Objektorientierte Softwareentwicklung nach WAM mit SQLWindows

Diplomarbeit
aus dem Fachbereich Informatik der Universität Hamburg,
Arbeitsbereich Softwaretechnik

Betreuer: Prof. Dr. Heinz Züllighoven
Prof. Dr. Winfried Lamersdorf

vorgelegt von: Holger Koschek
Matr.-Nr. 4603330
Gartenstr. 128
60596 Frankfurt am Main

Hamburg / Frankfurt am Main
September 1997

Inhaltsverzeichnis

Vorwort

Diese Diplomarbeit entstand aus der Zusammenarbeit des Softwaretechnik-Centers der Universität Hamburg mit der Firma IVE.

IVE setzt für die Realisierung seiner Softwareentwicklungsprojekte das Client/Server-Entwicklungssystem SQLWindows ein. Um die am Arbeitsbereich Softwaretechnik der Universität Hamburg entwickelte Softwareentwicklungsmethode WAM (für: Werkzeug-Aspekt/Automat-Material) nutzen zu können, die auf dem objektorientierten Modell beruht, mußte untersucht werden, ob bzw. inwieweit SQLWindows als objektorientierte Programmiersprache bzw. objektorientiertes Entwicklungssytem betrachtet werden kann. Da sehr schnell deutlich wurde, daß SQLWindows nur mit Einschränkungen als objektorientiert bezeichnet werden kann, war ferner zu zeigen, daß mit Hilfe einer Klassenbibliothek oder eines Frameworks die Schwächen von SQLWindows umgangen werden können – mit dem Ziel, eine objektorientierte Softwareentwicklung nach WAM möglich zu machen. Diese Untersuchung und die Konstruktion eines solchen Frameworks sind Gegenstand dieser Diplomarbeit.

Ich danke Herrn Prof. Dr. Heinz Züllighoven und Herrn Prof. Dr. Winfried Lamersdorf für die Betreuung dieser Arbeit. Mein besonderer Dank gilt Wolfgang Strunk, der mir jederzeit mit Rat und Tat zur Seite stand und immer im rechten Augenblick motivierend auf mich einzuwirken vermochte.

Ferner danke ich Herrn Geerken von der Centura Software GmbH, der mir mit einer unbürokratisch zur Verfügung gestellten SQLWindows-Lizenz die nötige technische Unterstützung gewähren konnte.

Herrn Fischer und Herrn Werkhäuser von der E.I.S. Konzept GmbH ist es zu verdanken, daß ich eine Phase des Zweifelns meistern konnte, in der ich mich fragte, ob man mit SQLWindows tatsächlich objektorientierte Software entwickeln kann. Die kritische Diskussion und nicht zuletzt das Produkt, der E.I.S. Application Builder, gaben mir die nötigen Ideen und Impulse.

Hamburg / Frankfurt am Main, im September 1997

Holger Koschek

1 Einleitung

Ziel dieser Diplomarbeit ist es, die Eignung des Client/Server-Entwicklungssystems SQLWindows mit seiner Programmiersprache SAL (SQLWindows Application Language) für die Umsetzung der am Arbeitsbereich entwickelten objektorientierten Softwareentwicklungs-Methode WAM (für: Werkzeug-Aspekt/Automat-Material) zu untersuchen. SAL soll hier als Beispiel gelten für eine datenbankorientierte Programmiersprache der 4. Generation.

Von SQLWindows wird vom Hersteller behauptet, es sei objektorientiert – eine sinnvolle Voraussetzung für die Softwareentwicklung nach WAM. Objektorientierung ist heute in aller Munde. Viele Unternehmen und Forschungseinrichtungen erkennen das objektorientierte Programmier-Paradigma als die zur Zeit beste softwaretechnische Lösung an. So wurde z.B. 1992 von Apple und IBM die Firma Taligent gegründet mit dem Ziel, einen Ausweg aus der Misere zu finden, die gemeinhin als „Software-Krise“ bezeichnet wird:

- Die Softwareentwicklung dauerte zu lange.
- Die Wartung der Software war zu aufwendig.
- Die Software war nur schwer zu erlernen und zu benutzen.

Taligent wollte mit Hilfe der objektorientierten Programmierung diese Probleme in den Griff bekommen (vgl. [CP95]). Die Firma begann mit der Entwicklung einer Sammlung von objektorientierten Frameworks, die nahezu alle Anwendungsgebiete der Softwareentwicklung abdecken sollten. Auch wenn Taligent letztendlich an der Komplexität dieser Aufgabe scheiterte, so wurde doch deutlich, daß die objektorientierte Programmierung zur Realisierung größerer softwaretechnischer Projekte geeignet ist und Ziele wie Änderbarkeit, Erweiterbarkeit und Wiederverwendbarkeit unterstützt.

Die Idee der objektorientierten Programmierung hat ihren Ursprung in den 60er Jahren mit dem Simula-Projekt. Die objektorientierte Programmierung hat demnach eine lange Geschichte aufzuweisen und ist im Laufe der Zeit zu einer praktikablen Technik gereift, was viele Projekte in der Praxis beweisen. Im Laufe der Zeit haben sich mit C++, Smalltalk und Eiffel drei objektorientierte Programmiersprachen etabliert, eine vierte – Java – darf sich gerade in der Praxis beweisen.

Neben diesen „klassischen“ objektorientierten Programmiersprachen nehmen in letzter Zeit auch Programmiersprachen der 4. Generation (4GL, für: 4th Generation Language) objektorientierte Konzepte auf. Eine dieser Sprachen ist SAL, die Programmiersprache von SQLWindows. Im

Rahmen dieser Diplomarbeit wird untersucht, inwieweit die am Arbeitsbereich entwickelte objektorientierte Softwareentwicklungs-Methode WAM mit SQLWindows realisiert werden kann. Die Methode stellt bestimmte Anforderungen an eine objektorientierte Programmiersprache, um den fachlichen Entwurf möglichst ohne Bruch in einen technischen Entwurf überführen zu können.

Eine solche Untersuchung kann nur dann sinnvoll durchgeführt werden, wenn der Kontext der Untersuchung klar definiert ist. Abbildung 1.1 stellt den Kontext dar, in dem sich diese Diplomarbeit bewegt.

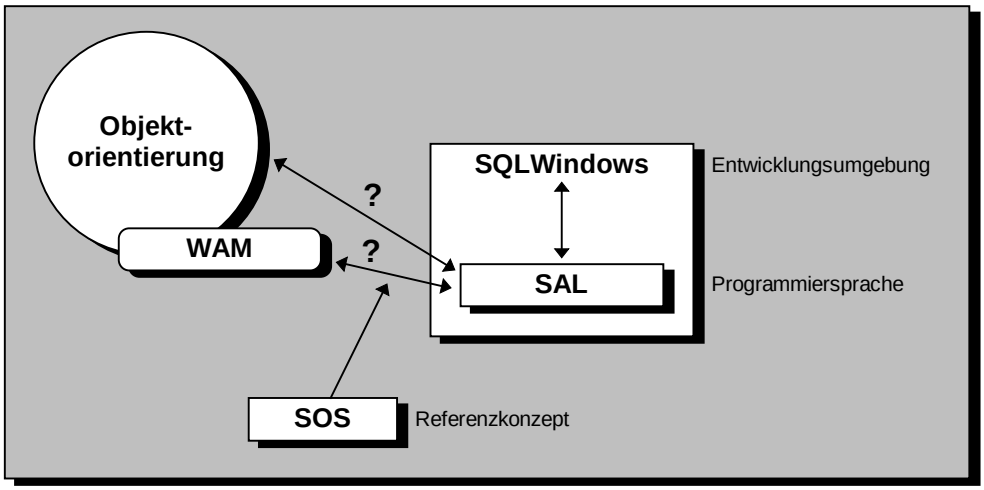
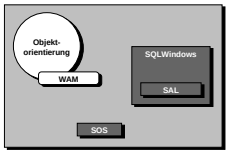
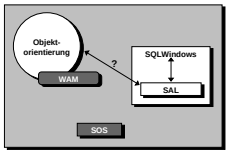


Abbildung 1.1: Der Kontext dieser Diplomarbeit

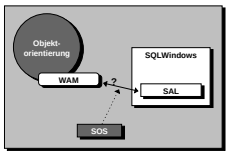
Die Bereiche des Kontextes lassen sich grob den einzelnen Kapiteln dieser Arbeit zuordnen. Als Orientierungshilfe für den Leser wird am Anfang eines jeden Kapitels – wie auch in der folgenden Beschreibung der Gliederung – die Einordnung des Kapitels in den Kontext dieser Arbeit aufgezeigt.



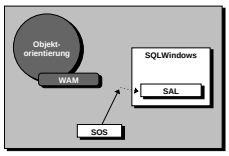
Bevor SAL auf seine objektorientierten Konzepte hin untersucht werden kann, muß zunächst ein Verständnis vom Begriff der „Objektorientierten Programmiersprache“ geschaffen werden. In der Literatur gibt es viele Definitionen von Objektorientierung, von denen einige in **Kapitel 2** beschrieben werden. Ein besonderes Augenmerk liegt hier auf dem im WAM-Kontext vertretenen Verständnis von Objektorientierung. Auf Grundlage dieser Definitionen wird ein Dimensionskatalog für objektorientierte Programmiersprachen entwickelt, der in **Kapitel 3** beschrieben ist.



In **Kapitel 4** wird die Sprache SAL und ihre Einbindung in das SQLWindows-Entwicklungssystem beschrieben. SAL ist eng an SQLWindows gebunden. So wird die Struktur eines SQLWindows-Programmes durch die Entwicklungsumgebung vorgegeben. Die prozeduralen Eigenschaften von SAL werden beschrieben und vor dem Hintergrund der objektorientierten Programmierung bewertet. Anschließend werden die objektorientierten Eigenschaften von SAL an dem oben erwähnten Dimensionskatalog gemessen, um schließlich beurteilen zu können, ob bzw. inwieweit SAL eine objektorientierte Programmiersprache ist.



Neben der Untersuchung der objektorientierten Konzepte von SAL ist es das Hauptziel dieser Arbeit, eine Klassenbibliothek oder ein Framework für SQLWindows zu entwickeln, das die objektorientierte Softwareentwicklung nach WAM ermöglicht. Da sehr schnell deutlich wird, daß SQLWindows nur mit Einschränkungen als objektorientiert bezeichnet werden kann, muß damit gerechnet werden, daß das objektorientierte Repertoire von SAL nicht ausreicht, um alle Aspekte von WAM bestmöglich umzusetzen. Hier kann – neben eigenen Erweiterungen des SQLWindows-Systems – nur eine durchgängig strukturierte Dokumentation dabei helfen, den technischen Entwurf wartbar und verständlich zu halten. Die Beschreibung der WAM-Metaphern und des daraus entwickelten WAM-Frameworks sowie Hinweise zur Benutzung dieses Frameworks bilden den Inhalt von **Kapitel 5**.



Das SQLWindows Object System (SOS), das in **Kapitel 6** beschrieben wird, soll den fehlenden Referenzmechanismus in SQLWindows weitestmöglich kompensieren. Neben einem Beispiel enthält dieses Kapitel auch Anmerkungen zu Problemen, die während des Entwurfs zu lösen waren. Das SOS steht exemplarisch für die Erweiterung einer Programmiersprache um neue Sprachkonzepte.

In **Anhang A** finden sich die Klassenbeschreibungen des WAM-Frameworks.

Anhang B enthält die Klassen- und Funktionsschnittstelle des SOS.

2 Der Begriff der "Objektorientierten Programmiersprache"

Die SQLWindows Application Language (SAL) wird vom Hersteller Gupta[1] als objektorientierte Programmiersprache der vierten Generation bezeichnet und wartet mit einem Katalog von Eigenschaften auf, die den objektorientierten Charakter dieser Sprache ausmachen sollen. Diese Eigenschaften werden in Kapitel 4 beschrieben und untersucht. Hier soll es zunächst darum gehen, ein eigenes Bild des Begriffs der „Objektorientierten Programmiersprache“ zu prägen, das zum einen bestimmt ist durch die Definitionen, die man in der Literatur findet, und zum anderen durch die Anforderungen, die von der WAM-Methodik an eine objektorientierte Programmiersprache gestellt werden (die WAM-Methodik wird in Kapitel 5 beschrieben). Das aus diesen Definitionen und Anforderungen gewonnene Verständnis vom Begriff der „Objektorientierten Programmiersprache“ bildet die Grundlage für den Dimensionskatalog, der in Kapitel 3 beschrieben ist, und an dem SQLWindows im folgenden gemessen wird.

In der Literatur finden sich viele Definitionen für den Begriff der „Objektorientierten Programmiersprache“. Bei eingehender Analyse läßt sich feststellen, daß die meisten Definitionen im Kern äquivalent sind und sich vor allem in der Anzahl der Dimensionen unterscheiden, die eine objektorientierte Programmiersprache ausmachen sollen. Einige Autoren differenzieren zwischen notwendigen Dimensionen einerseits und erweiternden, nicht-essentiellen Dimensionen andererseits.

Ein weiteres Unterscheidungsmerkmal der verschiedenen Definitionen ist deren Nähe zu programmiersprachlichen Konzepten und Begriffen. Die sprachunabhängigste und umfassendste der hier untersuchten Definitionen stammt von Madsen und Møller-Pedersen und ist in Abschnitt 2.1 beschrieben. Sie läßt sich nicht nur auf die objektorientierte Programmierung anwenden, sondern

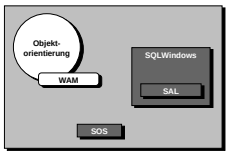
[1] Die Firma Gupta Corporation wurde im Frühjahr 1996 in Centura Software Corporation umbenannt. In dieser Arbeit wird der alte Firmenname Gupta verwendet, da das Produkt SQLWindows mit diesem Firmennamen verbunden ist.

spannt den Bogen von der Analyse über den Entwurf bis hin zur Programmierung, indem sie beschreibt, wie man zu einem Modell des Anwendungsgebietes gelangt, das schließlich als Grundlage für die Programmierung dient.

In Abschnitt 2.2 wird eine technischere Sicht auf die objektorientierte Programmierung dargestellt. Hier werden die von Wegner eingeführten Dimensionen objektorientierter Programmiersprachen beschrieben, ergänzt um Eigenschaften, die Meyer von objektorientierten Programmiersprachen fordert. Diese Dimensionen werden abschließend mit den bei Madsen und Møller-Pedersen geforderten Eigenschaften zusammengefaßt.

In Abschnitt 2.3 wird schließlich die objektorientierte Programmierung aus Sicht der WAM-Methode beschrieben, an der letztendlich die Programmiersprache SAL gemessen werden soll.

Dieses Kapitel stellt verschiedene Definitionen des Begriffs der „Objektorientierten Programmiersprache" vor, wobei jede Definition unterschiedliche Begriffe mit einer bestimmten Bedeutung belegt. Einige dieser Begriffe werden auch in der WAM-Methodik verwendet, haben aber dort oft eine andere Bedeutung. Um den mit der WAM-Methodik vertrauten Leser vor Mißverständnissen zu schützen, wird bei der Definition derartiger Begriffe jeweils auch deren Bedeutung im WAM-Kontext genannt.



Dieses Kapitel liefert die Grundlagen für eine Definition des Begriffes der „Objektorientierten Programmiersprache" im Kontext dieser Diplomarbeit, wobei ein besonderes Augenmerk auf dem im WAM-Kontext vertretenen Verständnis von Objektorientierung liegt.

2.1 Objektorientierte Programmierung als Modellbildung

Objektorientierte Programmierung wird von Madsen und Møller-Pedersen in [MMP88] als eine Art der Modellbildung verstanden:

> „Objektorientierte Programmierung: Die Ausführung eines Programmes wird als ein *physisches Modell* verstanden, welches das Verhalten eines realen oder imaginären Ausschnitts der Welt simuliert."

Physische Modelle basieren auf Erkenntnissen der Realität in Form von *Phänomenen* und *Begriffen*. Modellbildung ist dann der Prozeß der Identifikation relevanter und interessierender Phänomene und Begriffe. Welche Phänomene und Begriffe nun relevant bzw. notwendig oder hinreichend für das zu entwickelnde physische Modell sind, ist abhängig von der Modellklasse, der das Modell zugeordnet werden kann – in diesem Falle der Modellierung von Informationsprozessen.

Interessant an dieser Definition ist vor allem, daß im Unterschied zu anderen Definitionen die *Ausführung* eines Programms als Modell betrachtet wird, und nicht etwa das Programm selbst. Hier stellt sich nun die Frage, was das Wesen eines Modells von einem (realen oder imaginären) Ausschnitt der Welt ist: Die Elemente (Objekte) und die statischen Beziehungen zwischen diesen, oder die Kooperation und Kommunikation zwischen den Objekten und ihre dynamischen Beziehungen. Erstere Sicht läßt sich in einem Programm modellieren, während die zweite Sichtweise eben erst in der Ausführung eines Programms deutlich wird. Dieser Gedanke verdient sicherlich eine gewisse Beachtung, soll hier aber nicht weiter vertieft werden, da dies den Rahmen der Diplomarbeit sprengen würde.

2.1.1 Ein Beispielmodell

Als Beispiel für einen zu modellierenden Ausschnitt aus der realen Welt soll der Bürobereich dienen. Kilberth et. al. nutzen in [KGZ94] ebenfalls diesen Anwendungsbereich zur Illustration der objektorientierten Softwareentwicklung unter Benutzung der WAM-Methodik. Die konsistente Benutzung eines Beispiel-Anwendungsbereiches vereinfacht den Vergleich der unterschiedlichen Sichtweisen von Objektorientierung.

Zu den typischen Gegenständen eines Büroarbeitsplatzes gehört ein Ordner. Mit einem konkreten Ordner verbindet man sowohl einen allgemeinen Begriff als auch ein bestimmtes Verhalten oder eine bestimmte Umgangsform, die das Wesen eines Ordners ausmachen. So kann man z.B. aus einem Ordner etwas entnehmen, etwas in ihn einfügen oder ihn beschriften.

Im folgenden wird nun erarbeitet, wie man einen Ordner bei der Modellierung entsprechender Informationsprozesse in Form von Phänomenen und Begriffen beschreiben kann.

2.1.2 Phänomene

Im Kontext von Informationsprozessen wird ein *Phänomen* durch folgende drei Aspekte definiert:

- *Substanz* bezeichnet die physische Dimension, ausgedrückt durch das Volumen und eine Position in Raum und Zeit.
- *Meßbare Eigenschaften* der Substanz.
- *Transformationen*: Partiell geordnete Folgen von Ereignissen, welche die meßbaren Eigenschaften der Sustanz verändern.

Die Substanz eines konkreten Ordners „Rechnungen A-B" ist seine materielle Existenz an einem bestimmten Ort, wie z.B. einem Aktenschrank für Rechnungsordner. Meßbare Eigenschaften sind z.B. seine Farbe, die Herstellerfirma, seine Beschriftung, sein Inhaltsverzeichnis und sein Inhalt, bestehend aus alphabetisch nach Kunden und Datum sortierten Rechnungen, wobei die einzelnen Kunden durch Zwischenblätter voneinander getrennt sind. Es kann eine Rechnung in den Ordner eingefügt oder eine Rechnung aus dem Ordner entnommen werden. Ferner kann man den Ordner beschriften und das Inhaltsverzeichnis führen. Das Einfügen einer neuen Rechnung in den Ordner sei immer mit einem entsprechenden Eintrag im Inhaltsverzeichnis verbunden. Damit ist eine Folge von Ereignissen definiert, welche die Transformation „Einfügen einer neuen Rechnung in den Ordner" beschreibt.

2.1.3 Begriffe

Zur Erfassung der allgemeinen Eigenschaften von Phänomenen werden Begriffe als Abstraktionen eingeführt. Ein *Begriff* hat folgende drei Aspekte:

- *Name*
- *Intension*: Die gemeinsamen Eigenschaften, die dieser Begriff umfaßt.
- *Extension*: Die Phänomene, die dieser Begriff abdeckt.

Diese rationalistische Sicht des Begriffs steht im Gegensatz zur Bedeutung der Begriffsbildung im WAM-Kontext, die „von ihrem Wesen her ein sozialer Prozeß im Rahmen der Verwendung und Bildung sprachlicher Zeichen" ist (vgl. [KGZ94]). Begriffe fassen sprachlich solche Gegenstände zusammen, die als gleichartig angesehen werden. Dabei ist die Einschätzung von Gleichartigkeit immer situativ und subjektiv geprägt. Der Begriff „Gegenstände" bezeichnet die relevanten Gegenstände des zu modellierenden Ausschnittes aus der realen Welt.

