMaker-Help, White Ants

```
TMemberVisitor = class (TObject)
  public
    procedure VisitField(Instance: TField); virtual;
    procedure VisitMember(Instance: TMember); virtual;
    procedure VisitMethod(Instance: TMethod); virtual;
    procedure VisitProperty(Instance: TProperty); virtual;
  end;
```

Im Falle einer Codegenerierung ermöglicht nun die konkrete Klasse `TCodeGenerationVisitor` das Besuchen jedes einzelnen Elementes. Die einzelnen Elemente wie `TField` oder `TMethod` lassen sich als Objektreferenzen übergeben, das ist wie eine Adresse, die ich an einen weiteren Besucher weiterreiche:

```
Type
  TCodeGenerationVisitor = class (TMemberVisitor)
  private
    FOutput: TTextStream;
  public
    procedure VisitField(Instance: TField); override;
    procedure VisitMethod(Instance: TMethod); override;
    procedure VisitProperty(Instance: TProperty); override;
    property Output: TTextStream read FOutput write FOutput;
  end;
```

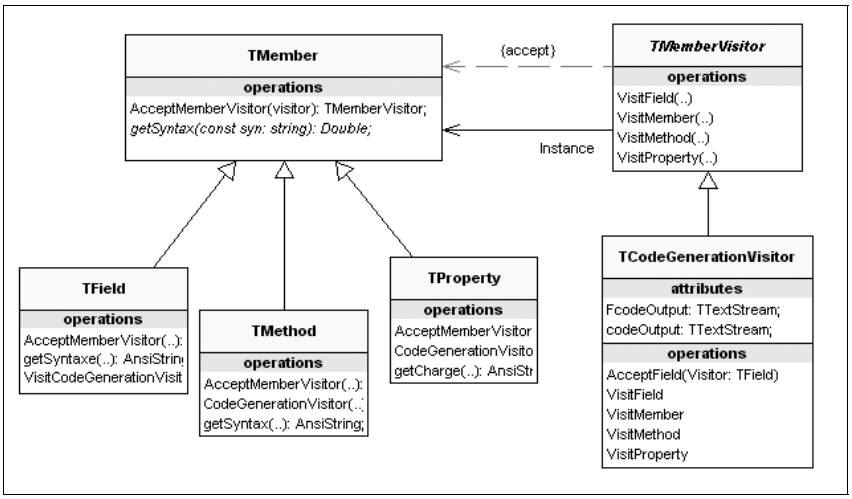


Abb. 2.97: Ein Codegenerator als Visitor modelliert

Jeder Member muss die Methode `AcceptMemberVisitor` implementieren, als OCL spezifiziert. Eine neue Methode wird durch Subclassing von `TMemberVisitor` imple-

mentiert. Im Klassendiagramm ist nur der Codegenerator sichtbar, die anderen Unterklassen sind gemäß Abb. 2.97 nicht mehr sichtbar modelliert.

Durch die abstrakte Klasse `TMemberVisitor` ist ein Output des Codegenerators in HTML durch eine weitere Unterklasse möglich. Die Elemente bleiben, solange die Typisierung nicht wechselt, unberührt.

Die einzelnen Methoden der Member (Elemente) bieten nach dem Akzeptieren des Besuchers das Weiterleiten an den Visitor an (Double Dispatch):

```
procedure TField.AcceptMemberVisitor(Visitor: TMemberVisitor);
begin
  Visitor.VisitField(Self);
end;

procedure TMethod.AcceptMemberVisitor(Visitor: TMemberVisitor);
begin
  Visitor.VisitMethod(Self);
end;
```

Die Stärke des Musters ist zusätzlich, das Traversieren der Elemente zu steuern, das beim Generieren des Codes über alle Member, sprich Elemente, nun stattfindet. Beim Beispiel des TRadeRoboter ist es ja das Erstellen der Gesamtstatistik, hier in der Verwendung wird der Code von allen Elementen generiert:

```
TCodeGenerator = class (TObject)
  public
    procedure Generate(Members: TList; Output: TTextStream);
  end;
```

Ein Auszug aus der Codeproduktion zeigt die Implementierung eines Member:

```
procedure TCodeGenerationVisitor.VisitField(Instance: TField);
begin
  Output.WriteLnFmt('  %s: %s;', [Instance.Name,
                                  Instance.DataName]);
end;
```

Bei der Generierung einer Methode ist mit einer Fallunterscheidung zuerst die Art der Methode auszumachen, sobald (auszugsweise) die Formatierung mittels des zugehörigen Streams erfolgt. Wird `F` als Text in `WriteLn` (`var F: Text`) weggelassen, so kommt die globale Variable Output ans Licht, die auf die Standardausgabedatei der Prozesse zugreift. Beim Einsatz von Output in GUI-Anwendungen muss man deshalb auf ein spezielles Vorgehen achten.