Begriffe werden (nach Madsen und Møller-Pedersen) durch *Abstraktion* gebildet. Dabei wird zwischen verschiedenen Arten der Abstraktion unterschieden:

- *Klassifikation* ist die allgemeine Form der Abstraktion. Ihre Umkehrung ist die *Exemplifikation*.
- *Aggregation* bezeichnet die Definition eines Begriffes mit Hilfe anderer Begriffe, also das Zusammensetzen neuer Begriffe aus vorhandenen Begriffen. Die Umkehrung ist die *Dekomposition*.
- Unter *Generalisierung* versteht man die Einordnung von Begriffen in einer Klassifizierungshierarchie. Oberbegriffe generalisieren Unterbegriffe, umgekehrt sind Unterbegriffe eine *Spezialisierung* ihrer Oberbegriffe.

Man hat nun den Ordner „Rechnungen A-B“ aus obigem Beispiel und zusätzlich alle weiteren Rechnungsordner sowie andere Ordner, die man am Büroarbeitsplatz verwendet. Abstrahiert man nun von den konkreten Eigenschaften der unterschiedlichen Ordner, so kommt man schließlich zu einem allgemeinen Ordnerbegriff, der wie folgt aussehen könnte:

- *Name*: Ordner
- *Intension*: Ein Ordner enthält ein Inhaltsverzeichnis, außerdem kann er eines oder mehrere Dokumente und eines oder mehrere Zwischenblätter enthalten. Ein Ordner kann beschriftet werden, man kann das Inhaltsverzeichnis eines Ordners führen, und in einen Ordner können Dokumente und Zwischenblätter an einer bestimmten Position eingefügt oder aus dem Ordner entnommen werden.
- *Extension*: Der Ordner »Rechnungen A-B«, ...

Abbildung 2.1: Der Begriff „Ordner“

Wie bereits gesagt, ist die Relevanz der Eigenschaften abhängig von der Modellklasse. Deshalb tauchen in der Definition des Begriffs „Ordner“ nicht alle der oben beschriebenen meßbaren Eigenschaften des Phänomens „Ordner »Rechnungen A-B«“ als Intension auf. Die Ordnerfarbe und der Hersteller wurden in der Intension nicht berücksichtigt, da diese Eigenschaften für die Modellierung des Ordnerbegriffs in diesem Zusammenhang (d.h. im Kontext der Modellierung von Informationsprozessen) für nicht relevant gehalten werden.

Die Beschreibung der Intension beinhaltet auch die Möglichkeit eines leeren Ordners ohne Dokumente und Zwischenblätter (aber mit einem Inhaltsverzeichnis). Allerdings macht die Intension keine Angaben über das „Fassungsvermögen“ eines Ordners.

2.1.4 Eine genauere Definition des physischen Modells

Mit diesem Begriffswerk können nun die Elemente eines physischen Modells beschrieben werden:

Ein *physisches Modell* besteht aus *Objekten*, die durch *Attribute* und eine Folge von *Aktionen* charakterisiert sind. Objekte kapseln den Substanzaspekt der Phänomene. Die Aktionsausführung von Objekten spiegelt den Transaktionsaspekt von Begriffen wieder. Objekte können aus Teilobjekten bestehen. Attribute können sein:

- Meßbare Eigenschaften des Objektes (bzw. der gekapselten Substanz)
- Verweise auf Teilobjekte oder selbständige Objekte
- Muster (siehe unten)

Der *Zustand* eines Objektes zu einem bestimmten Zeitpunkt ist definiert durch seine Substanz, seine meßbaren Eigenschaften und die gerade stattfindenden Aktionen. Der *Zustand des Gesamtmodells* ist definiert durch die Zustände aller Objekte im Modell.

Im physischen Modell werden Begriffe durch Muster dargestellt. Ein *Muster* definiert die gemeinsamen Eigenschaften einer Klasse von Objekten, wobei es eine Abstraktion der Substanz, der meßbaren Eigenschaften oder der Aktionsfolgen sein kann. Muster können in Klassifikations-

hierarchien angeordnet sein. In konkreten Programmiersprachen kann das Konzept des Musters durch Klassen oder Typen, aber auch durch Prozeduren (Methoden) oder Funktionen realisiert werden.

Im Rahmen der WAM-Methodik versteht man unter Mustern das, was Gamma et. al. mit dem Begriff „Design Patterns" (deutsch: Entwurfsmuster) bezeichnen, wenn sie Christopher Alexander zitieren: „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way you can use this solution a million times over, without ever doing the same way twice" (zitiert in [GHJV95]). Während der Musterbegriff von Madsen und Møller-Pedersen auf den Eigenschaften von Klassen von Objekten definiert ist, beschreiben Entwurfsmuster Konstruktionen für Probleme auf einer abstrakten Ebene, die jedoch sehr leicht konkretisiert werden können. Die Entwürfe beschreiben die Eigenschaften und das Verhalten bestimmter programmiersprachlicher Objekte bzw. das Zusammenspiel solcher Objekte.

Aktionen werden (wieder nach Madsen und Møller-Pedersen) im physischen Modell von Objekten ausgeführt, und zwar parallel, quasi-parallel oder sequentiell.

2.1.5 Eigenschaften objektorientierter Programmierung

Das Konzept des physischen Modells ist sehr allgemein gehalten. Angewandt auf das Gebiet der Programmierung, werden Programmausführungen als physische Modelle verstanden, was uns wieder zur obigen Definition der objektorientierten Programmierung zurückführt.

Nach Madsen und Møller-Pedersen sollte eine objektorientierte Programmiersprache folgende Konzepte unterstützen:

- Objekte (als Modell von Phänomenen)
- Klassen (als Modell von Begriffen bzw. Mustern)
- Klassifikationshierarchien (hierarchische Beziehungen zwischen Klassen, wobei Unterklassen alle Attribute ihrer Oberklasse(n) erben)
- Virtuelle Attribute (Attribute, die in einer Oberklasse als virtuell deklariert sind, werden erst in den Unterklassen implementiert; dadurch wird den Unterklassen ein bestimmtes Verhalten vorbestimmt. Virtuelle Attribute sollen die einzigen Attribute sein, die dynamisch gebunden werden können)
- Nebenläufigkeit (Unterstützung der nebenläufigen Ausführung von Aktionen in unterschiedlichen Objekten und der Synchronisation von Objekten)
- Persistenz (Unterstützung der Lebensdauer von Programmen über die Programmlaufzeit hinweg)

Als Vorteile der objektorientierten Programmierung nennen Madsen und Møller-Pedersen:

- Physische Modelle spiegeln die reale Welt in einer sehr direkten, natürlichen Art wieder.
- Physische Modelle sind stabiler als die Funktionalität eines Systems.

  Dies entspricht dem von Meyer beschriebenen *Stetigkeitsprinzip*: „Eine Entwurfsmethode erfüllt das Kriterium der Stetigkeit, wenn sie solche Architekturen erzeugt, die nicht wegen jeder kleinen Änderung der Systemanforderungen geändert werden müssen" (vgl. [Mey90]). Meyer stellt ferner fest, daß im Werdegang eines Systems die Funktionen i.A. die flüchtigsten

Teile eines Systems sind; Objekte weisen hingegen eine größere zeitliche Beständigkeit auf. Es ist daher sinnvoll, den Entwurf eines Systems auf den Objekten zu gründen.

- Die Bildung von Unterklassen und die Definition virtueller Klassen erhöhen die Wiederverwendbarkeit.

2.2 Dimensionen objektorientierter Programmiersprachen

Die Definition objektorientierter Programmierung von Madsen und Møller-Pedersen ist sehr allgemein gehalten im Vergleich mit den Definitionen anderen Autoren, die konkret beschreiben, welche Konzepte sie für eine objektorientierte Programmiersprache für notwendig halten. Solche Konzepte werden in dieser Arbeit in Anlehnung an Wegner als *Dimensionen* bezeichnet.

Wegner stellt in [Weg87] folgenden Katalog von Dimensionen für objektbasierte[2] Sprachen auf:

- Objekte
- Klassen
- Vererbung
- Datenabstraktion
- Statische Typisierung
- Nebenläufigkeit
- Persistenz

Einen ähnlichen Katalog findet man bei Booch ([Boo91]). Dieser geht davon aus, daß alle Programmierstile auf jeweils einem konzeptuellen Rahmenwerk aufbauen. Das konzeptuelle Rahmenwerk der objektorientierten Programmierung ist das *Objektmodell*. Dieses besteht aus den vier Kerndimensionen

- Abstraktion
- Kapselung
- Modularität
- Hierarchie

und den drei erweiternden Dimensionen

- Typisierung
- Nebenläufigkeit
- Persistenz

Die Forderung nach Nebenläufigkeit und Persistenz ist auch bei Madsen und Møller-Pedersen zu finden.

Die oben beschriebenen Dimensionen dürfen nicht mit dem Dimensionskatalog verwechselt werden, der in Kapitel 3 beschrieben ist, und an dem SQLWindows im folgenden gemessen werden

[2] Objektbasierte Sprachen sind nach Definition von Wegner eine Oberklasse der objektorientierten Sprachen (vgl. Abschnitt 2.2.1)

soll. Allerdings werden einige der hier beschriebenen Dimensionen in dem Katalog enthalten sein.

Meyer definiert in [Mey90] zwar keine Dimensionen, beschreibt aber „sieben Stufen zur objektbasierten Glückseligkeit", die seiner Meinung nach zu wahrer Objektorientiertheit führen. Je mehr Stufen ein System erreicht, desto berechtigter darf es sich nach Meyer als objektorientiert bezeichnen. Die Stufen sind im einzelnen:

1. *Objektbasierte modulare Struktur*: Systeme werden auf der Grundlage ihrer Datenstrukturen modularisiert.
   Entspricht der Definition objektbasierter Sprachen bei Wegner.
2. *Datenabstraktion*: Objekte müssen als Implementierungen abstrakter Datentypen beschrieben werden.
   Diese Forderung wird auch von Stroustrup in [Str87] aufgestellt. Er bezeichnet objektorientierte Programmierung als eine Erweiterung der Programmierung mit benutzerdefinierten Typen (Datenabstraktion) um das Konzept der Vererbung.
3. *Automatische Speicherplatzverwaltung*: Unbenutzte Objekte sollten ohne Programmiereingriff vom unterliegenden Sprachsystem freigegeben werden.
4. *Klassen*: Jeder nicht-einfache Typ ist ein Modul, und jedes Modul höherer Ebene ist ein Typ.
5. *Vererbung*: Eine Klasse kann als Einschränkung oder Erweiterung einer anderen definiert werden.
   Eine allgemeinere Definition als bei Wegner.
6. *Polymorphismus und dynamisches Binden*: Programm-Elemente dürfen sich auf Objekte aus mehr als einer Klasse beziehen, und Operationen dürfen unterschiedliche Realisierungen in unterschiedlichen Klassen haben.
   Entspricht der Definition von Delegation bei Wegner (vgl. Abschnitt 2.2.4).
7. *Mehrfaches und wiederholtes Erben*: Man kann Klassen deklarieren, die Erben von mehr als einer Klasse sind und mehr als einmal von einer Klasse erben.

   Wegner faßt diese beiden Begriffe unter dem Begriff der Mehrfachvererbung zusammen. Dabei weist er ausdrücklich darauf hin, daß es kein allgemeingültiges Konzept über die Art und Weise der Methodenkombination gibt, die im Falle von wiederholtem Erben zum Tragen kommt.

Mit der automatischen Speicherplatzverwaltung (Stufe 3) ordnet Meyer eine sehr spezielle und – wie er selbst zugibt – system- und nicht sprachspezifische Eigenschaft vor den allgemeineren Konzepten wie Klassen und Vererbung in seiner Hierarchie an. Dieser Punkt ist deshalb diskussionswürdig:

Es scheint fast, als wolle Meyer mit dieser Konstruktion Sprachen wie C++, die alle übrigen Stufen „erklimmen" können, davon abhalten, den Olymp der Objektorientierung zu erreichen, so daß „sein" Eiffel allein an der Spitze der objektorientierten Sprachen (und solchen, die es sein wollen) steht. Betrachtet man jedoch die sehr junge Programmiersprache Java, die den Anspruch hat, eine Essenz älterer Programmiersprachen zu bilden, so findet man auch dort eine automatische Speicherplatzverwaltung (vgl. [Fla96]). Und vergleicht man Eiffel und Java, so stellt man fest, daß in beiden Sprachen alle nicht-primitiven Datentypen als Referenztypen behandelt werden – eine Variable enthält immer nur eine Referenz auf ein Objekt, aber nie das Objekt selbst, und auch bei Methodenaufrufen werden nur Objektreferenzen übergeben. In C++ muß in diesem Fall explizit

eine Zeigervariable deklariert werden – mit all den Nachteilen, die Zeiger mit sich bringen. Mit anderen Worten: In Eiffel und Java unterliegen alle einfachen Datentypen der *Wertsemantik* und alle nicht-primitiven Datentypen der *Referenzsemantik*, während in Sprachen wie C++ Objekte sowohl Wert- als auch Referenzsemantik aufweisen. Das ist zum einen unnötig, wie Eiffel oder Java beweisen, und fördert zum anderen nicht unbedingt die Lesbarkeit von Programmen, da man immer zwischen den beiden semantischen Konzepten und ihren jeweiligen Möglichkeiten und Konsequenzen unterscheiden muß. Meyer möchte anscheinend mit seiner „dritten Stufe zur objektbasierten Glückseligkeit" die Zeiger aus objektorientierten Programmiersprachen verbannen, um die mit diesen Sprachen entwickelten Programme besser lesbar und somit besser wartbar, robuster und zuverlässiger zu machen.

Im folgenden sollen die Dimensionen Wegners näher untersucht werden, da diese eine Obermenge der bei Booch aufgeführten Dimensionen bilden. Anschließend werden alle Stufen Meyers beschrieben, die nicht durch die Dimensionen Wegners abgedeckt sind.

2.2.1 Objekte, Klassen und Vererbung

Wegner definiert den Begriff der objektbasierten Sprache wie folgt:

„Eine Sprache ist objektbasiert, wenn sie Objekte als Sprachelemente unterstützt."

Wegners Objektbegriff umfaßt Operationen und einen Objektzustand und entspricht somit weitgehend der Definition von Madsen und Møller-Pedersen. Deren Muster bezeichnen in etwa das, was Wegner unter Klassen versteht: Eine Schablone („cookie cutter"), nach der Objekte erzeugt werden können. Klassen bieten eine oder mehrere „Schnittstellen" an. Eine Schnittstelle ist definiert durch eine Anzahl von Operationen, die einem Klienten der Klasse (genauer: einer Schnittstelle der Klasse) zur Verfügung stehen.

Der Begriff der Vererbung ist bei Wegner wesentlich enger gefaßt als bei anderen Autoren. Er betrachtet die Vererbung lediglich als Mechanismus zur gemeinsamen Nutzung von Ressourcen. Später verallgemeinert er mit der Definition der Delegation das Konzept der Vererbung. Die Delegation wird in Abschnitt 2.2.4 eingeführt.

Als objektorientiert bezeichnet Wegner objektbasierte Sprachen, bei denen jedes Objekt einer Klasse zugeordnet werden kann und eine Klassenhierarchie inkrementell durch einen Vererbungsmechanismus definiert werden kann. Mit der Definition von klassenbasierten Sprachen als Sprachen, in denen jedes Objekt eine Klasse hat, baut Wegner folgende Sprachenhierarchie auf:

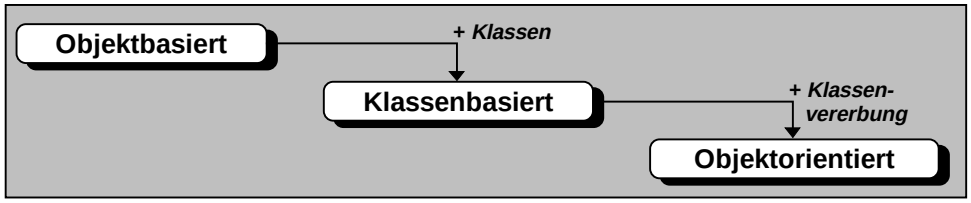


Abbildung 2.2: Klassifikation von Programmiersprachen nach Wegner

2.2.2 Datenabstraktion und statische Typisierung

Nach Wegner ist ein Objekt eine Datenabstraktion, wenn dessen Zustand nur über die Operationen des Objektes zugreifbar ist, wobei der Zustand eines Objektes im allgemeinen durch Exemplarvariablen repräsentiert wird.

Eine Sprache bezeichnet Wegner dann als statisch getypt, wenn die Typkompatibilität für alle werttragenden Ausdrücke aus der statischen Programmbeschreibung zur Übersetzungszeit bestimmt werden kann. Objektorientierte Programmiersprachen mit statischer Typisierung, in der alle Objekte Datenabstraktionen sind, bezeichnet Wegner als streng getypte objektorientierte Sprachen. Diese Forderungen stellen eine Einschränkung in der Freiheit des Softwareentwicklers dar, ermöglichen aber nach Wegners Meinung erst die Realisierung größerer Softwareprojekte. Eine Hauptanwendung für nicht statisch getypte Sprachen sieht Wegner im Prototyping, zumal eine Prototyp-Beschreibung später in eine statisch getypte Sprache übersetzt werden kann und so direkt als Grundlage für die Realisierung dienen kann.

Wegners Begriff der Datenabstraktion findet man bei Meyer in stark formalisierter Form im Begriff des abstrakten Datentypen wieder, der im nächsten Abschnitt näher betrachtet werden soll, bevor die übrigen Dimensionen Wegners in den darauffolgenden Abschnitten beschrieben werden.

2.2.3 Abstrakte Datentypen

Meyer definiert in [Mey90] Klassen als Implementierungen abstrakter Datentypen. Dabei versteht er unter einem *abstrakten Datentyp* eine Klasse von Datenstrukturen, die durch eine äußere Sicht beschrieben werden, nämlich durch die verfügbaren Dienste und die Eigenschaften dieser Dienste. Verglichen mit Wegners Dimensionen, ist dies kein gänzlich neues Konzept; der Begriff des abstrakten Datentyps macht aber explizit, was in Wegners Definition klassenbasierter Sprachen implizit im Objektbegriff enthalten ist.

Die Beschreibung eines abstrakten Datentyps besteht aus folgenden Abschnitten:

- *Typen*: Zählt die Typen der Spezifikation auf. Dabei kann es sich auch um generische Typen handeln, die mit einem Formalparameter versehen werden, der einen beliebigen Typ darstellt. Dabei wird zunächst der zu spezifizierende abstrakte Datentyp aufgeführt, gefolgt von den Typen, die zu seiner Spezifikation notwendig sind.
- *Funktionen*: Führt die Dienste (in Form mathematischer Funktionen) auf, die von Exemplaren dieses Typs zur Verfügung gestellt werden. Man unterscheidet zwischen erzeugenden Konstruktor-Funktionen, wertliefernden Zugriffsfunktionen und zustandsverändernden Transformationsfunktionen. Die Funktionen können auch partiell sein, d.h. sie sind nicht notwendigerweise für jedes Objekt des abstrakten Datentyps definiert.
- *Vorbedingungen*: Definieren die Anforderungen an die Benutzbarkeit jeder partiellen Funktion des abstrakten Datentyps.
- *Axiome*: Beschreiben semantische Eigenschaften, die charakteristisch für diesen abstrakten Datentyp sind.

Ein klassisches Beispiel für einen abstrakten Datentyp ist der Keller (Stack). Bei Meyer wird er wie folgt definiert (vgl. [Mey90], [KGZ94]):

TYPEN

STACK[X], BOOLEAN

FUNKTIONEN

empty:	STACK[X] $\rightarrow$ BOOLEAN	
new:	$\rightarrow$ STACK[X]	
push:	X × STACK[X] $\rightarrow$ STACK[X]	
pop:	STACK[X] $\rightarrow_{/}$ STACK[X]	*partiell*
top:	STACK[X] $\rightarrow_{/}$ X	*partiell*

VORBEDINGUNGEN

pre *pop* (s: STACK[X]) = (**not** *empty* (s))
pre *top* (s: STACK[X]) = (**not** *empty* (s))

AXIOME

Für alle x: X, s: STACK[X]:

empty (*new* ())
not *empty* (*push* (x, s))
top (*push* (x, s)) = x
pop (*push* (x, s)) = s

Abbildung 2.3: Der abstrakte Datentyp STACK

Der zu definierende abstrakte Datentyp STACK[X] ist ein generischer Typ. Der Formalparameter X steht für einen beliebigen Typ. Konkrete Kellertypen erhält man, indem man X durch einen konkreten Datentyp wie z.B. INTEGER ersetzt.