Codegeneratoren machen auch intensiv Gebrauch von Formatierungen:

```
TableName:= 'Kunden';
s:= Format('f%0:sTableAuto:=
              %0:sTableAuto.Create(f%0:Table);', [TableName]);
```

soll dann im definitiven Output ergeben:

```
s:= 'fKundenTable:= TKundenTable.Create(Kunden);'
```

Unter Windows haben die meisten Prozesse keine Standardausgabedatei und das Schreiben in Output löst einen Fehler aus. Wenn Programme als Konsolenanwendungen gelinkt sind, verfügen sie über eine Standardausgabedatei.

Unter Linux haben alle Prozesse eine Standardausgabedatei. Standardfehlerausgaben sind auf verschiedene Weisen einsehbar. Eine einfache Möglichkeit ist, das Programm aus einem Shell-Fenster aus auszuführen. Hier nun der Output:

```
procedure TCodeGenerationVisitor.VisitMethod(Instance: TMethod);
var
  MKStr, DTStr: string;
begin
  case Instance.MethodKind of
    mkConstructor: MKStr:= 'constructor';
    mkDestructor: MKStr:= 'destructor';
    mkProcedure: MKStr:= 'procedure';
    mkFuntion: MKStr:= 'function';
  end;
  if Instance.MethodKind = mkFunction then
    DTStr := ': ' + Instance.DataName
  else
    DTStr := '';
  Output.WriteLnFmt('  %s %s%s%s;'[MKStr, Instance.Name,
                                  Instance.Parameters, DTStr]);
end;
```

Schließlich gelange ich zum eigentlichen Generator, der die Aggregation aller Elemente in einer Schleife übernimmt und so den gesamten Code jedes Member erzeugen kann:

```
procedure TCodeGenerator.Generate(Members: TList;
                                          Output: TTextStream);
var
  i: Integer;
begin
  Output.WriteLine('TSample = class (TObject)');
  Visitor := TCodeGenerationVisitor.Create;
  try
  //provide context to visitor, so can handle VisitXXX methods
```

```
    for i:= 0 to Members.Count - 1 do
      TMember(Members[i]).AcceptMemberVisitor(Visitor);
  finally
    Visitor.Free;
  end;
 Output.WriteLine('end;');
end;
```

Nach diesem eleganten Abschluss und dem endlichen Ende aller Muster wechsle ich von der Anwendung des Codegenerators zu den Code und Solution Patterns, die mit kurzen Beispielen den Entwickler-Alltag bereichern sollen.

2.6 Idiome (CODESIGN)

Idiome (Idioms) sind im Grunde Code Patterns, die eine generische Struktur aufweisen und sich mehrfach in einer jeweiligen Sprache anwenden lassen. Sie geben die jeweilige Code-Struktur vor. Solution Patterns sind demgegenüber kurze und lauffähige Methoden für den Soforteinsatz. Diese Beispiele unter dem Stichwort CODESIGN stammen größtenteils aus der exzellenten Bewertung durch Bernhard Angerer von der Delphi-Site:

http://www.delphi3000.com

Solution Patterns stellen einen Versuch dar, häufige Probleme mit den häufigsten Lösungen gleichzusetzen. Idiome sind vor allem häufig verwendete „Grundgerüste“, die Sie Ihrem Quelltext hinzufügen und dann ergänzen können. Jede Sprache enthält einige Standardvorlagen, beispielsweise Vorlagen für Array-, Klassen- und Funktionsdeklarationen sowie Vorlagen für viele typische Statements.

CODESIGN ist ein Begriff aus eigener Küche, der folgende zwei Kategorien enthält:

- Idiome (Code Patterns)
- Snippets (Solution Patterns)

Als ersten Einstieg in die 12 und 16 Patterns will ich mal die beiden Sprachen Delphi und C# in einer direkten Übersicht vergleichen:

Typ	**Delphi**	**C#**
Geburt	1995, Borland, Heilsberg	2001, Microsoft, Heilsberg
Plattformen	Win32 und Linux (Kylix)	.NET
Ursprung	Pascal, Smalltalk	VCL, Java, C++
Library	VCL, CLX, JVCL	CLR, FCL
Vererbung	Einfachvererbung, Schnittstellen	Einfachvererbung, Schnittstellen
Variablen-deklaration	var a: integer; b: real; c: char; Deklaration zu Beginn	int a; float b; char c; Deklaration auch innnerhalb

Typ	**Delphi**	**C#**
Konstanten	const wert = 24;	const int wert = 24;
Bezeichner	Case Insensitiv	Case Sensitiv

Block	begin
	  ...
	end;
		{
		  ...
		}

Mainprogram	program name; const ... type ... var ..., begin ... end.	public static void Main(string[] args) { ... }

Module	Interface
	uses
	  type ...
	implementation
	uses ...
	end.
		namespace name {
		using ...
		  class ... {
		  ...
		  }
		}

Bedingung	if (a=b) or not (b<>c) then begin
	  a:=c;
	  ok:=true;
	end
	else ok:=false;
		if (a==b || !(b!=c)) {
		  a=c;
		  ok=true;
		} else
		  ok=false;

for loop	for index:=low to high do ...	for (index=low; index<=high; index++) ...