Die Funktionen werden in der Form

$$name: A_1 \times A_2 \times ... \times A_m \rightarrow B_1 \times B_2 \times ... \times B_n$$

notiert. Die Funktion bekommt *m* Argumente A_1, A_2, ... , A_m und liefert *n* Ergebnisse B_1, B_2, ... , B_n. Mindestens eines der A_i oder B_j muß vom zu spezifizierenden abstrakten Datentyp sein. Es wird unterschieden zwischen totalen Funktionen ($\rightarrow$) und partiellen Funktionen ($\rightarrow_{/}$).

Funktionen eines abstrakten Datentyps T können kategorisiert werden in:

- *Konstruktor-Funktionen*: T tritt nur in den Ergebnissen der Funktion auf. Derartige Funktionen erzeugen neue Objekte von T. Diese T-Objekte können nun als Argumente für andere Funktionen dienen. So erzeugt die Funktion *new* in obigem Beispiel ein neues STACK-Objekt.
- *Zugriffsfunktionen*: T tritt nur in den Argumenten der Funktion auf. Zugriffsfunktionen erzeugen Attribute vorhandener T-Objekte, ausgedrückt als Werte anderer Typen. Nach Louden ([Lou94]) kann man diese Funktionen weiter aufgliedern in *Prädikate*, die boolesche Werte zurückgeben (z.B. *empty*), und *Selektoren*, die nicht-boolesche Werte als Ergebnis liefern (z.B. *top*).
- *Transformationsfunktionen*: T tritt sowohl in den Argumenten als auch in den Ergebnissen der Funktion auf. Eine solche Funktion erzeugt aus vorhandenen T-Objekten (und evtl. anderen Argumenten) neue T-Objekte. Beispiele hierfür sind die Funktionen *push* und *pop*.

Für jede partielle Funktion müssen die Bedingungen ihrer Anwendbarkeit in Form von Vorbedingungen festgelegt werden.

Die Axiome machen schließlich Aussagen, die für alle Objekte des zu spezifizierenden abstrakten Datentyps gelten, d.h. sie legen die Semantik des Typs fest.

2.2.4 Delegation

Unter Delegation versteht Wegner einen Mechanismus, der es Objekten erlaubt, die Verantwortlichkeit für das Ausführen einer Operation oder das Aufspüren eines Wertes an einen oder mehrere festgelegte „Vorfahren" zu delegieren. Wegner verwendet den allgemeinen Begriff des „Vorfahren", um die Delegation unabhängig vom Klassenbegriff definieren zu können.

Selbstreferenz und dynamisches Binden erweitern die Objekte um ihre Vorfahren bzw. deren Ressourcen, beim dynamischen Binden sogar flexibel in Abhängigkeit vom Zeitpunkt der Programmausführung. Man kann Selbstreferenz und dynamisches Binden als eine Art von Delegation „nach unten" verstehen:

- Durch die Verwendung von Selbstreferenz in einer Oberklasse kann in dieser Klasse eine Methode aufgerufen werden, die in einer abgeleiteten Klasse überschrieben wurde, so daß bei einem Objekt, das vom dynamischen Typ der abgeleiteten Klasse ist, diese überschriebene (und in der Oberklasse bzgl. der Implementierung unbekannte) Methode aufgerufen wird. Die Oberklasse delegiert also „nach unten" an die abgeleitete Klasse.
- Beim dynamischen Binden kann der statische Typ eines Objektes vom dynamischen Typ verschieden sein. In diesem Fall ist der dynamische Typ immer vom statischen Typ abgeleitet. Wird nun eine Methode der Klasse des statischen Typs aufgerufen, so wird durch das dynamische Binden tatsächlich eine (evtl. überschriebene) Methode der den dynamischen Typ repräsentierenden abgeleiteten Klasse aufgerufen: Die Arbeit wird an die abgeleitete Klasse delegiert.

Demgegenüber realisiert der Vererbungsmechanismus eine Delegation „nach oben", da Methoden, die in einer Oberklasse deklariert und implementiert sind, in (bzw. von) abgeleiteten Klassen aufgerufen werden können. Nach Wegners Meinung verallgemeinert das Konzept der Delegation das Vererbungskonzept, indem es dessen Abhängigkeit vom Klassenbegriff auflöst. Dabei stehen für die Delegation folgende Entwurfsalternativen zur Verfügung:

1. Klassenlose Delegation gegenüber Vererbung: Gemeinsame Nutzung in einer Instanzenhierarchie gegenüber gemeinsamer Nutzung in einer Klassenhierarchie.
2. Strikte gegenüber nicht-strikter Vererbung: Bei strikter Vererbung müssen die Erben in ihrem Verhalten verträglich mit den Vorfahren sein („is a"-Beziehung), wohingegen nicht-strikte Vererbung das Überschreiben von Operationen erlaubt („like"-Beziehung).
3. Einfache gegenüber mehrfacher Vererbung: Mehrfache Vererbung erlaubt es einem Objekt, mehrere Vorfahren zu haben.
4. Abstrakte Schnittstelle gegenüber gemeinsamer Nutzung von Code.

Delegation beschreibt demnach die gemeinsame Nutzung von Ressourcen (Code) auf einer sehr abstrakten Ebene.

Sprachen, die Delegation unterstützen, bezeichnet Wegner als delegationsbasierte Sprachen.

Der Begriff der virtuellen Ressourcen aus den klassenbasierten Sprachen kann auf delegationsbasierte Sprachen übertragen werden: Virtuelle Ressourcen (bei Madsen und Møller-Pedersen „virtuelle Attribute" genannt) sind im „Vorfahren" spezifiziert, werden aber erst im „Nachkommen"

implementiert. Dies findet in klassenbasierten Sprachen im Konzept der abstrakten Klasse Verwendung. Eine abstrakte Klasse ist eine Klasse mit virtuellen Ressourcen, die nur in Form einer konkreten Unterklasse instantiiert werden kann, in der die virtuellen Ressourcen implementiert sind.

2.2.5 Prototypen

Unter einem Prototyp versteht Wegner ein Objekt, das sowohl Exemplar als auch Schablone ist. Objekte dürfen die Verantwortlichkeit für das Ausführen einer Operation oder das Aufspüren eines Wertes an einen Prototyp delegieren. Der Prototyp stellt anfragenden Objekten Voreinstellungen für seine Operationen und Werte zur Verfügung. Anstatt die allgemeinen Eigenschaften eines Objektes in einer Klasse zu abstrahieren, wird dieses Objekt zum Prototypen deklariert. „Ähnliche" Objekte werden nur durch ihre Unterschiede gegenüber dem Prototyp repräsentiert. Dieses Vorgehen kommt nach Meinung Wegners dem Prozeß des Wissenserwerbs für menschliche kognitive Prozesse näher als der Klassenmechanismus, da Menschen erst dann eine Abstraktion bzw. Klassenbildung vornehmen, wenn sie bereits mehrere Exemplare einer solchen Klasse gesehen haben.

Prototypische Sprachen sind delegationsbasierte Sprachen, in denen die Delegation durch Prototypen realisiert wird: Vom Prototypen können beliebig viele Kopien erzeugt werden, welche die „Arbeit", also die Ausführung von Methoden, an den Prototyp delegieren.

Die Unterscheidung zwischen klassenbasierten und prototypischen Sprachen reflektiert die philosophische Diskussion um die Bedeutung und Repräsentation der Abstraktion. Klassenbasierte Sprachen kann man in ihrer Benutzung von Klassen zum Zweck der Abstraktion als platonisch bezeichnen. Platon glaubte nämlich an eine eigene Wirklichkeit hinter der „Sinnenwelt". Diese Wirklichkeit nannte er Welt der Ideen. Hier findet man die ewigen und unveränderlichen „Musterbilder" („Klassen" in unseren Begriffen), die Urbilder hinter den verschiedenen Phänomenen, die uns in der Natur begegnen (vgl. [Gaa93]).

2.2.6 Nebenläufigkeit

Objektbasierte Nebenläufigkeit definiert Wegner auf der Grundlage folgender Definition des Prozeßbegriffes:

> „Prozesse haben eine Schnittstelle, bestehend aus ausführbaren Operationen oder Eintrittspunkten, und einen oder mehrere Kontrollstränge, die aktiv oder unterbrochen sein können."

Unterstützt eine objektbasierte Sprache die Modellierung von Prozessen, so bezeichnet Wegner sie als prozeßbasierte Sprache. Dabei unterscheidet er Prozesse (und entsprechende Sprachen) nach der Anzahl (aktiver) Kontrollstränge in sequentielle, quasi-nebenläufige und nebenläufige Prozesse (Sprachen).

2.2.7 Persistenz

Persistenz ist die Eigenschaft der Daten, welche deren Lebensdauer bestimmt. Das Erweitern einer objektorientierten Sprache um Persistenz prädestiniert eine solche Sprache als Grundlage

für die objektorientierte Datenbankentwicklung. Wegner ist der Meinung, daß die Organisation von Daten in Form von Objekten dem relationalen Datenmodell überlegen ist. Eine Datenbank sieht er als langlebigen Prozeß, der asynchrone Benutzeranfragen verarbeitet. Voraussetzung für Persistenz in objektorientierten Programmiersprachen ist ein Konzept der Objekt-Identität, das nicht auf den Attributen der Objekte basieren darf (wie es beim Konzept des Schlüsselattributes im relationalen Datenmodell der Fall ist) und eine programmübergreifende und über die Programmlaufzeit hinausgehende Gültigkeit haben muß.

2.2.8 Orthogonale Redefinition von Wegners Dimensionen

Mit den soeben eingeführten Begriffen und Konzepten Wegners redefiniert dieser seine Dimensionen derart, daß sie orthogonal sind. Das bedeutet, daß sich die Dimensionen nicht mehr gegenseitig bedingen, wie dies z.B. beim Vererbungsbegriff der Fall ist, der zur Definition den Begriff der Klasse benötigt. Wegners neuer Dimensionskatalog umfaßt folgende Dimensionen:

- Objekte
- Typen
- Delegation
- Abstraktion
- Nebenläufigkeit
- Persistenz

Auch die hier aufgeführten Dimensionen stellen noch nicht den zu Beginn dieses Kapitels erwähnten Dimensionskatalog für objektorientierte Programmiersprachen dar. Dieser wird erst in Kapitel 3 beschrieben.

2.2.9 Zusammenfassung der Konzepte

Wie im Verlauf des vorigen Abschnitts durch Verweise auf die Definition von Madsen und Møller-Pedersen deutlich geworden ist, findet man alle von den beiden geforderten Konzepte in z.T. begrifflich leicht abgewandelter Form in den Dimensionen von Wegner, Booch und Meyer wieder. Wie aber bereits erwähnt, ist der Anwendungsrahmen bei Madsen und Møller-Pedersen wesentlich weiter gefaßt, indem sich deren Definition von objektorientierter Programmierung nicht nur auf die Programmierung selbst anwenden läßt, sondern gleichermaßen die Grundlage für Analyse und Entwurf bildet.

2.3 Objektorientierte Programmierung im WAM-Verständnis

Um die objektorientierten Konzepte und Entwurfsprinzipien der WAM-Metapher verwirklichen zu können, bedarf es nach Ansicht von Kilberth et. al. einer adäquaten Programmiersprache, einer Programmierumgebung und der Möglichkeit, ein Datenbanksystem an ein objektorientiertes System anzuschließen (vgl. [KGZ94]). Um den Übergang von dem aus der Analyse des Anwendungsbereiches entwickelten fachlichen Entwurf in einen technischen Entwurf möglichst ohne

konzeptuellen Bruch durchführen zu können, benötigt man eine Programmiersprache, welche die Verwendung objektorientierter Konzepte erlaubt bzw. unterstützt. Kilberth et. al. greifen die Klassifikation von Programmiersprachen nach Wegner (vgl. Abbildung 2.2) auf und fordern als Programmiersprache für den technischen Entwurf nach WAM eine in Wegners Sinne objektorientierte Programmiersprache. Deshalb schien es sinnvoll, die in dieser Arbeit zu untersuchende Programmiersprache SAL an einem Dimensionskatalog zu messen, der mindestens die Dimensionen Wegners umfaßt.

2.3.1 Referenzen

Ein Konzept, das in obigen Dimensionskatalogen fehlt, das aber von großer Bedeutung für die objektorientierte Programmierung ist, ist das Referenzkonzept. In Eiffel sind alle Klassenattribute, die von einem Klassentyp definiert sind, Referenzen auf Objekte der entsprechenden Klasse (Meyer benutzt den Begriff „Verweis" statt „Referenz", vgl. [Mey90]). In Java haben die einfachen Datentypen Wertsemantik, während Objekte und Arrays der Referenzsemantik unterliegen, d.h. die Adresse eines Objektes oder Arrays wird in einer Variablen gespeichert, an eine Methode übergeben etc., nicht aber das Objekt oder Array selbst. Solche Datentypen werden in Java *Referenz-Datentypen* (*reference data types*) genannt (vgl. [Fla96]).

Die Möglichkeit, Referenzen auf Objekte definieren zu können, ist unter anderem für die Implementierung von Benutztbeziehungen zwischen Objekten notwendig: Der Zustand eines Objektes, d.h. seine (privaten) Attribute, muß neben Variablen einfachen Typs auch Objekte beinhalten. Da ein solches Objektattribut eventuell auch für andere Objekte als Attribut von Bedeutung ist, muß anstatt des Objektes selber eine Referenz auf das Objekt als Attribut gespeichert werden, oder das Objekt wird global deklariert. Letztere Lösung verstieße aber gegen das Prinzip der Datenabstraktion. Objektreferenzen als Attribute sind somit ein notwendiges Sprachmittel.

Referenzen auf Objekte werden auch für bestimmte Konzepte der WAM-Architektur benötigt:

- Um die Austauschbarkeit von Interaktionskomponenten unter Beibehaltung der Funktionskomponente realisieren zu können, hält i.d.R. die Interaktionskomponente eine Referenz auf ihre Funktionskomponente.
- Die lose Kopplung zwischen Funktions- und Interaktionskomponente wird durch den Beobachtermechanismus ermöglicht. Dazu registriert sich das Beobachter-Objekt beim zu beobachtenden Objekt. Letzteres benachrichtigt bei Zustandsänderungen alle beobachtenden Objekte. Dazu hält es eine Liste von Referenzen auf die Beobachter-Objekte.
- Beim Einsatz von Sub-Komponenten in Werkzeugen halten die Kontext-Komponenten Referenzen auf ihre Sub-Komponenten.
- Materialien werden per Referenz an das bearbeitende Werkzeug übergeben.

Die angesprochenen Konzepte werden in Kapitel 5 und in [RZ95] ausführlich erläutert. Das Referenzkonzept wird in den Abschnitten 4.3.5 und 6.2.1 weiter vertieft.

Die Beispiele zeigen, daß Referenzen sowohl generell in der Objektorientierung als auch speziell im WAM-Kontext einen großen Stellenwert einnehmen.

2.3.2 Werte

Neben dem Objektkonzept kommt im Rahmen des WAM-Ansatzes auch einem Wertkonzept – im speziellen dem Konzept der fachlichen Werte oder Fachwerte – eine besondere Bedeutung zu. Werte und Objekte sind sehr unterschiedliche Konzepte. Die Unterschiede sollen im folgenden herausgearbeitet werden. Dazu werden die Eigenschaften, die einen Wert auszeichnen, gegen die Eigenschaften eines Objektes gesetzt (vgl. [Zül97]):

Wert	**Objekt**
zeit- und ortlos	existiert in Raum und Zeit
abstrakt; keine Identität, nur Gleichheit	konkret; Identität und Gleichheit
unveränderlich	veränderbar, bei Wahrung der Identität
kann nicht gemeinsam benutzt werden	kann über Referenzen gemeinsam benutzt werden

Tabelle 2.1: Eigenschaften von Wert und Objekt

- Auf Werte sind Begriffe wie Zeit, Dauer und Ort nicht anwendbar – sie existieren „ewig und universell", unabhängig von einem zeitlichen und räumlichen Kontext (Beispiel: In der Gleichung `40 + 2 = 42` wird kein neuer Wert erzeugt, und die drei Werte sind auch nicht an diese Gleichung gebunden); Objekte hingegen werden zu konkreten Zeitpunkten erzeugt und zerstört und befinden sich an einem Ort, der sie auch von anderen Objekten unterscheidbar macht.
- Ein Wert abstrahiert von allen konkreten Kontexten. Wenngleich es auch mehrere Exemplare seiner Repräsentation geben kann, so ist der Wert an sich doch identitätslos (Beispiel: Der Wert „50 DM" hat als Repräsentation z.B. Banknoten, von denen es viele Exemplare gibt); Objekte hingegen haben eine Identität – sie sind konkrete Exemplare eines allgemeinen Konzeptes (einer Klasse).
- Werte können zwar berechnet und aufeinander bezogen werden, verändern sich dabei aber nicht (Beispiel: In der Gleichung `40 + 2 = 42` wird die Zahl 40 nicht durch die Addition zum Wert 42 verändert – es werden lediglich Werte aufeinander bezogen); demgegenüber hat ein Objekt einen Zustand, der sich ändern kann – unter Beibehaltung der Identität des Objektes.
- Werte sind – wie bereits gesagt – identitätslos, ortsungebunden und unveränderlich. Damit eignen sie sich nicht für Benutztbeziehungen, denn sie können nicht ausgetauscht und bearbeitet werden; zu einem Objekt kann es jedoch mehrere Referenzen geben, die an verschiedenen Orten bekannt sind und die gemeinschaftliche Nutzung des Objektes erlauben.

Ziel des WAM-Ansatzes ist es, die Konzepte und Gegenstände des Anwendungsbereiches zu modellieren. Als Modellierungsmittel stellt die objektorientierte Methodik die Konzepte der Klasse und des Objektes zur Verfügung. Bei der Analyse des Anwendungsbereiches sind meistens auch Werte zu finden, die es schließlich auch zu modellieren gilt. Im WAM-Kontext werden solche Werte als Fachwerte bezeichnet. In [Zül97] sind diese wie folgt definiert:

> „Ein *Fachwert* entspricht einem benutzerdefinierten Datentyp. Er ist ein anwendungsfachlich motivierter Typ, der die charakteristischen Eigenschaften eines Werts besitzt, mit definierter Wertemenge und Operationen.

> Fachwerte werden in objektorientierten Sprachen als Klassen definiert. Wichtig ist, daß die Exemplare eines Fachwerts immer Wertsemantik besitzen, d.h. daß ein einmal gesetzter Wert nicht mehr verändert werden kann."

Neben den Objekten müssen also auch die Werte bei der Modellierung des Anwendungsbereiches berücksichtigt werden. Das Wertkonzept stellt demnach ein für den WAM-Ansatz notwendiges Konzept dar, das von der für die softwaretechnische Realisierung verwendeten Programmiersprache angemessen unterstützt werden muß.

2.4 Zusammenfassung

Dieses Kapitel führt mehrere Definitionen des Begriffs der „Objektorientierten Programmiersprache" auf, wie man sie in der Literatur zur Objektorientierung findet. Neben unterschiedlichen Auffassungen über Anzahl und Art von Merkmalen, die eine objektorientierte Programmiersprache ausmachen, ist es vor allem die Nähe zu programmiersprachlichen Konzepten und Begriffen, in der sich die einzelnen Definitionen unterscheiden.

Die sprachunabhängigste und umfassendste der hier untersuchten Definitionen stammt von Madsen und Møller-Pedersen. Diese verstehen die Ausführung eines Programmes als ein physisches Modell, welches das Verhalten eines realen oder imaginären Ausschnitts der Welt simuliert. Dieser Begriff des physischen Modells bildet den Kern der Definition.

Wegner führt den Begriff der Dimensionen objektorientierter Programmiersprachen ein, der dem Dimensionskatalog (vgl. Kapitel 3) seinen Namen gab. Wegner konstruiert eine hierarchische Klassifizierung der Programmiersprachen in objektbasierte, klassenbasierte und objektorientierte Sprachen. Kriterium für die Zuordnung einer Programmiersprache zu einer dieser Klassen sind die Dimensionen, die diese Programmiersprache erfüllt. Neben dieser Klassifizierung ist noch Wegners Definition von prototypischen Programmiersprachen hervorzuheben, in denen ein Objekt sowohl Exemplar als auch Schablone sein kann und somit – aus Sicht der objektorientierten Programmierung – die Rolle von Objekt und Klasse übernimmt.