Selektion	case ordWert of
	  1: ... ;
	  2: ... ;
	  3: ... ;
	  else ... ;
	end;
		switch (wert) {
		  case 1: ... ; break;
		  case 2: ... ; break;
		  case 3: ... ; break;
		  default: ... ;
		}

Arrays	var x: array [20..1000] of real;	Float[] x = new float[20];

Funktion	function f(i: integer): integer;
	begin
	  result := i + 69;
	end
		int f(int i)
		{
		  return i + 69;
		}

Prozedur	procedure codesign(i: integer);
	begin
	  a := i + 1;
	end;
		void codesign(int i)
		{
		  a = i++;
		}

Typ	Delphi	C#

Klassen	TmyClass = class(Oberklasse)
	private
	  Index: Integer;
	public
	  constructor Create;
	  procedure just;
	end;
	
	constructor TmyClass.Create;
	begin Index:= 0; end;
	
	procedure TMyClass.just;
	begin
	  Index:=Index+1;
	end;
		class MyClass: Oberklasse {
		  int Index;
		  public MyClass() {
		  Index = 0;
		  }
		
		  void just(int element) {
		  Index++;
		  }
		}

Tab. 2.2: Vergleich Delphi – C#

2.6.1 Code Patterns

„Never change a winning team, never touch a running system."

Um Softwareteile wiederzuverwenden,[11] muss man sie zunächst finden. Damit dies überhaupt möglich ist, sind Bibliotheken, Libraries oder Repositories anzulegen. Darin befinden sich die wieder verwendbaren Teile.

Isakowitz und Kauffmann haben festgestellt, dass vorhandene Softwareteile sich nur dann wieder verwenden lassen, wenn der Aufwand dafür maximal ca. 70 % gegenüber dem Neuschreiben beträgt. Außerdem fanden sie heraus, dass in nahezu 90 % der untersuchten Fälle die verwendete Komponente oder Methode von einem Entwickler derselben Projektgruppe stammte. Dies zeigt deutlich, wie wichtig eine erfolgreiche, effektive Suche nach Code Patterns ist. Alle Beispiele sind auf der CD-ROM zu finden.

Reference Count Idiom

Das einzig offizielle Idiom fungiert unter dem Titel Reference Count. Meistens erscheint dieses Idiom kombiniert mit dem Singleton. Das Original-Zitat dazu:

„The pattern is used to control the life cycle of an object through reference counting. As long as the object is referenced it should be available. When it is no longer referenced, it should destroy itself."

```
function TInterfacedObject._AddRef: Integer;
begin
```

[11] Es gibt rund 250 Sites zu Object Pascal und Design Patterns.

```
  Result:= InterlockedIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
  Result:= InterlockedDecrement(FRefCount);
  if Result = 0 then
    Destroy;
end;
```

C#-Code-Beispiel:

```
  private void btnInstanceCreate_Click(object sender,
                    System.EventArgs e) {
    //Ausgabefeld leeren
    rtxtbAusgabe.Text = "";
    try  {
      for (long int64Counter = 0;
           int64Counter < INT64_FAKTOR; int64Counter++) {
        objALReferenceCounter.Add(new ReferenceCounter());
      }
    } catch (Exception objException) {
      Console.WriteLine(objException);
    }
    rtxtbAusgabe.Text= "Anzahl Instanzen von ReferenceCounter:
              " + ReferenceCounter.getNoReferences()
                  + "\nAnzahl Elemente in ArrayList: "
      + objALReferenceCounter.Count;
    rtxtbAusgabe.Visible = true;
  }
```

Callback Idiom

Callbacks ist eine häufig eingesetzte Technologie, die bspw. auch für Makro Interpreter oder in systemnahen Routinen ihren Einsatz findet.

```
1. TCallBackFunction = function(sig: integer):boolean;

2. function callME(sig: integer):boolean;
     implement...

3. procedure TestCallBack(myCBFunction: TCallBackFunction);
                 register; external('watchcom.dll');
```

```
4. function callMe(sig: integer): boolean;
   begin
     {whatever you need to do, case of...}
     showmessage('I was called with'+ inttostr(sig));
   end;

5. procedure TForm1.Button1Click(sender: TObject);
   begin
     testCallBack(callMe);  //subscribe function in DLL
   end;
```

Eine mögliche Verwendung ist das Ausüben eines Makros als Prozedurtyp, den man als Callback in einer Liste verwaltet:

```
Type
  Ts1s2i = function(s: string, i: integer): string;
    MyParamStrList.add(,functionName=Ts1s2i');
    MyTypeName:= myParamStrList.value[,functionName'];
```