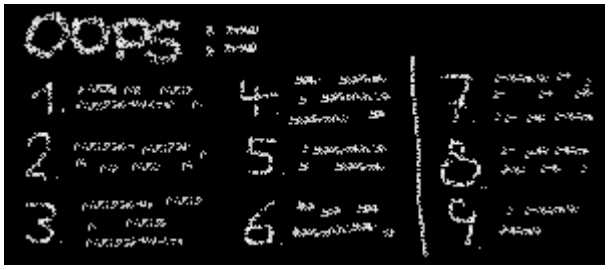
Von Booch stammt die Unterscheidung in Kerndimensionen und erweiternde Dimensionen. Erstere sind notwendig, damit eine Programmiersprache als objektorientiert bezeichnet werden kann. Letztere bezeichnen Konzepte, die bei einigen objektorientierten Programmiersprachen zu finden sind, die aber nicht essentiell für die Objektorientierung sind. Die bei Booch genannten Dimensionen objektorientierter Programmiersprachen bilden eine Teilmenge von Wegners Dimensionen.

Meyer definiert Klassen als Implementierungen abstrakter Datentypen. Den Begriff des abstrakten Datentypen definiert er sehr formal, so daß sein Klassenbegriff sehr exakt gefaßt ist. Meyers Auffassung von Objektorientierung deckt sich weitgehend mit Wegners Definitionen.

Schließlich wurde betrachtet, was im WAM-Kontext unter objektorientierter Pogrammierung verstanden wird. Hier zeigt sich, daß ein wichtiges Konzept in keiner der obigen Definitionen auftaucht: Das Referenzkonzept. Die Möglichkeit, Referenzen auf Objekte definieren zu können, ist Voraussetzung für die Umsetzung mehrerer wichtiger objektorientierter Konzepte wie z.B. der Implementierung von Benutztbeziehungen. Auch bestimmte Konzepte der WAM-Architektur benötigen ein Referenzkonzept. Ein weiteres Konzept, das im Rahmen des WAM-Ansatzes eine große Rolle spielt, ist das des Wertes, genauer: des Fachwertes, da bei der Analyse des Anwen-

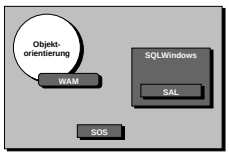
dungsbereiches neben Objekten meistens auch Werte identifiziert werden, die im Modell berücksichtigt werden müssen. Aus diesem Grund müssen das Referenzkonzept und das Wert- und Fachwert-Konzept mit in den Katalog der Dimensionen aufgenommen werden. Die Definition eines solchen Dimensionskataloges soll Inhalt des nächsten Kapitels sein.

3 Ein Dimensionskatalog für objektorientierte Programmiersprachen



Auf der Grundlage der in Kapitel 2 beschriebenen Sichten auf die Objektorientierung wurde ein Dimensionskatalog für objektorientierte Programmiersprachen entwickelt, der im folgenden dargestellt werden soll. Dabei wird zwischen Kerndimensionen und erweiternden Dimensionen unterschieden. Erstere sind notwendig, damit eine Programmiersprache als objektorientiert bezeichnet werden kann. Letztere bezeichnen Konzepte, die bei einigen objektorientierten Programmiersprachen zu finden sind, die aber nicht essentiell für die Objektorientierung sind.

In Abschnitt 3.1 werden die Kerndimensionen definiert, gefolgt von den erweiternden Dimensionen in Abschnitt 3.2. Abschnitt 3.3 stellt abschließend die Dimensionen zusammenfassend dar.



Dieses Kapitel definiert den Begriff der „Objektorientierten Programmiersprache“ für den Kontext dieser Diplomarbeit.

3.1 Kerndimensionen

Als Kerndimensionen werden folgende Dimensionen bezeichnet:

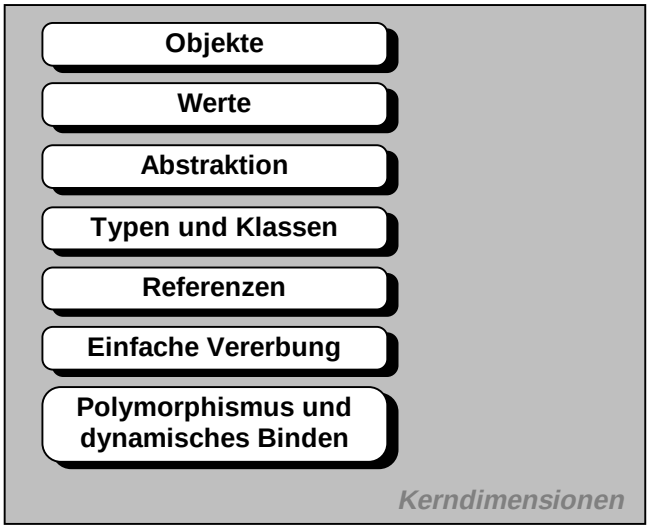


Abbildung 3.1: Kerndimensionen

Diese Dimensionen müssen von einer Programmiersprache mindestens erfüllt sein, damit sie sich (im hier vertretenen Verständnis von Objektorientierung) als objektorientiert bezeichnen darf.

3.1.1 Objekte

Ein *Objekt* besitzt eine Menge von Funktionen oder Diensten, die es einem Klienten (d.h. einem Benutzer des Objektes) zur Verfügung stellt, und einen Zustand, der im Allgemeinen durch lokale Variablen repräsentiert wird. Ein Objekt ist über seinen Namen eindeutig identifizierbar.

3.1.2 Werte

Ein *Wert* modelliert eine abstrakte Größe. Werte sind durch folgende Eigenschaften charakterisiert:

- Auf Werte sind Begriffe wie Zeit, Dauer und Ort nicht anwendbar.
- Ein Wert abstrahiert von allen konkreten Kontexten. Wenngleich es auch mehrere Exemplare seiner Repräsentation geben kann, so ist der Wert an sich doch identitätslos.
- Werte können zwar berechnet und aufeinander bezogen werden, verändern sich dabei aber nicht.
- Werte lassen sich nicht gemeinsam benutzen, denn sie können nicht ausgetauscht und bearbeitet werden.

Eine Spezialisierung des Wertkonzeptes ist der Fachwert:

Ein *Fachwert* entspricht einem benutzerdefinierten Datentyp. Er ist ein anwendungsfachlich motivierter Typ, der die charakteristischen Eigenschaften eines Werts besitzt, mit definierter Wertemenge und Operationen.

Fachwerte werden in objektorientierten Sprachen als Klassen definiert. Wichtig ist, daß die Exemplare eines Fachwerts immer Wertsemantik besitzen, d.h. daß ein einmal gesetzter Wert nicht mehr verändert werden kann.

3.1.3 Abstraktion

Der Zustand eines Objektes kann nach außen nur über entsprechende Dienste bekannt gemacht werden. Die interne Struktur eines Objektes bleibt dem Benutzer dieses Objektes verborgen.

3.1.4 Typen und Klassen

Typen sind Verhaltensspezifikationen in Form von *abstrakten Datentypen* (vgl. Abschnitt 2.2.3). Objekte sind Instanzen von Implementierungen solcher abstrakter Datentypen, die als *Klassen* bezeichnet werden.

3.1.5 Referenzen

Referenzen (auch *Verweise* genannt) auf Objekte müssen

- als Attribute (lokale Variablen) speicherbar sein.
- als Parameter einer Funktion oder Methode[3] angegeben werden können.

Initial sind solche Objektreferenzen leer, d.h. sie verweisen auf nichts. Dies läßt sich auf zwei Arten ändern:

- Man erzeugt ein neues Objekt und verbindet es mit der Referenz.
- Man verbindet die Referenz mit einem bereits existierenden Objekt.

3.1.6 Einfache Vererbung

Eine Klasse kann mit den Begriffen einer anderen (Basis-)Klasse definiert werden. Die abgeleitete Klasse „erbt“ dann die Eigenschaften und Dienste der Basisklasse (*einfache Vererbung*).

3.1.7 Polymorphismus und dynamisches Binden

Eine Funktion kann sich zur Laufzeit eines Programmes auf Objekte verschiedener Klassen beziehen (*Polymorphismus*). Das Laufzeitsystem wählt automatisch die zur entsprechenden Klasse passende Version der Funktion aus (*dynamisches Binden*).

[3] Der Begriff „Methode“ bezeichnet hier – in Übereinstimmung mit einem Großteil der Literatur zur Objektorientierung – eine Operation einer Klasse.

3.2 Erweiternde Dimensionen

Als erweiternde Dimensionen werden folgende Dimensionen bezeichnet:

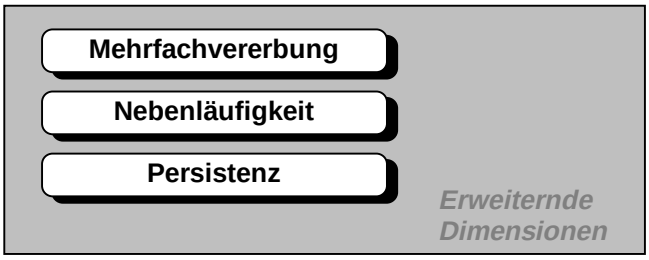


Abbildung 3.2: Erweiternde Dimensionen

Diese Dimensionen sind nicht notwendig für eine objektorientierte Programmiersprache, aber doch hilfreich für die objektorientierte Programmierung.

3.2.1 Mehrfachvererbung

Wird eine Klasse von mehreren Basisklassen abgeleitet, so spricht man von *Mehrfachvererbung*. Erbt eine Klasse dabei (u.U. über mehrere Vererbungsstufen hinweg) mehrfach von derselben Klasse, so wird dies *wiederholtes Erben* genannt.

3.2.2 Nebenläufigkeit

Um den Begriff der Nebenläufigkeit definieren zu können, benötigt man ein Verständnis vom Konzept des Prozesses:

Ein *Prozeß* ist allgemein definiert als (möglichst zweckvolle) Herstellung oder Änderung eines Objektes (in Anlehnung an den in [JV87] definierten Handlungsbegriff). Prozesse haben eine Schnittstelle, bestehend aus ausführbaren Operationen oder Eintrittspunkten, und einen oder mehrere Kontrollstränge, die aktiv oder unterbrochen sein können (nach Wegners Prozeßbegriff).

Nun kann Nebenläufigkeit wie folgt definiert werden:

Eine objektorientierte Programmiersprache ist *nebenläufig*, wenn sie die nebenläufige Ausführung von Prozessen unterstützt.

3.2.3 Persistenz

Persistenz ist die Eigenschaft der Objekte, welche deren Lebensdauer bestimmt. Die Lebensdauer der Objekte sollte länger sein als die Lebensdauer des Programms.

3.3 Der vollständige Dimensionskatalog

In Abbildung 3.3 wird zusammenfassend der gesamte Dimensionskatalog dargestellt:

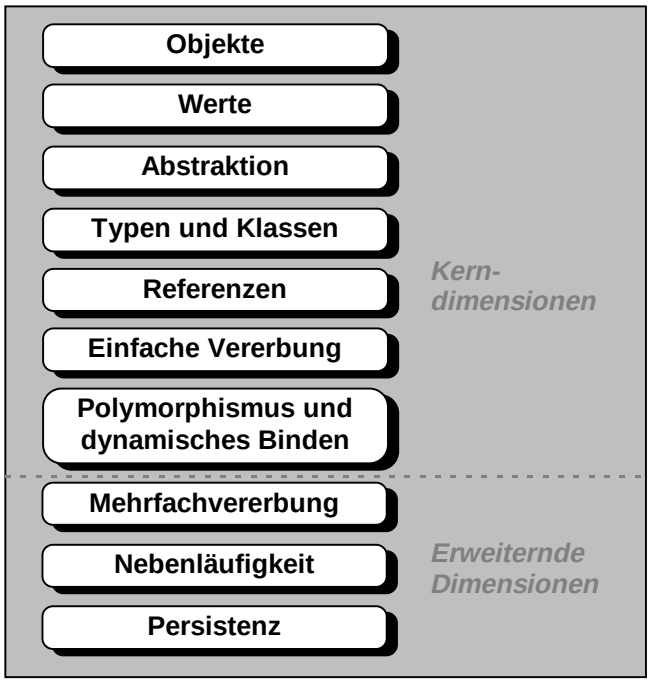
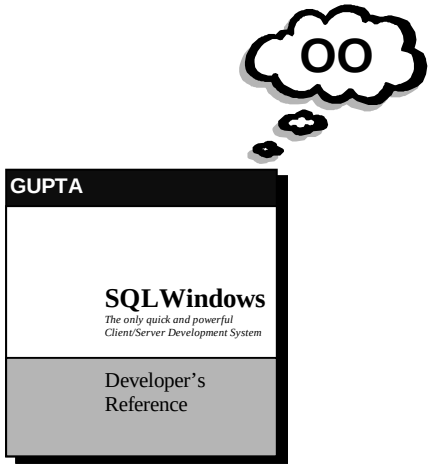


Abbildung 3.3: Dimensionskatalog für objektorientierte Programmiersprachen

An diesem Dimensionskatalog soll nun SQLWindows gemessen werden, um feststellen zu können, ob es sich bei dessen 4GL SAL tatsächlich um eine objektorientierte Programmiersprache handelt. Die objektorientierte Programmierung im SQLWindows-Verständnis und die Untersuchung anhand des Dimensionskataloges sind Inhalt des nächsten Kapitels.

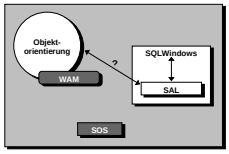
4 Objektorientierte Programmierung im SQLWindows-Verständnis



Die 4GL SAL (SQLWindows Application Language) wurde 1988 als Programmiersprache für die Client/Server-Entwicklungsumgebung SQLWindows eingeführt. Mit dem Erscheinen von SQLWindows 4.0 im Jahre 1994 wurde SAL um objektorientierte Eigenschaften erweitert (vgl. [Nie]).

SAL ist keine eigenständige Programmiersprache – sie ist sehr eng an das Entwicklungssystem SQLWindows gebunden. So wird die Struktur eines SQLWindows-Programms nicht durch die Programmiersprache SAL festgelegt, sondern durch den sogenannten *Outline*, das Editor-Werkzeug von SQLWindows. Die Entwicklungsumgebung von SQLWindows hat einen großen Einfluß auf das Vorgehen bei der Softwareentwicklung. Aus diesem Grund wird in Abschnitt 4.1 zunächst das SQLWindows-Entwicklungssystem vorgestellt, bevor in Abschnitt 4.2 eine SAL-Sprachdefinition gegeben wird. Die Sprachbeschreibung ist notwendig, um die Programmbeispiele und das in Kapitel 5 vorgestellte WAM-Framework verstehen zu können. Die prozeduralen Eigenschaften von SAL werden beschrieben und vor dem Hintergrund der objektorientierten Programmierung bewertet. Anschließend wird in Abschnitt 4.3 versucht, zu jeder der objektorientierten Dimensionen aus Kapitel 3 ein kurzes Beispielprogramm in SAL anzugeben. Hier zeigt sich, inwieweit SAL die einzelnen Dimensionen unterstützt. Abschnitt 4.4 versucht schließlich die Frage zu beantworten, ob bzw. inwieweit SQLWindows eine objektorientierte Sprache im Sinne des Dimensionskataloges ist. Abschnitt 4.5 faßt die Ergebnisse dieses Kapitels zusammen.

Dem mit SQLWindows vertrauten Leser sei folgendes Vorgehen empfohlen: Beginnend mit der Zusammenfassung (Abschnitt 4.5), kann bei weitergehendem Interesse für eine bestimmte Spracheigenschaft bzw. objektorientierte Dimension zum entsprechenden Abschnitt (4.2 bzw. 4.3) verzweigt werden. Zur einfacheren Orientierung sind in der Zusammenfassung immer die jeweiligen Abschnittsnummern angegeben.



Dieses Kapitel zeigt die enge Bindung von SAL an das SQLWindows-Entwicklungssystem auf, die einen großen Einfluß auf die Programmstruktur und die Vorgehensweise bei der Softwareentwicklung hat. Nach der Beschreibung und Bewertung der prozeduralen SAL-Spracheigenschaften wird SAL am Dimensionskatalog gemessen, um schließlich beurteilen zu können, ob bzw. inwieweit SAL eine objektorientierte Programmiersprache ist.

4.1 Das SQLWindows Entwicklungssystem

Das SQLWindows Entwicklungssystem besteht im wesentlichen aus zwei Werkzeugen: Dem *Fenster-Editor* (*Window Editor*) und dem *Source-Editor* (*Outline Window*, kurz: *Outline*)[4]. Mit dem Window Editor erstellt der Benutzer die grafischen Objekte der Applikation, wie Fenster und Kontrollelemente, zu denen SQLWindows automatisch im Outline Code erzeugt. Das Verhalten der Applikation wird im Outline mit Hilfe von SAL (SQLWindows Application Language) und SAM (SQLWindows Application Messages) beschrieben.

Abbildung 4.1 zeigt ein einfaches SQLWindows-Beispielprojekt. Die SQLWindows-Applikation besteht aus einem Hauptfenster („Frame Window“), das eine Schaltfläche „Exit“ enthält, mit der die Applikation beendet werden kann:

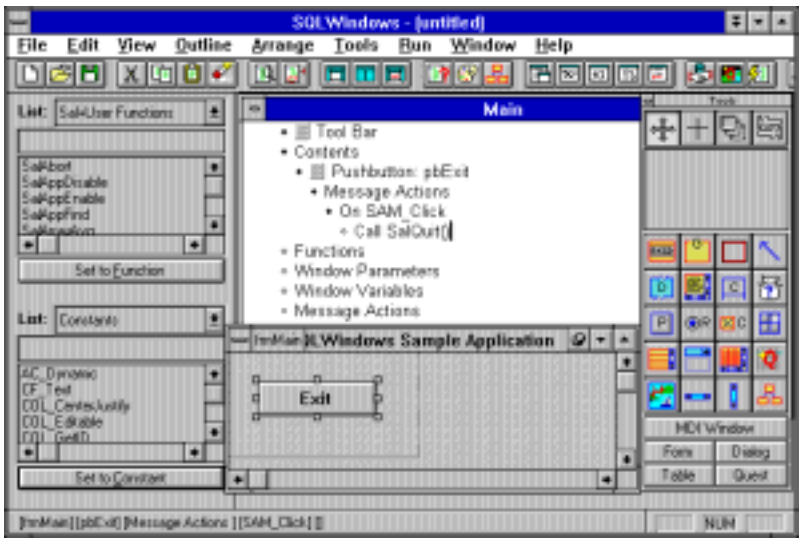


Abbildung 4.1: Die Elemente des SQLWindows-Entwicklungssystems

Die Entwicklung dieser Applikation soll im folgenden in Form eines Szenarios Schritt für Schritt beschrieben werden. Dabei werden auch die einzelnen Entwicklungswerkzeuge von SQLWindows eingeführt. Auch wenn dieses Szenario einen (wenig wissenschaftlichen) Handbuchcharakter hat, so ist es doch notwendig, um zu verstehen, wie sehr sowohl die Struktur eines SQLWindows-Programmes als auch das Vorgehen bei der Softwareentwicklung mit SQLWindows durch die Entwicklungsumgebung geprägt sind.

Szenario: Erstellen einer SQLWindows-Applikation

Nach dem Start von SQLWindows präsentiert sich das Entwicklungssystem mit einem rudimentären Applikationsgerüst im Outline:

[4] Im folgenden werden die englischen Originalbegriffe verwendet.

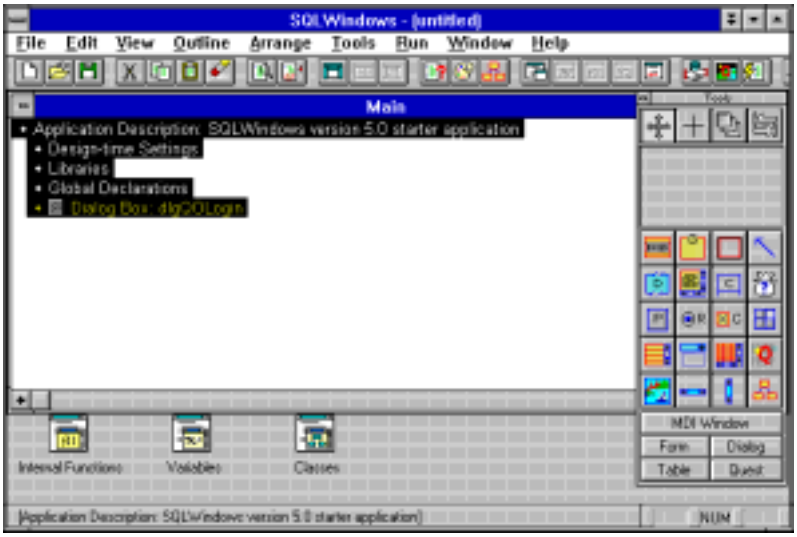


Abbildung 4.2: Standard-Applikationsgerüst

Bereits hier deutet sich der hierarchische Aufbau von SQLWindows-Programmen an – im Outline durch entsprechende Einrückungen visualisiert.

Das Programmgerüst hat bereits eine SQLWindows-Bibliothek eingebunden, mit der die sogenannten QuickObjects genutzt werden können – vorgefertigte Klassen zur schnellen visuellen Applikationsentwicklung. Da diese für das Beispielprogramm nicht benötigt werden, wird die Bibliothek aus dem Programm entfernt.

Für das weitere Vorgehen lassen wir uns von SQLWindows helfen: Nach dem Einschalten der Outline Options geben diese kontextabhängig alle Elemente vor, die in den Outline eingefügt werden können:

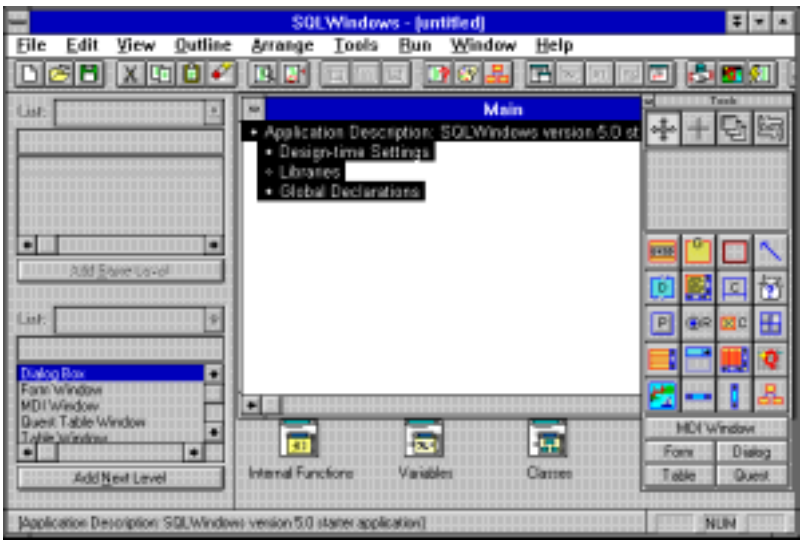


Abbildung 4.3: Outline Options

Die Outline Options vereinfachen – je nach Kontext – das

- Erzeugen von Objekten
- Definieren von Menüs
- Definieren von Funktionen, Konstanten und Variablen
- Schreiben von SAL-Anweisungen
- Setzen von Parametern für Funktionen

Da wir uns auf der obersten Applikationsebene befinden, schlagen die Outline Options vor, ein Hauptfensterobjekt zu erzeugen. Da keine benutzerdefinierten Fensterklassen im Programm vorhanden sind, werden nur die Standard-Fenstertypen zur Auswahl gestellt. Unsere Applikation soll ein einfaches Fenster als Hauptfenster erhalten – in SQLWindows „Form Window“ genannt. Ein Doppelklick auf den entsprechenden Eintrag in den Outline Options (oder alternativ eine Auswahl aus dem Menü *Tools*/*Windows*), und schon erscheint das gewünschte Fensterobjekt in einem neuen Werkzeug, dem Fenster-Editor:

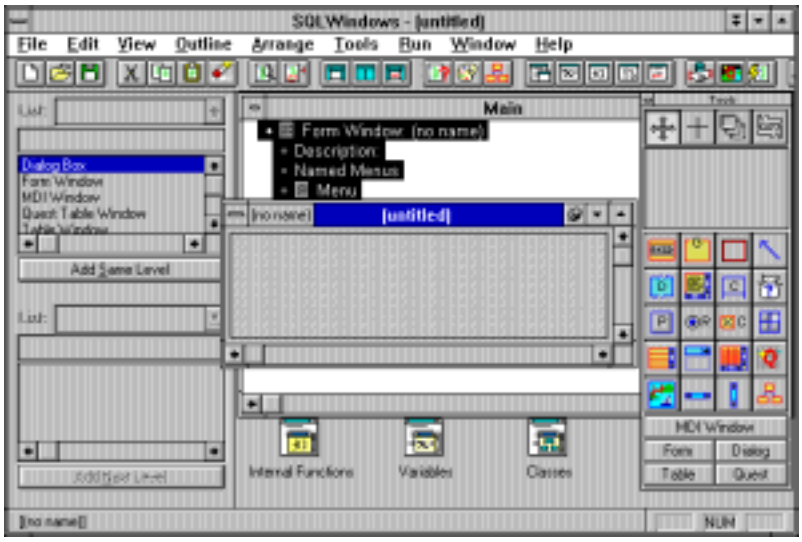


Abbildung 4.4: Das Hauptfenster im Fenster-Editor

Mit dem Erzeugen des Fensters wird automatisch Code im Outline generiert (in obiger Abbildung schwarz hinterlegt): Jede Änderung am Fenster wird automatisch Änderungen am Code nach sich ziehen. Das Fenster ist – wohlgemerkt – ein Objekt, keine Klasse. Es ist in SQLWindows möglich, Änderungen direkt am Objekt durchzuführen, und nicht etwa an der Klasse, von der dieses Objekt erzeugt wurde. Die erste Änderung am Fenster soll jetzt vorgenommen werden: Das Fenster erhält einen Objektnamen (im Fenster-Editor rechts neben dem Systemfeld dargestellt). Da es ein Hauptfenster vom Typ eines Frame Window ist, soll es frmMain heißen. Anstatt jedoch die Änderung direkt im Code durchzuführen, bedienen wir uns des Customizers. Der Customizer ist ein Kontextmenü für grafische Objekte, mit dem die Attribute der grafischen Objekte bearbeitet werden können. Nicht-grafische Attribute wie z.B. Instanz- und Klassenvariablen einer Fensterklasse, auf der ein grafisches Objekt beruht, können nur im Outline bearbeitet werden. Der Cu-

stomizer erscheint durch einen Klick mit der rechten Maustaste auf das Hauptfenster im Fenster-Editor:

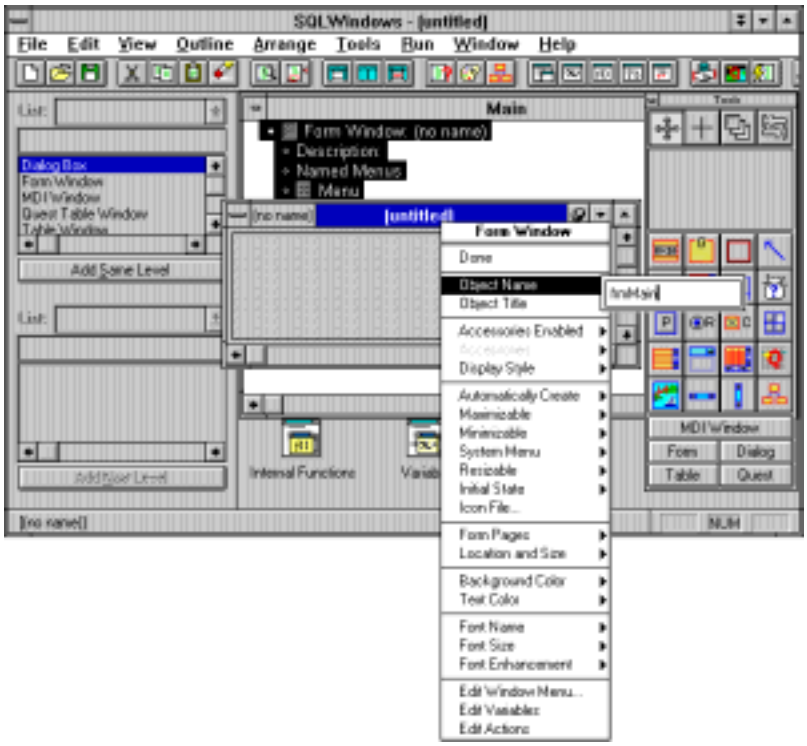


Abbildung 4.5: Customizer

Nach Auswahl des Attributs „Object Name“ kann der Objektname in einem Editierfeld eingegeben werden. In selbiger Weise wird der Fenstertitel geändert. Ein Klick auf „Done“, und die Änderungen werden im Fenster-Editor und im Outline übernommen.

Nun soll das Fenster mit einer Schaltfläche versehen werden, die es später erlaubt, mit einem Klick die Applikation zu beenden. Hier kommt ein weiteres Werkzeug zum Einsatz. Die Werkzeugpalette (Tool Palette) enthält eine Schaltfläche für jedes grafische Objekt, das in SQLWindows benutzt werden kann. Grafische Objekte können in SQLWindows-Fenstern plaziert werden, die im Fenster-Editor erzeugt wurden. Zusätzlich bietet die Palette Schaltflächen für die Werkzeuge Window Grabber, Window Selector, Object Duplicator und Tab Order (zur Manipulation grafischer Objekte) und für den Class Editor.

Nach Auswahl des Schaltflächen (Push Button) - Symbols aus der Werkzeugpalette kann die neue Schaltfläche im Fenster plaziert werden. Über den Schaltflächen-Customizer geben wir ihr den Objektnamen pbExit und den Titel „Exit“:

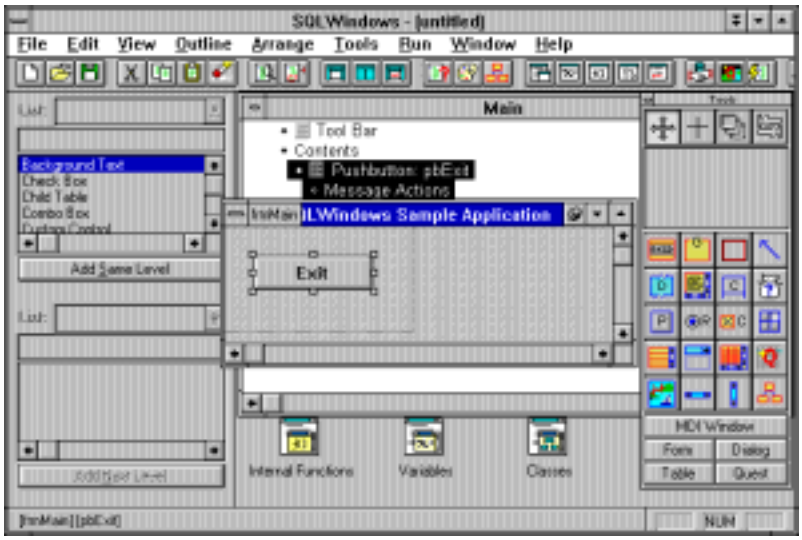


Abbildung 4.6: Das Hauptfenster mit der Schaltfläche „Exit“

Wie bereits gesagt, soll ein Klick auf diese Schaltfläche das Programm beenden.

Wird eine Schaltfläche angeklickt, dann sendet SQLWindows die Nachricht SAM_Click an das Schaltflächen-Objekt. Im Abschnitt Message Actions von pbExit kann eine Reaktion auf diese Nachricht definiert werden. Ein Klick auf die Zeile Message Actions im Outline, und die Outline Options zeigen alle SQLWindows-Nachrichten an, die eine Schaltfläche empfangen kann:

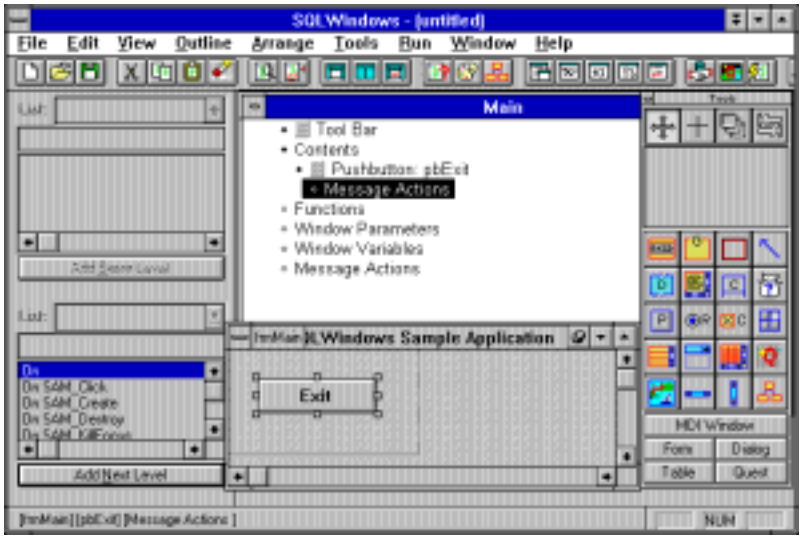


Abbildung 4.7: Nachrichtenbehandlung der Schaltfläche „Exit“

Ein Doppelklick auf On SAM_Click, und die Nachricht wird im Outline eingefügt. Sofort zeigen die Outline Options alle verfügbaren SAL-Schlüsselwörter:

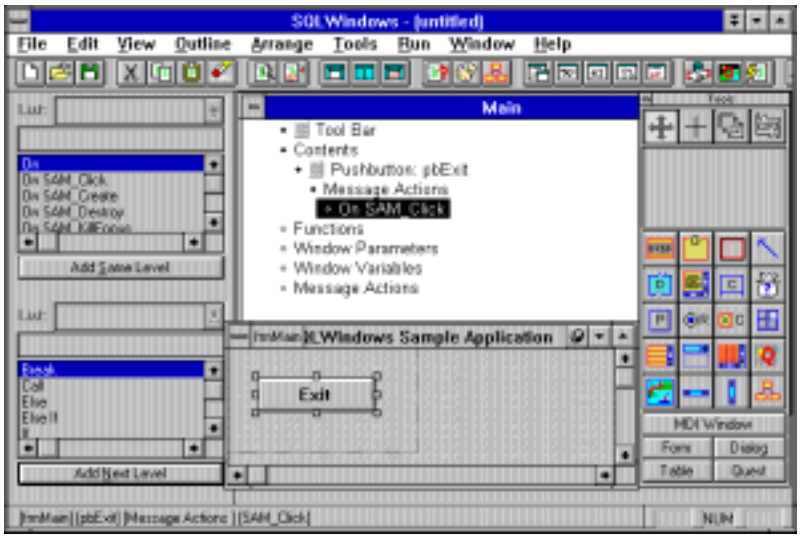


Abbildung 4.8: Verarbeiten der Nachricht SAM_Click

Wir wählen Call zur Einleitung eines Funktionsaufrufes, ergänzen die Programmzeile um die SQLWindows-Systemfunktion SalQuit(), die eine SQLWindows-Applikation beendet (dies hätten wir auch über die Outline Options erledigen können) – und fertig ist die Applikation!

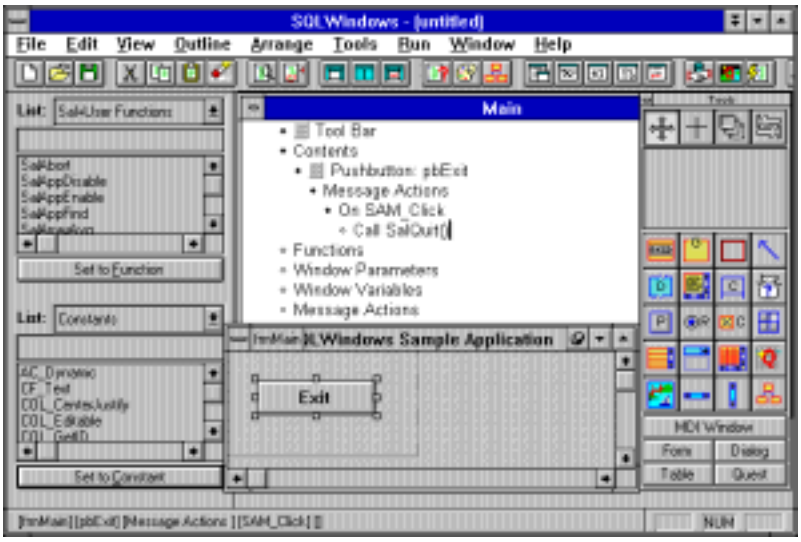


Abbildung 4.9: Beenden der Applikation als Reaktion auf SAM_Click

Deutlich sichtbar ist der hierarchische Aufbau von SQLWindows-Programmen, der vom Outline unterstützt wird durch entsprechende Einrückungen, welche die einzelnen Hierarchieebenen voneinander trennen: Jede Zeile im Outline ist ein sogenanntes *Element* (*item*). Ein Element kann ein *Eltern-Element* (*parent item*) oder ein *Kind-Element* (*child item*) sein.

Der Outline bietet die Möglichkeit zum Expandieren bzw. Zusammenziehen der Programm-Ansicht: Eine gefüllte Raute ♦ vor einem Element zeigt an, daß das Element weitere Kind-Elemente enthält. Ein Element mit einer leeren Raute ◊ hat keine weiteren Kind-Elemente. In der

Statuszeile am unteren Rand des Outline-Fensters wird für die Zeile, in der sich der Cursor momentan befindet, die gesamte Element-Hierarchie aufgelistet.

Die Applikation kann nach der Verarbeitung durch den SQLWindows-Compiler sofort ausgeführt werden:

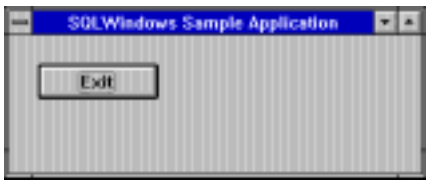


Abbildung 4.10: Die Beispielapplikation

Das Beispiel zeigt, wie schnell man mit SQLWindows Applikationen entwickeln kann. Nicht gezeigt wurde der Einsatz von QuickObjects, den bereits erwähnten vorgefertigten Klassen zur schnellen visuellen Applikationsentwicklung, mit deren Hilfe z.B. auf einfache Art und Weise eine Datenbankanbindung realisiert werden kann. Die QuickObjects zur Datenbankanbindung erlauben die Verknüpfung von Datenbanktabellen und Fenster-Kontrollelementen wie Eingabefeldern und Tabellenfenstern und stellen Standard-Werkzeuge zur Datenbank-Navigation bereit. Da die QuickObjects auch die logische Verknüpfung mehrerer Tabellen erlauben, sind sie auch zur Realisierung komplexer Sachverhalte geeignet. Problematisch an diesem Konzept ist sicherlich die direkte Verbindung von Datenbank und Benutzeroberfläche, die eine solche SQLWindows-Applikation sehr anfällig gegenüber Änderungen am Datenmodell oder an den Interaktionselementen macht. Aus WAM-Sicht ist ferner die fehlende Abstraktion vom Datenmodell zu kritisieren, wodurch die Architektur vom gewählten Persistenzmechanismus (hier: eine relationale Datenbank) abhängig wird.

Das SQLWindows-Entwicklungssystem eignet sich auf jeden Fall für die Erstellung von Prototypen. Dank der schnellen Anbindung von Datenbanken können mit vertretbarem Aufwand Demonstrationsprototypen (im Verständnis von Budde et. al., vgl. [BKKZ92]) mit einer gewissen Grundfunktionalität entwickelt werden. Auch zur Erstellung von Oberflächen-Prototypen als Grundlage der Diskussion über die Benutzeroberfläche zwischen Entwicklern und Anwendern eignet sich SQLWindows, da Änderungen an der Benutzeroberfläche sehr schnell durchgeführt werden können.

Das Beispiel vermittelt auch einen ersten Eindruck von der Programmiersprache von SQLWindows, SAL. Der nächste Abschnitt stellt diese Sprache ausführlich vor.

4.2 Die Programmiersprache SAL (SQLWindows Application Language)

Im folgenden soll SAL (SQLWindows Application Language), die Programmiersprache von SQLWindows, vorgestellt werden, wobei der Schwerpunkt in diesem Abschnitt auf der Grundstruktur der Programme und den prozeduralen Sprachelementen liegt. In Kapitel 4.3 werden dann die objektorientierten Eigenschaften von SAL beschrieben und am Dimensionskatalog aus Kapi-

tel 3 gemessen. Eine vollständige SAL-Sprachbeschreibung findet man in [Gup94a] und [Gup94b].