DLL Template Idiom

```
library dlltest;

procedure <Name der Prozedur>; export;
begin
  <Quelltext der Prozedur>
end;

function <Name der Funktion>; export;
begin
  <Quelltext der Funktion>
end;

exports
<Name der Prozedur> index <Indexwert>;
<Name der Funktion> index <Indexwert>;
{$R *.RES}
begin
end.
```

Free&NIL Idiom

Die Prozedur gibt eine Objektreferenz frei und ersetzt die Referenz mit NIL (Delphi) oder mit NULL (C++).

```
Type
  TObjectPtr = ^TObject;

procedure S_FreeAndNilObject(oObj: TObjectPtr);
Begin
  oObj^.Free;
  oObj^:=NIL;
end;
```

Exception Idiom

Eine Kombination von `try..finally` und `try..except` macht vielfach Sinn:

```
var myFile :TFileSream;
    str: String;
begin
  try
    myFile:= TFileStream.create(,idiom.txt',fmOpenReadWrite);
    try
      setLength(str, myFile.size);
      myFile.read(str[1], myFile.size)
      self.caption:= str;
    finally
       freeAndNil(myFile);
    end;
  except
    on E: EWin32Error do
          bar.simpleText:= SysErrorMessage(getLastError);
  end;
end;
```

Dass der Konstruktor vor dem `try..finally` ist, hat gute Gründe. Delphi reagiert bei einer Exception im Konstruktor wie folgt:

1. Laufzeitbibliothek ruft automatisch den Destruktor auf.
2. Die Klassenreferenzvariable wird auf NIL gesetzt.
3. Die Exception wird (weiter) geworfen.

Somit kann im `finally`-Teil das `free` oder `Release` nur noch mit einer Schutzverletzung reagieren, da die Instanz in der Exception des Konstruktors schon auf NIL gesetzt wurde und nicht nochmals freigegeben werden kann.

Double Linked List Idiom

Diese klassische Struktur ist mit Knoten realisiert, hier die Struktur einer doppelt verketteten Liste als Record:

```
PMyNode = ^TMyNode;
  TMyNode = record
    Next : PMyNode;  {Link to next node in the chain}
    Prev : PMyNode;  {Link to previous node }
    Data : SomeRecord;
  end;
```

Two Dimension List Idiom

Eine effektvolle Eigenschaft der Delphi-Klassenbibliothek ist mit der Möglichkeit gegeben, an eine Liste von Einträgen beliebige Objekte mit Zusatzinformationen, die für den Benutzer nicht sichtbar sein sollen, anzuhängen.

Das folgende Beispiel zeigt eine simple Stringliste (`TListbox` oder `TCombobox` arbeiten beispielsweise mit dieser Klasse), mit deren Einträgen ein Daten-Objekt mit beliebigen Datenpunkten (beispielsweise ein String und ein Integer) assoziiert wird.

```
type
  TSLData = class
    public
    X1: integer;
    x2: string;
  // etc.
  end;
  ...
  hSL.Objects[hIndex]:= TSLData.Create;
  // hSL is from type TStringList
  TSLData(hSL.Objects[hIndex]).X1:= 4711;
  TSLData(hSL.Objects[hIndex]).X2:= 'Eau de Cologne';
```

Objects in List Idiom

Bei einem mehrfachen Instanzieren einer Klasse benötigt man zur Laufzeit Zugriff auf diese Instanzen, die sich in einer Liste verwalten lassen:

```
type
  TMiniClass = class
    MyString: string;
    constructor Create(S: string);
  end;


procedure TForm1.Button1Click(Sender: TObject);
var
  MyList: TList;
  MyObject: TMiniClass;
begin
```

```
 MyList:= TList.Create; { Liste erzeugen }
  try
    MyObject:= TMiniClass.Create('Never ending!');
    MyList.Add(MyObject);
    MessageDlg(TMiniClass(MyList.Items[0]).MyString,
            mtInformation, [mbOk], 0);
  finally
    TMiniClass(MyList.Items[0]).Free;
    MyList.Free;
  end;
```

Windows Message Idiom

Nicht visuelle Komponenten der VCL – also Klassen, die von `Tcomponent`, aber nicht von `TWinControl` abstammen – sind nicht an die Windows-Nachrichtenschleife angebunden und empfangen daher auch keine Nachrichten.

Es gibt eine einfache Möglichkeit, dies nachzubauen. Zentraler Einstiegspunkt ist der API-Befehl `AllocateHWnd`, über den eine Methode als Nachrichtenempfänger registriert werden kann und somit bei jeglichen Nachrichten angestoßen wird.

```
TMyClass.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  FHWnd:= AllocateHWnd(WndMethod);
end;

TMyClass.Destroy;
begin
  DeallocateHWnd(FHWnd);
  ...
  inherited Destroy;
end;


TMyClass.WndMethod(var Msg: TMessage);
 begin
  case Msg.Msg of
    WM_SOMETHING: DoSomething;
// Code to handle a message
```

Loaded Idiom

Bei der Entwicklung von eigenen Komponenten bzw. Controls ist die Verwendung der `Loaded` Funktion unerlässlich. Dabei handelt es sich um eine virtuelle Methode der Klas-

se `Tcomponent`, steht somit allen nicht visuellen und visuellen Komponenten vor und wird daher an den Großteil der VCL-Klassen vererbt.