Eine SAL- Sprachbeschreibung ist an dieser Stelle durchaus notwendig und sinnvoll, da nicht davon ausgegangen werden kann, daß der Leser dieser Arbeit mit SQLWindows und SAL vertraut ist. Um jedoch die Beispiele und das in Kapitel 5 vorgestellte WAM-Framework verstehen zu können, sind zumindest grundlegende SAL-Kenntnisse vonnöten.

SAL ist nach [Gup94b] eine prozedurale Sprache[5], die zur Implementierung von SQLWindows-Aktionen verwendet wird. Diese Aktionen werden als Reaktion auf Ereignisse ausgeführt. SAL-Instruktionen können in bestimmten Abschnitten eines SQLWindows-Programms benutzt werden. Diese Abschnitte werden vom Outline vorgegeben und bilden die Grundstruktur eines SQLWindows-Programms.

4.2.1 Die Grundstruktur eines SQLWindows-Programms

Beginnt man mit einem neuen SQLWindows-Projekt, so gibt der Outline eine Programmstruktur vor, die mindestens[6] aus folgenden Elementen besteht:

- ♦ Application Description:
  - ♦ Design-time Settings
  - ◊ Libraries
  - ♦ Global Declarations
    - ♦ Window Defaults
    - ♦ Formats
    - ◊ External Functions
    - ♦ Constants
    - ◊ Resources
    - ◊ Variables
    - ◊ Internal Functions
    - ◊ Named Menus
    - ◊ Class Definitions
    - ◊ Application Actions

Im Abschnitt Application Description kann die SQLWindows-Applikation textuell beschrieben werden.

Im Abschnitt Design-time Settings werden Einstellungen der Entwicklungsumgebung und des SQLWindows-Compilers gespeichert. Eine SQLWindows-Programmdatei (`*.APP`) ist demnach nicht einfach eine Quelldatei im Sinne von z.B. C++, sondern vielmehr eine Projektbeschreibung, da sie Elemente enthält, die man bei C++ - Projekten in einer Projektdatei (Makefile) oder den Werkzeugeinstellungen der Entwicklungsumgebung wiederfände. Das eigentliche Programm ist nur ein Teil dieser Programmdatei.

---

[5] Prozedural ist SAL in dem Sinne, als daß sich mit dieser Sprache Prozeduren (bzw. Funktionen) implementieren lassen. Zusätzlich bietet SAL Sprachkonstrukte, die eine Programmierung nach dem objektorientierten Paradigma erlauben.

[6] Je nach SQLWindows-Version kann der Outline bestimmte Bibliotheken oder auch bereits komplette Fenster vordefinieren, die aber bei Bedarf gelöscht werden können.

Der Abschnitt Libraries enthält die Namen von (Teilen von) SQLWindows-Bibliotheken (`*.APL`), die in das SQLWindows-Programm eingebunden werden sollen. Das im Rahmen dieser Diplomarbeit entwickelte WAM-Framework bildet eine solche Bibliothek (`TANDEM.APL`), ebenso das in Kapitel 6 vorgestellte SQLWindows Object System (SOS) (`SOS.APL`).

Der Abschnitt Global Declarations definiert

- Window Defaults: Die Standard-Attributwerte für Fenster (Schriftart, Farbe etc.)
- Formats: Formate für Zahlen-, Datums- und Zeitangaben
- External Functions: Externe Funktionen, die in Dynamic Link Libraries (DLLs) definiert sind
- Constants: Globale Systemkonstanten (z.B. Windows-Nachrichtencodes) und globale Benutzerkonstanten
- Resources: Bitmaps, Cursors und Icons
- Variables: Globale Variablen
- Internal Functions: Globale Funktionen des SQLWindows-Programms
- Named Menus: Menüs der Applikationsfenster
- Class Definitions: SQLWindows-Klassendefinitionen
- Application Actions: Bindet die Ausführung von SAL-Instruktionen an das Eintreffen von Nachrichten auf Applikationsebene

SQLWindows-Programmdateien werden in einem speziellen binären Format gespeichert, was die Bearbeitung der Programme mit anderen Editoren unmöglich macht. Der Compiler ist fest in die SQLWindows-Entwicklungsumgebung integriert, so daß man bei der Entwicklung von SQLWindows-Programmen auf diese Entwicklungsumgebung festgelegt ist. Hier vermißt man die Flexibilität anderer Sprachen wie z.B. C++, deren Quelltexte i.d.R. im ASCII-Format gespeichert sind und somit mit den meisten Editoren bearbeitet werden können, und deren Compiler meist eigenständige Programme sind, die man in viele Editoren und Entwicklungsumgebungen einbinden kann.

In einer SQLWindows-Programmdatei wird als zusätzliche Information auch die Versionsnummer des Outline gespeichert. Die Entwicklungsumgebung fragt diese Versionsnummer beim Laden einer Programmdatei ab und bricht mit einer Fehlermeldung ab, falls die SQLWindows-Applikation mit einer neueren SQLWindows-Version erstellt wurde. Die Outline-Versionsnummer wird auch bei kleineren Revisionen einer SQLWindows-Version neu vergeben. Da mir zunächst die kostenlose Solo-Version von SQLWindows in der Version 5.0.1 zur Verfügung stand und ich dann auf eine ältere Vollversion mit der Versionsnummer 5.0 umsteigen wollte, konnte ich meine bereits entwickelten Programme nicht übernehmen. Dies gelang erst, als ich die Version 5.0 durch eine 5.0.1-Vollversion ersetzte. SQLWindows trennt hier nicht sauber zwischen einer Revision der Entwicklungsumgebung und der Revision seiner Sprache SAL. Solche Probleme gibt es bei Sprachen wie C++ nicht. Bei einer Sprachrevision muß hier „lediglich“ ein neuer Compiler entwickelt werden.

Innerhalb der oben beschriebenen Struktur bewegt sich der SQLWindows-Programmierer bei der Applikationsentwicklung. Seine Aufgabe ist es nun, Fenster, Klassen und Funktionen hinzuzufü-

gen und miteinander (und evtl. einer Datenbank) zu verbinden. Dazu bedient er sich der Entwicklungsumgebung und der Programmiersprache SAL.

4.2.2 SAL-Instruktionen

SAL-Instruktionen können in folgenden Abschnitten eines SQLWindows-Programms benutzt werden:

- im Abschnitt Application Actions zur Implementierung der Aktionen als Reaktion auf die applikationsspezifischen Nachrichten SAM_AppStartup, SAM_AppExit, und SAM_SqlError.
- im Abschnitt Actions von internen Funktionen zum Zwecke der Funktionsimplementierung.
- im Abschnitt Actions von Klassenfunktionen zum Zwecke der Funktionsimplementierung.
- im Abschnitt Menu Actions eines Menüeintrags zur Implementierung der Aktionen als Reaktion auf das Anwählen des Menüeintrags.
- im Abschnitt Message Actions eines Objekts bzw. einer Klasse[7].

Die Sprache SAL dient demnach der Beschreibung von Aktionen, also der Dynamik einer SQLWindows-Applikation, wohingegen die Struktur statischer Elemente (Funktionen, Klassen, Objekte[8]) vom Outline beschrieben wird. Die angesprochenen Aktionen bestehen im Aufruf von Funktionen oder in der Manipulation von Variablen eines bestimmten Typs. Der Kontrollfluß kann innerhalb einer Aktion durch steuernde Konstrukte (Schleifen, bedingte Instruktionsausführung) beeinflußt werden. Diese Elemente von SAL – Datentypen, Variablen, Kontrollstrukturen und Funktionen – werden in den nun folgenden Abschnitten beschrieben.

4.2.3 Datentypen

SAL bietet für Variablen folgende Standard-Datentypen an:

- Boolean: Kann einen der vordefinierten Werte TRUE und FALSE annehmen.
- Date/Time: Datums- und Zeitangabe im ISO-Format YYYY-MM-DD-HH.MM.SS.MSMSMS. Die Date/Time-Arithmetik umfaßt die Addition und Subtraktion von Number-Werten (die Number-Werte repräsentieren Tage) und die Subtraktion von einem weiteren Date/Time-Wert.
- Number: Ganzzahl oder Fließkommazahl mit einer Genauigkeit von bis zu 22 Stellen.
- String: Zeichenkette; die Länge ist lediglich durch den verfügbaren Speicher beschränkt.
- Long String: Zeichenkette; die Länge ist lediglich durch den verfügbaren Speicher beschränkt. Dieser Datentyp wird verwendet, um Zeichenketten aus Datenbanken zu lesen, die länger als 254 Bytes sind.
- Sql Handle: Identifikator für eine offene Datenbankverbindung.
- File Handle: Identifikator für eine offene Datei.
- Window Handle: Identifikator für eine Fensterinstanz.

---

[7] Zur SQLWindows-Sicht der Begriffe „Objekt“ und „Klasse“ siehe Abschnitt 4.3.1 und 4.3.4.
[8] Ein SQLWindows-Objekt ist insofern statisch, als es im Outline definiert werden kann (vgl. Abschnitt 4.1).

Konstanten können vom Typ Boolean, Date/Time, Number oder String sein.

Auffällig an den Datentypen von SAL ist die Orientierung an den in Datenbanken üblichen Datentypen (z.B. Long String), was für ein Client/Server-Datenbank-Entwicklungssystem jedoch nicht ungewöhnlich ist. Bemerkenswert ist aber der Datentyp Window Handle, welcher die Herkunft von SQLWindows sehr deutlich macht: Dieser Datentyp ist ein originalgetreues Abbild des *Fensterbezugs* (*window handle*) von Microsoft Windows. Mit der Identifikation von Fenstern über solche Fensterbezüge bindet sich SQLWindows eng an dieses Betriebssystem.

Die benutzerdefinierten Typen in SQLWindows beschränken sich auf die Definition von Klassen, die in Abschnitt 4.3.4 beschrieben ist. Es ist nicht möglich, Aufzählungs-, Unterbereichs- oder Rekord-Typen zu definieren.

Von allen Standard-Datentypen und von benutzerdefinierten Variablen (siehe Abschnitt 4.3.1) können Arrays angelegt werden. SQLWindows erlaubt sowohl statische als auch dynamische Arrays und stellt einen Satz von Funktionen zur Array-Sondierung und -Bearbeitung zur Verfügung.

Der Datentyp Date/Time zeigt zum einen die nahe Verwandtschaft von SQLWindows zu Datenbanksystemen und läßt zum anderen den prozeduralen Ursprung von SQLWindows erkennen. Zur Bearbeitung von Date/Time-Variablen hält SQLWindows eine Reihe von Systemfunktionen vor. So extrahiert z.B. die Funktion SalDateYear() die Jahreszahl aus einem Date/Time-Wert. In objektorientierten Programmiersprachen hätte man anstatt eines Datentyps Date/Time zwei Klassen Date und Time definiert, wobei erstere z.B. eine Methode GetYear() zur Verfügung stellte, mit der man die Jahreszahl eines Date-Objektes ermitteln könnte.

4.2.4 Variablen

Variablen werden durch folgende Syntax deklariert:

Datentyp: *Variablenname*

In SQLWindows gilt die Regel „declaration before use“: Eine Variable darf im Programm erst dann verwendet werden, wenn sie bereits deklariert wurde.

Neben den in 4.2.3 aufgeführten Datentypen sind auch grafische Objekttypen zulässig. Als Beispiel soll folgende Deklaration eines Fensters dienen:

```
♦Form Window: frm1
    ...
    ♦Contents
        ♦Data Field: dfName
        ...
```

Data Field ist ein grafisches Objekt, das unter dem Namen dfName angesprochen werden kann.

Auf Variablen kann durch eine *unqualifizierte Referenz* (*einfache Referenz*) oder eine *qualifizierte Referenz* zugegriffen werden. Im ersten Fall wird lediglich der Variablenname angegeben, während im zweiten Fall eines oder mehrere Präfixe die Variable (genauer: ihren Kontext) näher spezifizieren. Eine vollständig qualifizierte Referenz beinhaltet einen Fensterbezug und einen Fensterobjektnamen, wie z.B.

```
hWnd1.frm1.dfName
```

Der Gültigkeitsbereich einer Variablen ist abhängig vom Ort ihrer Deklaration:

- Für globale Variablen erstreckt sich der Gültigkeitsbereich auf das gesamte Programm, inklusive aller eingebundenen SQLWindows-Bibliotheken.
- Für Variablen, die als lokale Variablen einer Funktion deklariert wurden, ist der Gültigkeitsbereich der Rumpf dieser Funktion. Die Funktion kann eine globale Funktion oder eine Klassen- oder Objektfunktion[9] sein.
- Variablen, die als Klassen- oder Instanzvariablen einer Klasse oder als Fenstervariablen eines Fensterobjektes deklariert sind, haben die Klasse bzw. das Fensterobjekt als Gültigkeitsbereich.

In SQLWindows spielt es keine Rolle, ob eine Variable, eine Klasse oder ein Objekt im Hauptprogramm oder in einer SQLWindows-Bibliothek definiert ist. Beim Einbinden der Bibliotheken in das Hauptprogramm findet eine „logische Verschmelzung“ statt: Eine Variable, die in einer Bibliothek global deklariert ist, steht jetzt auch dem Hauptprogramm und anderen Bibliotheken als globale Variable zur Verfügung. Außerdem können alle eingebundenen Bibliotheken auf die Variablen, Klassen und Objekte des Hauptprogramms und anderer Bibliotheken zugreifen.

4.2.5 Kontrollstrukturen

Die Kontrollstrukturen von SQLWindows werden im folgenden aufgeführt und kurz erläutert, meist anhand eines Beispiels, um dem Leser dieser Arbeit das Verständnis der SQLWindows-Programme und -Funktionen zu erleichtern, die in dieser Arbeit beschrieben sind. Die Beispiele stammen aus [Gup94a].

Break [*Schleifenname*]

Beendet eine Schleifenanweisung (Loop). Der Name der zu beendenden Schleife kann explizit genannt werden (siehe Loop-Beispiel).

Call *Funktionsname* (*Parameter 1*, *Parameter 2*, ...)

Führt eine Funktion aus. Das Schlüsselwort Call darf nicht weggelassen werden – eine unnötige Forderung, da andere imperative Sprachen wie z.B. Modula-2 auf ein Schlüsselwort zum Funktionsaufruf verzichten oder dieses – wie im Falle von Microsoft QuickBASIC – optional anbieten.

If / Else / Else If

Bedingte Anweisung.

♦ If *Ausdruck1*
  ◊ *Anweisung1*
♦ Else If *Ausdruck2*
  ◊ *Anweisung2*

[9] Der prototypische Ansatz von SQLWindows erlaubt die Definition von (Fenster-)Objekten, innerhalb derer Funktionen definiert werden können. Zum Klassen- und Objektbegriff von SQLWindows siehe Abschnitt 4.3.1 und 4.3.4.

♦Else
◊ *Anweisung3*

Loop

Wiederholt die Anweisungen innerhalb der Schleife, bis eine Break- oder eine Return-Anweisung ausgeführt wird.

```
◊ Set y = 0
◊ Set z = 0
♦Loop Outer
    ♦If z > 10
        ◊ Break
    ♦Loop Inner
        ♦If y > 10
            ◊ Break
        ♦If GetData = 10
            ◊ Break Outer
        ◊ Set GetData = ValCheck()
        ◊ Set y = y + 1
        ◊ Set z = z + 1
```

Die Break-Anweisungen geben den Kontrollfluß immer an den nächstäußeren Block zurück. Um aus der inneren Schleife heraus die gesamte Schleife verlassen zu können, muß Break mit einem Schleifennamen qualifiziert werden – hier der Name der äußeren Schleife, Outer.

Nachteilig an diesem Konstrukt ist vor allem, daß die Schleifenabbruchbedingungen syntaktisch nicht hervorgehoben werden und über die gesamte Schleife verteilt werden können, was die Lesbarkeit einer solchen Schleife stark beeinträchtigt.

On *Nachricht*

Bindet die Ausführung von SAL-Anweisungen an das Eintreffen einer Nachricht.

```
♦Application Actions
    ♦On SAM_AppStartup
        ◊ Call SalModalDialog(dlgLogBox, hWndNULL)
```

Das On-Konstrukt ist keine Kontrollstruktur im eigentlichen Sinne, sondern kann als eine Art Funktionsaufruf betrachtet werden. Ihm ist deshalb ein eigener Abschnitt (4.2.8) gewidmet.

Return *Rückgabewert*

Unterbricht den Kontrollfluß und liefert einen Wert an den Aufrufenden zurück. Return beendet

- eine Function
- eine Message Action
- eine Menu Action

Select Case

Auswahlkonstrukt.

- Select Case (SalDateQuarter(dtDate))
  - Case 1
    - ◊ Set strQuarter = ’First Quarter’
    - ◊ Break
  - Case 2
    - ◊ Set strQuarter = ‘Second Quarter'
    - ◊ Break
  - Case 3
    - ◊ Set strQuarter = 'Third Quarter'
    - ◊ Break
  - Case 4
    - ◊ Set strQuarter = 'Fourth Quarter'
    - ◊ Break
  - Default
    - ◊ Set strQuarter = 'Error'

Als Case-Ausdrücke sind nur Konstanten und numerische Werte erlaubt, aber keine Variablen.

Set *Variablenname = Ausdruck*

Weist einer Variablen einen Wert zu. Dabei müssen der Variablentyp und der Typ des zugewiesenen Ausdrucks identisch sein. Für grafische Objekte gibt es jedoch eine Art Typkonvertierung: Der Name des grafischen Objektes, z.B. eines Data Field, steht bei einer Wertzuweisung stellvertretend für seinen Inhalt, wobei der Typ seines Inhalts (Number, String, etc.) als Attribut des grafischen Objektes festgelegt wird. Ein Beispiel:

- Form Window: frm1
  ...
  - Contents
    - Data Field: dfName
      ...

- Form Window: frm2
  ...
  - Window Variables
    - ◊ String: sName
      ...

dfName sei ein Datenfeld, das zur Eingabe von Zeichenketten dient. Der Inhalt des Datenfeldes kann nun wie folgt an die Variable sName zugewiesen werden:

```
Set sName = frm1.dfName
```

When SqlError

Ausnahmebehandlung für fehlgeschlagene SQL-Funktionen.

While

Wiederholt die Anweisungen innerhalb der Schleife, bis die Abbruchbedingung den Wert FALSE annimmt.

◊ Set n = 0
♦ While n < 10
    ◊ Set nArray[n] = 0
    ◊ Set n = n + 1

4.2.6 Kommentare

Kommentare beginnen mit einem Ausrufungszeichen '!'. Sie können sich über eine ganze Zeile erstrecken oder innerhalb der Zeile beginnen, reichen aber immer bis zum Ende der Zeile. Kommentare können die hierarchische Strukturierung des Outline nutzen, einschließlich der Möglichkeit des Ausblendens von Hierarchieebenen. Damit wird dem SQLWindows-Programmierer ein leistungsfähiges Werkzeug zur Quellcode-Dokumentation an die Hand gegeben. Wie später gezeigt wird, ist dies auch notwendig, wenn man wohlstrukturierte und verständliche SQLWindows-Programme schreiben möchte.

4.2.7 Funktionen

SQLWindows faßt unter dem Begriff „Funktion“ sowohl Prozeduren als auch Funktionen i.e.S. zusammen. Dabei werden (im Rahmen dieser Arbeit) *Prozeduren* als syntaktische und semantische Einheiten verstanden, bestehend aus einem Prozedurkopf mit Formalparametern und einem Prozedurrumpf mit Anweisungen, die eine bestimmte Aktion ausführen. *Funktionen* hingegen errechnen einen Wert (vgl. z.B. [Mey90], [Dud88]). Meyer verwendet den Begriff *Routine* als Oberbegriff für Prozeduren und Funktionen.

Diese strenge Unterscheidung zwischen Prozeduren und Funktionen ist von theoretischer Natur. Betrachtet man die Softwareentwicklung in der Praxis, so haben Routinen, die einen prozeduralen Charakter aufweisen, oftmals auch Rückgabewerte, die z.B. die erfolgreiche Ausführung der Prozedur dokumentieren. Ein Beispiel: Die Standard-Bibliothek `stdio.h` der Programmiersprache C enthält eine Funktion fputs (vgl. [KR90]):

```
int fputs(const char *s, FILE *stream)
```

Diese Funktion[10] ist eigentlich eine Prozedur, die eine Zeichenkette s in die Datei stream schreibt. Der Rückgabewert ist kein „errechneter Wert“ im Sinne der obigen Definition, sondern ein nicht-negativer Wert, wenn die Zeichenkette in die Datei geschrieben werden konnte, oder der Wert der Konstanten EOF, wenn ein Fehler aufgetreten ist. Dieses Beispiel sollte zeigen, daß die Grenze zwischen Prozeduren und Funktionen in der Praxis nicht immer so scharf gezogen werden kann wie in obiger Definition.