Die Funktion `Loaded` befindet sich in dem Abschnitt `protected` der Klassendefinition und kann daher bei der Neuerstellung einer Kindklasse überschrieben werden. Nachdem alle Properties der entsprechenden Komponente aus der DFM-Datei gelesen werden, wird die Funktion `Loaded` angestoßen. Somit bietet sie die Möglichkeit, gewisse Werte zu setzen – beispielsweise Ergebnisse, die sich aus gewissen Abhängigkeiten heraus ergeben. In der Praxis benötigt man dieses Verhalten meist erst zur Runtime. Über das Flag `csDesigning` – wie im Codebeispiel sichtbar – lässt sich dies steuern.

```
TMyComponent = class(TComponent)
  private
  protected
    procedure Loaded; override;
  published
end;
...
procedure TMyComponent.Loaded;
begin
  inherited;
  if (not (csDesigning in ComponentState)) then begin
  // do some things like the ICE 603
  end;
end;
```

Randomstring Idiom

Eine schnelle Folge zum Erzeugen von Zufallswörtern:

```
SLen: integer; S: String;
for x:= 1 to 650000 do begin
  Slen:= Random(16)+5;
  setLength(S, Slen)
  For y:= 1 to SLen do
    S[y]:= Chr(Random(26)+ Ord(,A'));
```

Assembler Idiom

Dieses exotisch anmutende Idiom zeigt symbolisch die Optimierung von Maschinencode:

```
x := x + 1;
```

und wird zu folgendem nicht objektorientiertem Maschinencode:

```
mov  ax,  [bp-02]
inc  ax
```

```
mov  [bp-02], ax

Now, inc(i) does it this way:
inc  word ptr [bp-02]
```

2.6.2 Solution Patterns

Compare two Files

```
Function Are2FilesEqual(Const fileName1, fileName2:
                                        String ):Boolean;
var
  ms1, ms2: TMemoryStream;
begin
  Result:= False;
  ms1:= TMemoryStream.Create;
  try
    ms1.LoadFromFile(fileName1);
    ms2:= TMemoryStream.Create;
    try
      ms2.LoadFromFile(fileName2);
      If ms1.size = ms2.size then
        Result:= CompareMem(ms1.Memory, ms2.memory, ms1.size);
    finally
      ms2.free;
    end;
  finally
    ms1.free;
  end
end;
```

Get File List

Im folgenden Beispiel wird ein `TStringList`-Objekt erstellt und mit den Namen der Dateien im übergebenen Pfad gefüllt:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result:= TStringList.Create;
  try
    I:= FindFirst(Path, 0, SearchRec);
```

```
    while I = 0 do begin
      Result.Add(SearchRec.Name);
      I:= FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

In dieser Funktion wird ein `TStringList`-Objekt erstellt und mithilfe der Funktionen `FindFirst` und `FindNext` (Unit `SysUtils`) mit Werten gefüllt. Tritt dabei ein Fehler auf (z. B. aufgrund eines ungültigen Pfades oder wegen Speichermangels), muss man das neue Objekt freigeben, da es der aufrufenden Routine noch nicht bekannt ist.

Aus diesem Grund muss die Initialisierung der Stringliste in einer `try...except`-Anweisung durchgeführt werden. Bei einer Exception wird das Objekt im Exception-Block freigegeben und anschließend die Exception erneut ausgelöst.

Save File to a Blob

```
blob := yourDataset.CreateBlobStream(yourDataset.
                          FieldByName('YOUR_BLOB'),bmWrite);
try
  blob.Seek(0, soFromBeginning);
  fs:= TFileStream.Create('c:\your_name.doc', fmOpenRead or
                                    fmShareDenyWrite);
  try
    blob.CopyFrom(fs, fs.Size)
  finally
    fs.Free
  end;
finally
  blob.Free
end;

//To load from BLOB:
blob := yourDataset.CreateBlobStream(yourDataset.
                          FieldByName('YOUR_BLOB'),bmRead);
try
  blob.Seek(0, soFromBeginning);
  with TFileStream.Create('c:\your_name.doc', fmCreate) do
    try
      CopyFrom(blob, blob.Size)
    finally
      Free
```

```
    end;
finally
  blob.Free
end;
```

User Permissions

RTTI (Runtime Type Information) ist das Zauberwort, welches folgende Codezeilen ermöglicht. In Delphi haben alle Objekte, welche von `TPersistent` erben, bestimmte Informationen über sich selbst gespeichert. So kann man über den Operator `is` abfragen, um welchen Typ es sich handelt, und so eine allgemeine Schleife bauen, die zum Beispiel alle `TmenuItem`-Objekte ausliest und über einen Typecast das Property `Visible` setzt.

```
for i := 0 to ComponentCount -1 do
  if Components[i] is TMenuItem then
    TMenuItem(Components[i]).Visible:=
    (UserLevel >= TMenuItem(Components[i]).Tag);
```

Query Search

```
function SeqSearch(AQuery: TQuery; AField, AValue: String):
                       Boolean;
  begin
    with AQuery do begin
      First;
      while (not Eof) and (not (FieldByName(AField).
            AsString = AValue)) do
        Next;
      SeqSearch := not Eof;
    end;
  end;
```

Get Records Independent

Eine sehr einfache, jedoch wirksame Methode ist es, oft benötigte Befehle in einer Routine zusammenzufassen. Beispielsweise muss ich immer wieder die Befehle `Close`, `Clear`, `Add` und `Open` der `Tquery`-Objekte ausführen. Wenn ich diese wie in dem hier gezeigten Beispiel in einer Funktion zusammenfasse, so kann ich nachträglich leicht über mein gesamtes Projekt hinweg eine Einstellung ändern oder eine bestimmte Exception abfangen, um darauf im speziellen Maße zu reagieren.

```
Procedure GetRecords(var qryField: TADOQuery; sSQL: String;
                       LockType: TADOLockType = ltReadOnly);
begin
  if qryField = Nil then begin
```

```
    qryField:= TADOQuery.Create(Application);
    qryField.CursorType:= ctOpenForwardOnly;
    qryField.Connection:= adoConnection;
  end;
  qryField.Close;
  qryField.LockType:= LockType;
  qryField.SQL.Clear;
  qryField.SQL.Add(sSQL);
  qryField.Open;
end;
```

Find all Links

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  for i:= 0 to Webbrowser1.OleObject.Document.links.length-1 do
  begin
    Listbox1.items.add(Webbrowser1.OleObject.
    Document.links.item(i));
  end;
end;
```

Is Internet Connection

```
uses
  StdCtrls,registry;

Function IsConnected: Boolean;
var
  reg: TRegistry;
  buff: dword;
begin
  reg:= tregistry.Create;
  Reg.RootKey:=HKey_local_machine;
  if reg.OpenKey('\System\CurrentControlSet\Services\
                      RemoteAccess',false) then begin
    reg.ReadBinaryData('Remote Connection',buff,sizeof(buff));
    result:= buff = 1;
    reg.CloseKey;
    reg.Free;
  end;
end;
```

Iterate and count Controls

Protokollieren aller Komponenten auf einem Form in eine Datei:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
logList: TStringList;
begin
logList: TStringList.create;
  for I:= 0 to ComponentCount -1 do begin
  {if Components[I] is TButton then
     TButton(Components[I]).Font.Name := 'Courier';
     Edit1.Text := IntToStr(ComponentCount) + ' components';}
     myList.Add(intToStr(i) + ' count');
     myList.Add(components[i].name);
   end;
  myList.SaveToFile('LOGLIST.TXT');
end;
```

Component Streaming

Der eingebaute Streaming-Mechanismus der VCL macht es leicht, eine gesamte Komponente in einen Memory-Stream zu verpacken, um es beispielsweise in den Datentyp `Variant` zu konvertieren (und wieder zurück). Der Datentyp `Variant` wurde mit Delphi 4 hauptsächlich aus Kompatibilitätsgründen zu COM eingeführt. Somit kann ein Objekt über ein late-binding COM oder DCOM Interface übergeben werden.

```
function ComponentToVariant(AComponent: TComponent): Variant;
var
  BinStream: TMemoryStream;
  Data: Pointer;
begin
  BinStream:= TMemoryStream.Create;
  try
    BinStream.WriteComponent(AComponent);
    result:= VarArrayCreate([0, BinStream.Size-1], varByte);
    Data:= VarArrayLock(result);
    try
      Move(BinStream.Memory^, Data^, BinStream.Size);
    finally
      VarArrayUnlock(result);
    end;
  finally
```

```
    BinStream.Free;
  end;
end;
```

Set All Properties

Um alle Komponenten zur Laufzeit mit einer Eigenschaftsänderung zu beglücken:

```
Procedure TForm1.SetProperties(ClassName,SomeProperty:
                                String; Value: boolean);
var
 i: integer;
 PropInfo: PPropInfo;
 Component: TComponent;
begin
  for i:=0 to ComponentCount-1 do begin
    Component:= Components[i];
    if (Component is TControl) and ((Component.ClassName
              = ClassName) or(ClassName ='all')) then begin
      PropInfo:= GetPropInfo(Component.ClassInfo,SomeProperty);
      if Assigned(PropInfo)and
       (PropInfo^.PropType^.Kind = tkEnumeration) then
         SetOrdProp(Component, PropInfo, integer(Value));
    end;
  end;
end;

procedure TForm1.Button1Click(Sender:TObject);
begin
//Make all components readonly
  SetProperties('all','readonly',true);
//Make all components of Class TEdit invisible
  SetProperties('TEdit','visible',false);
//Set ShowHint to false for allcomponents
  SetProperties('all','ShowHint',false);
end;
```