[10] In C gibt es keinen Prozedurbegriff – sowohl Funktionen als auch Prozeduren (nach obiger Definition) werden als „Funktionen“ bezeichnet.

SQLWindows unterscheidet zwischen fünf Funktionstypen:

- Die *Systemfunktionen* gehören zum SQLWindows-Entwicklungssystem. Funktionen mit dem Präfix Sal dienen der Fensterverwaltung, Eingabefeldvalidierung und der Datei-, Listen- und Zeichenkettenverwaltung. Funktionen mit dem Präfix Sql führen Datenbankoperationen aus.
- *Externe Funktionen* sind in Dynamic Link Libraries (DLLs) definierte Funktionen, die innerhalb von SQLWindows benutzt werden können.
- *Interne Funktionen* sind die globalen Funktionen eines SQLWindows-Programms.
- *Fensterfunktionen* werden innerhalb von Fensterdeklarationen definiert.
- *Klassenfunktionen* werden innerhalb von Klassendeklarationen definiert.

Im Outline besitzt eine Funktion folgende Struktur:

```
♦ Function: Funktionsname
    ◊ Description:
    ◊ Returns
    ◊ Parameters
    ◊ Static Variables
    ◊ Local variables
    ◊ Actions
```

Nach der Funktionsbeschreibung (Description) wird der Rückgabewert der Funktion angegeben (Returns). Parameter (Parameters) werden durch einen Formalparameter und einen Datentyp spezifiziert. Parameter werden normalerweise als Wert übergeben (*call by value*). Das Schlüsselwort Receive vor dem Datentyp veranlaßt jedoch eine Referenzübergabe (*call by reference*):

```
◊ Function: Inc
    ◊ Parameters
        ♦ Receive Number: nNumber
        ♦ Number: nIncrement
    ◊ Actions
        ♦ Set nNumber = nNumber + nIncrement
```

Der Aufruf Inc(nCounter, 10) erhöht den Wert der Variablen nCounter um 10, da nCounter als Referenz an Inc() übergeben wurde (zum Thema Referenzen siehe auch Kapitel 4.3.5).

Als Rückgabewert sind nur die in Abschnitt 4.2.3 beschriebenen Standard-Datentypen zulässig. Als Parameter können zusätzlich auch benutzerdefinierte Variablen (siehe Abschnitt 4.3.1) angegeben werden, die jedoch immer per Referenz übergeben werden.

Statische Variablen (Static Variables) sind lokale Variablen der Funktion, die ihren Wert auch zwischen den Funktionsaufrufen beibehalten. Demgegenüber verlieren die lokalen Variablen (Local variables) ihren Wert nach Verlassen der Funktion. Der Abschnitt Actions bildet den Funktionsrumpf.

4.2.8 Nachrichtenbehandlung (Message Actions)

Fensterklassen enthalten gegenüber funktionalen Klassen einen Abschnitt Message Actions, in dem sie Aktionen an das Eintreffen einer Nachricht binden können:

```
♦Message Actions
    ♦On SAM_Create
        ◊ Call PopulateTable()
```

Auf Applikationsebene existiert ein ähnlicher Abschnitt Application Actions, in dem auf die Nachrichten SAM_AppStartup, SAM_AppExit und SAM_SqlError reagiert werden kann.

Es gibt drei Arten von Nachrichten:

- SQLWindows Application Messages (SAM_*)
- Windows Messages (WM_*)
- Benutzerdefinierte Nachrichten

Einige Nachrichten benutzen die Systemvariablen wParam und lParam zur Parameterübergabe. Spätestens hier wird deutlich, daß SQLWindows den Nachrichtenmechanismus von Microsoft Windows adaptiert hat. Auch hier wird die enge Verbundenheit mit diesem Betriebssystem deutlich.

Die Nachrichtenbehandlung ist ein Sprachkonstrukt, das den meisten Programmiersprachen fehlt. Man kann die Nachrichtenbehandlung jedoch auch als eine Art von Funktionsaufrufen verstehen, wobei die Funktionsnamen (d.h. die Nachrichten) vorgegeben sind (abgesehen von den benutzerdefinierten Nachrichten). Mit Hilfe der Systemvariablen wParam und lParam können sogar Parameter an die Funktionen übergeben werden. Die Nachricht SAM_Create aus obigem Beispiel könnte man demnach wie folgt als Funktion schreiben:

```
♦Function: SAMCreate
    ◊ Description:
    ◊ Returns
    ♦Parameters
        ◊ Number: wParam
        ◊ Number: lParam
    ◊ Static Variables
    ◊ Local variables
    ♦Actions
        ◊ Call PopulateTable()
```

Mit dieser Betrachtungsweise erspart man sich eine gesonderte Berücksichtigung der Nachrichtenbehandlung.

4.2.9 Objektorientierte Sprachelemente

Die bisher behandelten SAL-Sprachelemente beschreiben den historisch gewachsenen prozeduralen Charakter dieser Sprache. Wie bereits erwähnt, wurde SAL im Jahre 1994 um objektorientierte Eigenschaften erweitert.

Nach [Gup94c] implementiert SQLWindows folgende Konzepte objektorientierter Programmierung:

- Objekte
- Klassen
- Vererbung (einfach und mehrfach)
- Dynamisches Binden (Polymorphismus)

Die weiterführende SQLWindows-Literatur (z.B. [Hol95]) führt ebenfalls nur die oben genannten Konzepte auf, ohne sie detaillierter zu beschreiben, als es in [Gup94c] der Fall ist.

Der Ovum-Report „Ovum Evaluates: 4GLs“ ([Ovum94]) beinhaltet auch eine Untersuchung von SQLWindows. Dem Report zufolge lassen sich folgende objektorientierte Konzepte in SQLWindows identifizieren:

- Klassen
- Vererbung
- Polymorphismus
- Objektidentität
- Methodenkapselung
- Abstraktion (Data Hiding)
- Nachrichtenmechanismus (Messaging)

Der Report nennt also wesentlich mehr Konzepte als die Originalliteratur.

Alle genannten Konzepte (abgesehen vom Nachrichtenmechanismus) bilden eine Teilmenge des Dimensionskataloges aus Kapitel 3. SAL soll nun auf die Erfüllung der einzelnen Dimensionen hin untersucht werden.

4.3 Identifikation objektorientierter Dimensionen in SAL

Im folgenden wird SQLWindows am Dimensionskatalog für objektorientierte Programmiersprachen gemessen, der in Kapitel 3 beschrieben ist.

Für jede Dimension werden zunächst die entsprechenden SAL-Konstrukte beschrieben und erläutert, gegebenenfalls unter Zuhilfenahme von Beispielen. Das Waagensymbol am Seitenrand kennzeichnet die Diskussion, ob bzw. inwieweit die jeweilige Dimension in SQLWindows identifiziert werden kann.

4.3.1 Objekte

4.3.1.1 Objektarten

SQLWindows unterscheidet drei Typen von Objekten:

- *Standard-Fensterobjekte* sind Objekte der Standard-Fenstertypen von SQLWindows.
- *Benutzerdefinierte Fensterobjekte* sind Instanzen von benutzerdefinierten Fensterklassen.
- *Benutzerdefinierte Variablen* sind Instanzen von benutzerdefinierten funktionalen Klassen.

Bei den Standard-Fensterobjekten wird weiter unterschieden zwischen *Hauptfenstern* (*top-level objects*) und *Kindfenstern* (*child objects*).

Es gibt vier Hauptfensterarten:

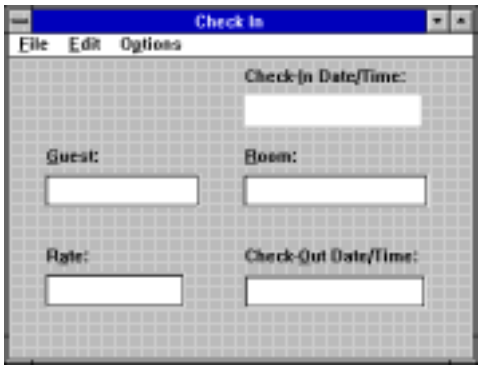


Abbildung 4.11: Form Window

Form Windows dienen der Eingabe und der Darstellung von Daten. Innerhalb eines Form Windows können Kindfenster wie Datenfelder, Schaltflächen etc. plaziert werden. Ein Form Window modelliert somit eine Art Formular – daher wohl auch der Name.

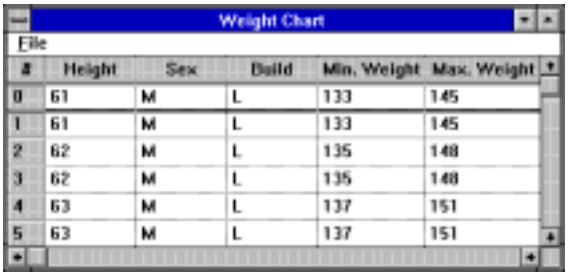


Abbildung 4.12: Table Window

Table Windows zeigen Daten in Form von Tabellen, die (Teilen von) Datenbanktabellen in relationalen Datenbanken entsprechen können.

Quest Windows (ohne Abbildung) dienen der Abfrage und Bearbeitung von Daten aus Datenbanken. Sie arbeiten auf sogenannten *query activities* und *table activities*, die im Programm *Quest* erzeugt werden, das zum Lieferumfang von SQLWindows gehört.

Abbildung 4.13: Dialog Box

Dialog Boxes dienen der Eingabe von Daten durch den Benutzer und der Anzeige von Informationen. Innerhalb einer Dialog Box können Kindfenster wie Datenfelder, Schaltflächen etc. plaziert werden.

Table Windows und Quest Windows können auch als Kindfenster benutzt werden.

MDI Windows (ohne Abbildung) gehören zu den Hauptfenstern, allerdings stehen sie noch eine Hierarchieebene über den soeben eingeführten Hauptfenstern, da sie diese – im Sinne des Multiple Document Interface (MDI) von Microsoft Windows – als Fenster enthalten können.

Die Kindfenster sollen hier nur tabellarisch aufgelistet werden; Tabelle 4.1 zeigt außerdem, in welchen Hauptfenstern die unterschiedlichen Kindfenster enthalten sein können.

Hauptfenster	**Kindfenster**
Form window Dialog Box	Background Text Group Box Frame Line Data Field Multiline Field Push Button Radio Button Check Box Option Button List Box Combo Box Table Window Quest Window Picture Scroll Bar (horizontal/vertical) Custom Control
Table window	Column

Tabelle 4.1: Typen von Kindfenstern

4.3.1.2 Objekterzeugung

Instanzen von Hauptfenstern

Hauptfenster-Objekte werden erzeugt, indem der entsprechende Fenstertyp aus dem SQLWindows-Menü *Tools/Windows* oder den Outline Options ausgewählt wird. Das ausgewählte Fensterobjekt wird (im Fenster-Editor) grafisch dargestellt und gleichzeitig im Outline eingetragen. Änderungen an der grafischen Darstellung werden automatisch im Outline nachgezogen, und umgekehrt. Innerhalb eines Hauptfensters können Kindfenster-Objekte plaziert werden, die aus der Tool Palette ausgewählt werden können. Applikations-Hauptfenster besitzen ein Attribut Automatically Create, das angibt, ob das Fenster beim Start der SQLWindows-Applikation automatisch erzeugt und angezeigt werden soll.

Ein Fenster, das nicht automatisch erzeugt wird, kann mit Hilfe der SAL-Funktion SalCreateWindow() manuell erzeugt werden. Mit diesem manuellen Erzeugungsmechanismus ist es möglich, mehrere Instanzen eines Fensterobjektes zu erzeugen, wie der folgende Ausschnitt aus dem Beispielprogramm `MULTINST.APP` zeigt:

```
♦Form Window: frmAWindow
    ◊ Description:
    ◊ Named Menus
    ◊ Menu
    ♦ Tool Bar
    ♦ Contents
        ◊ Background Text: Every click on the 'Dialog Box' ...
        ♦ Pushbutton: pbDialogBox
            ♦ Message Actions
                ♦ On SAM_Click
                    ◊ Call NewDialogBox()
        ♦ Pushbutton: pbExit
    ♦ Functions
        ♦ Function: NewDialogBox
            ◊ Description:
            ◊ Returns
            ◊ Parameters
            ◊ Static Variables
            ◊ Local variables
            ♦ Actions
                ◊ Set nInstance = nInstance + 1
                ◊ Call SalCreateWindow(dlgADialogBox, hWndForm, nInstance)
    ◊ Window Parameters
    ♦ Window Variables
        ◊ Number: nInstance
    ◊ Message Actions
◊ Dialog Box: dlgADialogBox
    ◊ Description:
    ♦ Tool Bar
    ♦ Contents
        ◊ Background Text: Instance No.
```

◊ Data Field: dfInstanceNo
◊ Background Text: Window Handle:
◊ Data Field: dfWindowHandle
♦ Pushbutton: pbClose
♦ Message Actions
♦ On SAM_Click
◊ Call SalDestroyWindow(hWndForm)
◊ Functions
♦ Window Parameters
◊ Number: nInstance
◊ Window Variables
♦ Message Actions
♦ On SAM_Create
◊ Set dfInstanceNo = nInstance
◊ Set dfWindowHandle = SalWindowHandleToNumber(hWndForm)

Das Hauptfenster frmAWindow enthält eine Schaltfläche pbDialogBox. Bei einem Klick auf diese Schaltfläche wird die Funktion NewDialogBox() aufgerufen, die mittels SalCreateWindow() eine neue Instanz des Dialog Box-Objektes[11] dlgADialogBox erzeugt und an diese eine Instanznummer übergibt, die in der Dialog Box – gemeinsam mit dem Fensterbezug der Dialog Box - Instanz – angezeigt wird:

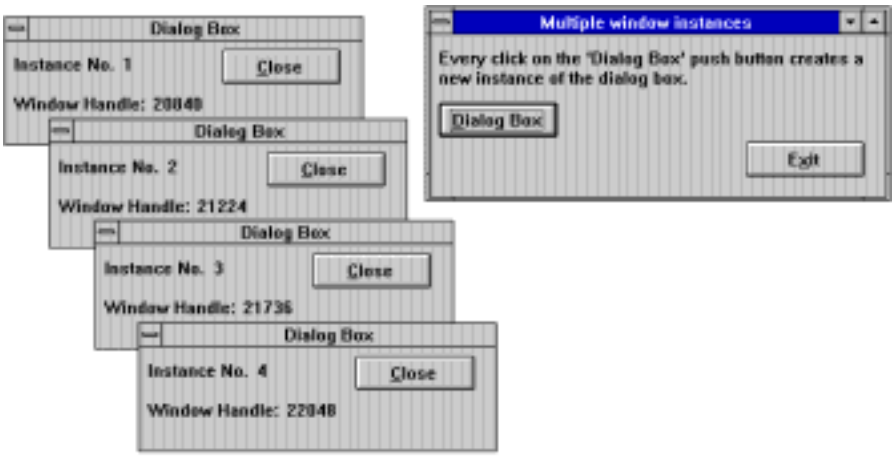


Abbildung 4.14: Mehrere Instanzen eines Fensterobjektes

Die einzelnen Dialog Box - Instanzen müssen voll qualifiziert referenziert werden, d.h. unter Angabe des Fensterbezuges und des Objektnamens, da der Objektname (dlgADialogBox) allein zur eindeutigen Referenzierung nicht mehr ausreicht.

Mit der Möglichkeit, von einem Hauptfenster-Objekt mehrere Instanzen zu erzeugen, wird das Objektkonzept sehr unsauber und führt zu den oben genannten Referenzierungsproblemen. Es gibt jedoch eine Möglichkeit, die Mehrfachinstantiierung zu verbieten: Der Abschnitt Design-time Settings im Outline enthält einen Eintrag Reject Multiple Window Instances?; wird dieser auf

[11] Es handelt sich tatsächlich um die Instanz eines Objektes und nicht etwa einer Klasse!

den Wert Yes gesetzt, so kann von einem Hauptfenster-Objekt maximal eine Instanz erzeugt werden.

Instanzen von Kindfenstern

Kindfenster werden automatisch beim Erzeugen des Hauptfensters, in dem sie enthalten sind, erzeugt.

Instanzen von benutzerdefinierten Fensterklassen

Im Gegensatz zu den oben beschriebenen Fensterobjekten, die auf den SQLWindows-Standard-Fenstertypen basieren, bilden benutzerdefinierte Fensterklassen die Grundlage für benutzerdefinierte Fensterobjekte. Fensterklassen werden in Abschnitt 4.3.4.1 beschrieben. Für Objekte von Fensterklassen gelten die oben genannten Eigenschaften, abhängig davon, ob sie auf Hauptfenster- oder Kindfenster-Typen basieren.

Instanzen von funktionalen Klassen

Funktionale Klassen (Beschreibung siehe Abschnitt 4.3.4.2) können nicht – wie Fensterklassen – dynamisch instantiiert werden – sie existieren nur in Form von *benutzerdefinierten Variablen* (*user-defined variables*). Diese sind mit Instanzen einer Klasse, also herkömmlichen Objekten (im Sprachgebrauch der objektorientierten Methodik) vergleichbar, allerdings mit einer wichtigen Einschränkung: In SQLWindows können keine Referenzen auf benutzerdefinierte Variablen gespeichert werden.

Benutzerdefinierte Variablen können per Parameter an Funktionen übergeben werden. Dabei wird der Formalparameter mit der funktionalen Klasse typisiert. Eine Funktion, die ein Objekt einer funktionalen Klasse fcMyFunctionalClass als Parameter übergeben bekommt, sähe wie folgt aus:

- Function: MyFunction
  - Description:
  - Returns
  - Parameters
    - fcMyFunctionalClass: udvMyUserDefinedVariable
  - Static Variables
  - Local variables
  - Actions

Benutzerdefinierte Variablen werden immer per Referenz übergeben. Der Formalparameter udvMyUserDefinedVariable im obigen Beispiel bezeichnet also tatsächlich einen Verweis auf ein fcMyFunctionalClass-Objekt. Ein solcher Verweis auf eine benutzerdefinierte Variable kann jedoch nicht innerhalb einer Funktion, einer Klasse oder eines Objektes gespeichert werden – man kann udvMyUserDefinedVariable nur während der Ausführungszeit der Funktion MyFunction() benutzen. Wird die Funktion beendet, so kann auf udvMyUserDefinedVariable – wie auch auf jeden anderen Formalparameter – nicht mehr zugegriffen werden; es gibt in SAL kein sprachliches Mittel, um diesen Verweis zur späteren Verwendung zu speichern.

Die soeben beschriebene Einschränkung hat ihre Ursache im Fehlen eines Objektreferenzkonzeptes in SQLWindows. Diese Problematik wird ausführlich in Kapitel 6 behandelt.

Es sei noch angemerkt, daß (Referenzen auf) Objekte von Fensterklassen nicht per Parameter an Funktionen übergeben werden können. Um ein Fensterobjekt an eine Funktion zu übergeben, bedient man sich des Fensterbezuges, über den ein Fensterobjekt eindeutig referenziert ist:

- Function: MyOtherFunction
  - Description:
  - Returns
  - Parameters
    - Window Handle: hWndMyWindow
  - Static Variables
  - Local variables
  - Actions

Dieser Fensterbezug trägt jedoch keinerlei Typinformation. An MyOtherFunction() kann jedes beliebige Fensterobjekt als Parameter übergeben werden. Es gibt keine Möglichkeit, ein Objekt einer bestimmten Fensterklasse als Parameter zu definieren, so wie es im Fall der funktionalen Klassen möglich ist.

Alle drei Objekttypen besitzen Funktionen und lokale Variablen. Verbietet man die Mehrfachinstantiierung von Hauptfenster-Objekten, so entspricht der Objektbegriff von SQLWindows dem Objektbegriff des Dimensionskataloges.