Broadcast Method

Jedes Delphi-Formular verfügt über eine Methode namens `Broadcast`, über die eine Nachricht an alle Controls dieses Formulars versendet werden kann. Mit Controls sind in diesem Fall alle Komponenten gemeint, welche in der Liste der Komponenten (`ComponentCount`) mithilfe des `Owner`-Parameters eingetragen sind. Die Struktur `TMessage` bildet eine typische Windows-Nachricht ab und muss als entsprechender Pa-

rameter übergeben werden. Das Schöne an dieser Lösung ist die Unabhängigkeit, die dabei erreicht wird.

Es liegt nämlich im Ermessen jeder einzelnen Objekte, darauf zu reagieren oder nicht. In unserem Beispiel implementiert die Klasse `TMyButton` eine Methode (EventHandler), welche durch die entsprechende Nachricht angestoßen wird. Es wird die Schaltfläche über eine Zustandsänderung auf diesem Wege informiert.

```
var i: integer;
  hMessage: TMessage;
begin
  hMessage.Msg:= WM_USER + 1;
  hMessage.WParam:= 0;
  hMessage.LParam:= 0;

  for i:= 0 to Screen.FormCount-1 do
    Screen.Forms[i].Broadcast(hMessage);
end;
...
// This is a event-handler:
TMyButton = class(TButton)
  protected
    procedure EventHandler(var Message: TMessage);
    message WM_USER + 1;
end;
...
procedure TMyButton.EventHandler(var Message: TMessage);
begin
  // commands
  Message.Result:= 0; //Event continues
end;
```

Check DLL Function

Prüfen, ob eine Funktion in einer DLL vorhanden ist:

```
function FuncAvail (VLibraryname, VFunctionname: string;
                       var VPointer: pointer): boolean; var
  Vlib: tHandle;
begin
  Result:= false;
  VPointer:= NIL;
   if LoadLibrary(PChar(VLibraryname)) = 0 then
      exit;
   VPointer:= GetModuleHandle(PChar(VLibraryname));
   if Vlib <> 0 then begin
```

```
    VPointer:= GetProcAddress(Vlib, PChar(VFunctionname));
    if VPointer <> NIL then
      Result:= true;
  end;
end;
```

PChar Iterator

Wir iterieren durch die null-terminierten Daten durch, bis ein doppeltes Nullzeichen erscheint. Die einzelnen Einträge sind ebenfalls mit Nullzeichen terminiert, deshalb hat das Ende ein zweifaches Nullzeichen. `StrPas` liefert dann alle Zeichen bis zum nächsten Null. `StrPas` ist nur aus Gründen der Abwärtskompatibilität vorhanden. Um einen null-terminierten String in einen AnsiString oder einen nativen Delphi-String zu konvertieren, verwenden Sie eine Typumwandlung oder eine Zuweisung.

```
Procedure IteratePChar(vpBuffer: PChar);
var pSavePuffer: PChar;
begin
  pSavePuffer:= vpBuffer;
  listbox2.clear;
  repeat
    listbox2.items.add(StrPas(vpBuffer));
    inc(vpBuffer, strLen(vpBuffer)+1);
  until vpBuffer^=#0;
  freeEnvironmentStrings(pSaveBuffer);
end;
```

Is Form Open

Die Klasse `Screen` wird von Delphi-Entwicklern oft übersehen. Sie ist bei gestartetem Projekt immer instanziert und gibt Auskunft über den Bildschirm und die Anzahl aller Formulare des aktuellen Projekts. So können Funktionen entwickelt werden, die über alle Masken hinweg bestimmte Einstellungen vornehmen.

```
function IsFormOpen(const FormName: string): Boolean;
var
  i: Integer;
begin
  Result:= False;
  for i:= Screen.FormCount - 1 DownTo 0 do
    if (Screen.Forms[i].Name = FormName) then begin
      Result:= True;
      Break;
    end;
end;
```

Is App Running

Als Letztes sei erwähnt, dass Delphi eigentlich AppBuilder heißt ;):

```
function IsDelphiRunning: boolean;
begin
  Result:= (FindWindow('TAppBuilder',nil) <> 0);
end;
```

2.6.3 Suchtechniken für Code Patterns

Wie leicht zu sehen ist, stellen sich hierbei zwei entgegengesetzte Anforderungen: Einerseits muss die Suchroutine möglichst alle infrage kommenden Komponenten und Idiome finden, da man sie sonst auf keinen Fall verwendet. Somit erfolgen eher zu viel als zu wenig Ergebnisse. Andererseits dauert die Identifikation umso länger, je mehr Objekte die Routine gefunden hat. Außerdem müssen die Angaben ausreichend für eine abschließende Entscheidung sein, Nachfragen, zum Beispiel beim Entwickler der Patterns oder Komponente, sind nicht erwünscht. Wie lässt sich dieser Zielkonflikt lösen?

Hierarchische Klassifikation

Eine Möglichkeit, Idiome gezielt zu finden, besteht darin, sie systematisch zu ordnen. Dabei wird von einer Startebene nach und nach zu niedrigeren verzweigt, bis das gesuchte Objekt gefunden ist oder ein leeres Ende als Ergebnis erscheint.

In der Theorie und simplen Fällen funktioniert dieses Verfahren gut. Bei spezielleren Software-Komponenten wird es allerdings schwer, diese konkret einer Klasse zuzuordnen. Es gibt die Möglichkeit, sie dann in alle möglicherweise passenden Kontexte einzuordnen. Damit wird das Verfahren aber mäßig effektiv, da nun zu viele nicht optimal passenden Komponenten gefunden werden, ohne dass man den Benutzer darauf hinweist.

Suchsystem

Im Gegensatz zu der hierarchischen Klassifikation führt hier eine Software die Suche im Datenbestand durch. Dafür bieten sich verschiedene Techniken an:

- Arten der Klassifikation und Suche
- Strukturierte Klassifikation nach Baum
- Ähnliche Konzepte und Kontext nach Kategorisierung
- Schlagworte (keywords) und Schlüsselwörter

Hierbei handelt es sich um eine weitere Technik, die den Wortumfang der Suche begrenzt und somit versucht, eine eindeutigere Klassifikation mit Filtern zu ermöglichen. Den Objekten werden bestimmte Schlagworte zugeordnet.

Eine Suchanfrage bezieht sich dann auf diese Schlagworte. Im Gegensatz zur strukturierten Klassifikation stehen die Schlagworte allerdings im freien Raum und sind nicht in bestimmte Bereiche geordnet.

Volltextsuche

Bei der Volltextsuche (full text retrival) wird der Sourcecode oder der Beschreibungstext einer Komponente nach bestimmten Schlagwörtern oder Texten abgesucht. Gerade die Suche im Sourcecode kann sehr effektiv sein, wenn dieser gut dokumentiert ist. So schlägt der Entwickler zwei Fliegen mit einer Klappe: Indem er seinen Code dokumentiert, lässt dieser sich einerseits leichter erweitern oder ändern und andererseits auch gleich zur Wiederverwendung nach der erfolgreichen Suche bereitstellen.

2.7 Pattern: Markt und Links

Es gibt fantastische Sites zu Entwurfsmustern in diversen Sprachen und Kulturen: Ein Katalog mit allen gängigen GoF Patterns mit UML-Diagrammen und Sourcecode gibt es sicher verstreut über alle der unten erwähnten Sites, keiner ist aber vollständig. Wie auch: Das Pattern-Universum ist groß und mit den ausgewählten Links sind meist auch weitere Links oder Tutorials zu finden:

Schwerpunkt	**Link**	**Downloads**
OO Design	http://ooTipps.org/ood-principles.html/	Als Portal
Patterns in C#	www.dofactory.com/patterns/patterns.asp/	Ja
Standard Patterns	www.hillside.net/patterns/	Ja
Design Patterns / Kylix	www.delphix.de	Ja
Pattern Stories	wiki.cs.uiuc.edu/PatternStories/DesignPatterns	Nein
Pattern Explorer	www.manarko.com/patterns0.htm	Ja
Patterns in Java	www.designpatterns.ch/patterns.html	Nein
Experten in Delphi	www.delphi-jedi.org/	Ja
OO Patterns Generell	www.cetus-links.org/oo_patterns.html	Als Portal
Patterns Central	http://www.patternscentral.com	Als Portal
Pattern Articles	www.burn-rubber.demon.co.uk/patterns.htm	
Pattern Collection	http://www.atug.com/andypatterns/	

Tab. 2.3: Patterns im Web

[i] [DP95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable OO-Software, Addison-Wesley, 1995

[ii] R. Helm, R. Johnson, J. Vlissides: Auffinden von wiederverwendbaren Objekten, Addison-Wesley, Forth printing 1995

[iii] DWP: www.dwp42.org oder sourceforge.net/projects/dwpl

[iv] Bruce Schneier: Angewandte Kryptographie, Addison-Wesley, 1996

[v] www.omg.org/technology/documents/

[vi] Wolfram Stephen: A New Kind of Science

[vii] Wirth Niklaus: Grundlagen und Techniken des Compilerbaus

[viii] http://www.delphi-jedi.org/

[ix] www.heimetli.ch, SMS-DLL

[x] De Marco, Tom: Structured Analysis and Design Methodology; Prentice Hall, 1979