4.3.2 Werte

Die in Abschnitt 4.2.3 vorgestellten SQLWindows-Datentypen haben alle Wertcharakter. Die Datentypen Sql Handle, File Handle und Window Handle stellen zwar Verweise auf Objekte (Datenbankverbindungen, Dateien bzw. Fenster) dar, sind an sich jedoch Werte. Mit diesen „eingebauten“ Datentypen können arithmetische Ausdrücke formuliert werden, deren Semantik der mathematischen entspricht.

Problematischer wird es bei der Modellierung von Fachwerten. Diese entsprechen – nach ihrer Definition – benutzerdefinierten Datentypen. In objektorientierten Programmiersprachen werden neue Datentypen mit Hilfe des Klassenkonzeptes eingeführt. Das Modellieren eines Fachwertes als Klasse hat den Vorteil, daß zusammen mit dem eigentlichen Wert auch die für diesen Wert zulässigen Operationen definiert werden können. Eine solche Modellierung ist auch in SQLWindows möglich. Nun soll aber ein Fachwert-Objekt – als ein Exemplar einer Fachwertklasse – sich wie ein Wert verhalten, und nicht wie ein Objekt, was es vom programmiersprachlichen Gesichtspunkt her ist. Die Aufgabe besteht also darin, dem Fachwert-Objekt Werteigenschaften zu geben. Fachwert-Objekte müssen z.B. immer der Wert- und nicht der Referenzsemantik gehorchen, d.h. es muß verhindert werden, daß auf ein Fachwert-Objekt verändernd zugegriffen werden kann. Diese Forderung ist z.B. dadurch zu erfüllen, daß für ein Exemplar einer Fachwertklasse nur ein einziges Mal (z.B. bei der Exemplarerzeugung) ein Wert gesetzt werden kann. Voraussetzung für eine solche Lösung ist ein Abstraktionsmechanismus, der den direkten Zugriff auf die Attribute eines Objektes verbietet. Nur so kann gewährleistet werden, daß der Wert eines Fachwert-Objektes nur über eine entsprechende Operation gesetzt werden kann, die ihrerseits dafür Sorge tragen muß, daß sie nur ein einziges Mal aufgerufen wird. Wie der folgende Abschnitt 4.3.3 zeigen wird, bietet SQLWindows keinen Abstraktionsmechanismus: Auf die Objektattribute

kann von jeder Stelle des Programmes aus zugegriffen werden. Es ist jedoch möglich, eine wertsetzende Funktion zu definieren, die nur ein einziges Mal aufgerufen werden kann – indem sie z.B. beim ersten Aufruf ein (als Objektattribut implementiertes) boolesches Flag setzt, das bei jedem Funktionsaufruf abgefragt wird und die Funktion gegebenenfalls beendet, ohne einen Wert zu setzen. Vorausgesetzt, dieses boolesche Flag wird nicht von außerhalb mutwillig verändert und der Wert des Objektes wird nur durch den Aufruf der entsprechenden Funktion gesetzt, so hat das Objekt einen Wertcharakter. Dieser Wertcharakter basiert jedoch auf der Disziplin des Benutzers des Objektes – für das Objekt einen Wertcharakter zu „erzwingen“, ist in SQLWindows unmöglich.

Die „eingebauten“ SQLWindows-Datentypen haben alle Wertcharakter. Das fehlende Abstraktionskonzept erlaubt es jedoch nicht, Fachwerte als Klassen zu realisieren, die einen Wertcharakter der Fachwert-Objekte garantieren.

4.3.3 Abstraktion

Die Attribute eines Objektes sollen dem Dimensionskatalog nach vor dem Zugriff von außerhalb geschützt werden. Die Dienste des Objektes sind durch eine Menge von Operationen definiert. Diese Beschränkung der Verwendung des Datentyps wird auch als *Kapselung* (*encapsulation*) bezeichnet. Bei überlegter Definition der Operationen kann die interne Implementierung des Objektes vollständig vor dem Benutzer des Datentyps verborgen werden – eine Eigenschaft, die als *Verbergen von Informationen* (*information hiding*) bezeichnet wird.

SQLWindows bietet keine Möglichkeit, um Kapselung zu erzwingen: Auf die Variablen und Funktionen eines jeden Objektes kann von jeder Stelle des Programms aus zugegriffen werden. Das Verbergen von Informationen kann in SQLWindows durch die Konvention erreicht werden, auf ein Objekt nur über Funktionen zuzugreifen. Nun macht aber i.d.R. nicht die Gesamtheit der Klassenfunktionen die Schnittstelle der Klasse aus, sondern nur eine Teilmenge der Funktionen. Eine gute Klassendokumentation sollte diejenigen Klassenfunktionen nennen, welche die Schnittstelle der Klasse bilden. Es ist aber letztendlich dem Programmierer überlassen, ob er tatsächlich nur diese Funktionen beim Zugriff auf ein Objekt dieser Klasse nutzt.

4.3.4 Typen und Klassen

Als Kern der objektorientierten Programmierung wird auch bei SQLWindows der Objektbegriff betrachtet. Motiviert wird die Verwendung von Objekten durch die Möglichkeit der Modularisierung und Abstraktion. Klassen werden auch in SQLWindows als Implementierung eines Datentypen verstanden. Ungewohnt ist jedoch die in SQLWindows übliche Art der Fensterobjekterzeugung und Fensterklassendefinition: Man erzeugt ein Objekt durch Auswahl einer der vordefinierten Fenster- und Kontrollelement-Typen. Hat man eines dieser kreierten grafischen Objekte als wiederverwendbar ausgemacht, so kommt die Objektorientierung ins Spiel: Man kann ein solches Objekt mit Hilfe des Class Editor-Werkzeuges von SQLWindows zu einer Klasse machen, von der dann wieder weitere Objekte abgeleitet werden können. Diese Art der Programmierung bezeichnet Wegner als *prototypische Programmierung* (vgl. auch Abschnitt 2.2.5): Unter einem *Prototyp* versteht Wegner ein Objekt, das sowohl Exemplar als auch Schablone ist. Anstatt

die allgemeinen Eigenschaften eines Objektes in einer Klasse zu abstrahieren, wird das Objekt zum Prototypen deklariert. „Ähnliche“ Objekte werden nur durch ihre Unterschiede gegenüber dem Prototyp repräsentiert.

Allerdings kann man in SQLWindows auch eigene Klassen „von Hand“ definieren. Dabei unterscheidet SQLWindows grundsätzlich zwischen *Fensterklassen* (*window classes*) und *funktionalen Klassen* (*functional classes*).

4.3.4.1 Fensterklassen

Fensterklassen basieren auf den Standard-Fenstertypen von SQLWindows. Diese lassen sich unterteilen in *Hauptfenster* (*top-level objects*) wie Frame Windows oder Table Windows, *Kindfenster* (*child objects*) wie Data Fields oder Push Buttons, und *Menüs* (*menus*). Für jeden dieser Typen gibt es eine vordefinierte Fensterklasse. Die SQLWindows-Beschreibung einer Fensterklasse besteht aus den folgenden Abschnitten:

- Form Window Class: *Klassenname*
  - Description:
  - Derived From
  - Class Variables
  - Instance Variables
  - Functions
  - Message Actions

Der Abschnitt Derived From wird im Abschnitt 4.3.6 bei der Untersuchung der Vererbung behandelt. *Klassenvariablen* (Abschnitt Class Variables) sind Variablen, die nur ein einziges Mal für eine Klasse angelegt werden und von allen Instanzen dieser Klasse gemeinsam genutzt werden (siehe auch Abschnitt 4.3.6.3). Demgegenüber werden *Instanzenvariablen* (im Abschnitt Instance Variables) für jede Instanz der Klasse angelegt. Im Abschnitt Message Actions können Aktionen als Antwort auf Nachrichten festgelegt werden.

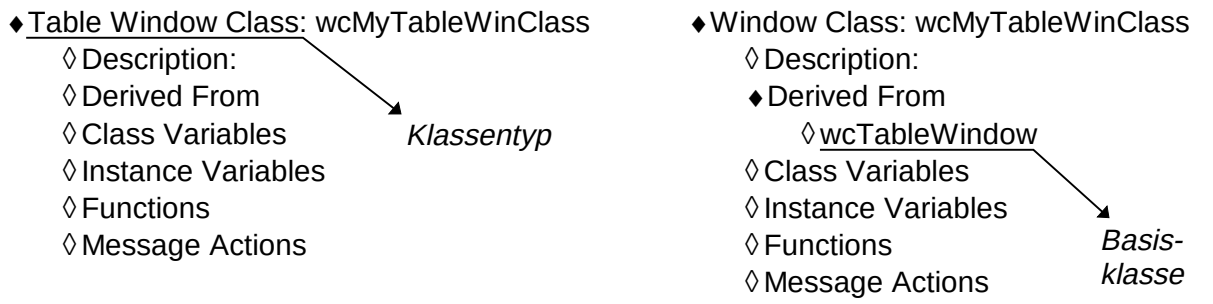
Von Fensterklassen können weitere Fensterklassen abgeleitet werden, die denselben SQLWindows-Fenstertyp wie ihre Basisklasse haben müssen.

Instanzen von Fensterklassen heißen *Standard-Fensterobjekte* (*standard window objects*), wenn sie direkt von einem der SQLWindows-Fenstertypen abgeleitet sind, oder ansonsten *benutzerdefinierte Fensterobjekte* (*user-defined window objects*). SQLWindows bietet mit der Basisklasse General Window Class die Möglichkeit, eine abstrakte Fensterklasse ohne Berücksichtigung eines bestimmten Fenstertyps zu deklarieren. Die von einer solchen Klasse abgeleiteten Klassen müssen jedoch einem der SQLWindows-Fenstertypen angehören.

Identifiziert werden Fensterklassen-Objekte über einen eindeutigen *Fensterbezug* (*window handle*). Es ist zwar möglich, ein (Klassen-)Objekt über seinen Namen anzusprechen; werden aber von einem Fenster (und somit von einem (Klassen-)Objekt) mehrere Instanzen erzeugt, so ist der Objektname nicht mehr eindeutig. An dieser Stelle wird deutlich, daß in SQLWindows die Begriffe Klasse und Objekt nicht sauber voneinander getrennt sind – wenn nämlich von einem Objekt mehrere Instanzen erzeugt werden können. Diese Problematik wurde bereits in Abschnitt 4.3.1.2 erörtert.

Verwirrend ist die Tatsache, daß SQLWindows zwischen verschiedenen Klassentypen unterscheidet. So erbt z.B. eine benutzerdefinierte Fensterklasse, die auf einem Table Window basieren

soll, nicht etwa von der Klasse Table Window, sondern *ist* eine Table Window Class. Geerbt werden kann nur von benutzerdefinierten Klassen:



SQLWindows **vs.** **OO**

Abbildung 4.15: Vererbung in SQLWindows – Klassentyp und Basisklasse

Die linke Klassendeklaration ist SQLWindows-typisch: Table Window Class wird als Klassentyp angegeben. Wünschenswert wäre aber eine Klassendeklaration der Art, wie sie auf der rechten Seite dargestellt ist: Die Fensterklasse erbt von einer Klasse TableWindow, die alle Eigenschaften eines Tabellenfensters kapselt.

Diese Unterscheidung in Klassentyp und Basisklasse weicht das Klassenkonzept auf. Die vordefinierten Klassentypen hätten ebenso gut als Basisklassen definiert werden können, von denen benutzerdefinierte Klassen erben. Dieser Mißstand liegt vermutlich in der Entwicklungsgeschichte von SQLWindows – vom rein prozeduralen zum objektorientierten Entwicklungssystem – begründet. Die Standard-Fenstertypen sind eine „Altlast“ aus den prozeduralen Tagen. Als SQLWindows um objektorientierte Elemente erweitert wurde, sollte der bisherige Mechanismus der Objekterzeugung (man erzeugte ein Objekt durch Auswahl eines der Fenstertypen) beibehalten werden. Gleichzeitig mußte man aber auch Klassen definieren können. Der prototypische Charakter von SQLWindows verlangte, daß man aus einem Objekt eine Klasse generieren lassen kann. Da die Objekte auf einem der Fenstertypen basierten, schuf man die Klassentypen als Pendant auf Klassenebene.

Genährt werden solche Sichten der Objektorientierung durch die Literatur zur Microsoft Windows-Programmierung. Hier wird vielfach behauptet, daß die Windows-Programmierung selbst unter Verwendung der Programmiersprache C in einer gewissen Art und Weise objektorientiert sei, da Windows auf dem Begriff des Fensters als Objekt beruht (vgl. z.B. [Pet90], [Buc92]). In der Tat basiert jedes Windows-Fensterobjekt auf einer Fensterklasse, die das Verhalten eines Fensters beschreibt. Es ist möglich, neue Windows-Fensterklassen zu definieren. Eine Fensterklasse im Sinne von Windows ist jedoch noch lange keine Klasse im objektorientierten Sinne, sondern lediglich eine Sammlung von Attributen, die das Verhalten und das Aussehen eines Fensters festlegen (Icon-Symbol, Mauszeigerbild, Farbmuster des Fensterhintergrundes etc.). Eine Windows-Fensterklasse kann keine Funktionen enthalten, da sie als einfache C-Datenstruktur (struct) definiert ist. Das unterscheidet sie von Klassen im objektorientierten Sinne, für die eine Windows-Fensterklasse jedoch als Basis dienen kann, wie viele GUI-Klassenbibliotheken für Windows zeigen (z.B. Microsoft Foundation Classes, StarView oder zApp): Alle diese Klassenbibliotheken definieren auf Grundlage der Standard-Fensterklassen von Windows (und evtl. eige-

ner Fensterklassen) entsprechende C++ - Klassen, die dem Benutzer der Klassenbibliothek als Basis-Fensterklassen zur Verfügung stehen. Diese Fensterklassen kapseln hauptsächlich das Verhalten von bzw. die Interaktion mit dem entsprechenden Fenstertyp. So definiert z.B. eine Klasse für Listenfelder (List Boxes) Methoden zum Einfügen bzw. Entfernen von Elementen in die bzw. aus der Liste. Die Entwickler von SQLWindows gingen einen anderen Weg, indem sie die Windows-Fensterklassen nicht als Basis für eigene Fensterklassen nutzten, sondern diese zu Klassentypen machten und sich somit vom objektorientierten Klassenbegriff lossagten.

4.3.4.2 Funktionale Klassen

Die funktionalen Klassen entsprechen am ehesten dem gängigen Klassenbegriff. Eine funktionale Klasse basiert nicht auf einem visuellen Objekt und somit auch nicht auf einem bestimmten Klassentyp. Funktionale Klassen kapseln Funktionen und Variablen.

- Functional Class: *Klassenname*
  - Description:
  - Derived From
  - Class Variables
  - Instance Variables
  - Functions

Von funktionalen Klassen können neben weiteren funktionalen Klassen auch Fensterklassen abgeleitet werden (vgl. Abschnitt 4.3.6.1). Instanzen von funktionalen Klassen heißen benutzerdefinierte Variablen (vgl. Abschnitt 4.3.1.2).

Nimmt man die Definition des abstrakten Datentyps sehr genau, dann entsprechen die SQLWindows-Klassen nicht dem Klassenbegriff des Dimensionskataloges, allein schon aus dem Grund, daß auf alle Attribute zugegriffen werden kann. Außerdem wären dann die Klassentypen von SQLWindows Typen von (Implementierungen von abstrakten Daten-)Typen, was das Typkonzept bräche. Betrachtet man eine Klasse jedoch unter dem Gesichtspunkt einer Schablone für Objekte, die Dienste und einen internen Zustand durch lokale Variablen definiert, dann findet man diese Betrachtungsweise auch in den SQLWindows-Klassen wieder. Der Klassenbegriff von SQLWindows soll deshalb als eingeschränkt bezeichnet werden.

4.3.5 Referenzen

Wie bereits in Abschnitt 4.2.7 beschrieben, können die Basis-Datentypen von SQLWindows per Referenz als Parameter einer Funktion übergeben werden (call by reference). Dazu werden die entsprechenden Formalparameter mit dem Schlüsselwort Receive gekennzeichnet. Das Schlüsselwort Receive darf nicht für lokale Variablen einer Funktion bzw. Klassen- und Instanzvariablen einer Klasse verwendet werden.

Benutzerdefinierte Variablen als Parameter einer Funktion werden immer per Referenz übergeben. Mit dieser Referenz kann man dann innerhalb der Funktion arbeiten, aber man kann sie nicht als Attribut des Objektes speichern.

Objekte, die Instanzen von Fensterklassen sind, können nicht als Parameter übergeben werden. Man kann sich zwar mit dem Fensterbezug behelfen, wie in Abschnitt 4.3.1.2 ausgeführt, verliert aber damit die Möglichkeit der Typprüfung.

Die Übergabe von Objektreferenzen als Parameter und das Speichern von Objektreferenzen als Attribute sind eine Grundvoraussetzung für die objektorientierte Programmierung. SQLWindows unterstützt lediglich die Übergabe von Referenzen auf Basis-Datentypen und benutzerdefinierte Variablen, nicht aber das Speichern von Objektreferenzen als Attribute. Mit dem in Kapitel 6 beschriebenen SQLWindows Object System (SOS) wurde ein eingeschränkter Referenzmechanismus für SQLWindows geschaffen.

4.3.6 Einfache Vererbung

Mit einfacher Vererbung kann eine Klasse von einer anderen Klasse, genannt *Basisklasse*, abgeleitet werden:

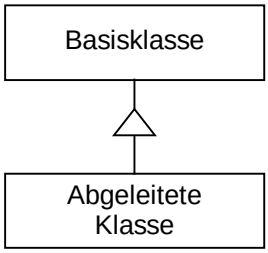


Abbildung 4.16: Einfache Vererbung

Auch mehrstufige Vererbung ist zulässig, d.h. von einer abgeleiteten Klasse kann wiederum eine Klasse abgeleitet werden, usw.

Funktionale Klassen und Fensterklassen haben unterschiedliche Vererbungsregeln.

4.3.6.1 Vererbungsregeln funktionaler Klassen

Eine funktionale Klasse kann sein:

- Basisklasse einer anderen funktionalen Klasse.
- Basisklasse einer Fensterklasse.
- Abgeleitete Klasse einer anderen funktionalen Klasse, aber *nicht* von einer Fensterklasse.

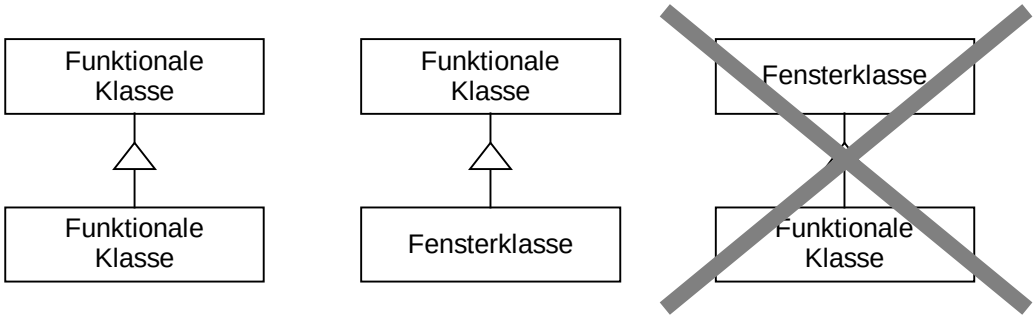


Abbildung 4.17: Vererbungsregeln funktionaler Klassen

4.3.6.2 Vererbungsregeln von Fensterklassen

Eine Fensterklasse kann sein:

- Basisklasse einer anderen Fensterklasse desselben Typs. Eine Data Field Class kann z.B. nur Basisklasse einer anderen Data Field Class sein.
- Abgeleitete Klasse einer anderen Fensterklasse desselben Typs oder einer funktionalen Klasse.

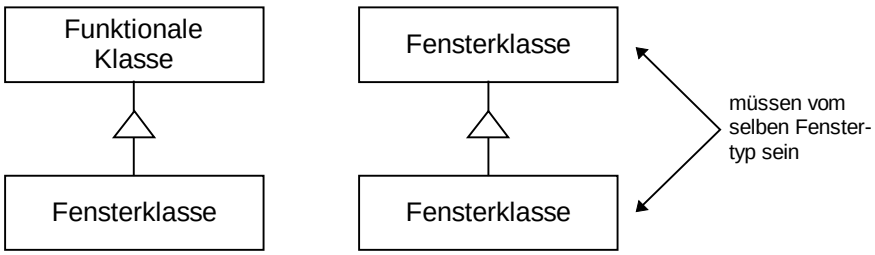


Abbildung 4.18: Vererbungsregeln von Fensterklassen

Alle Variablen (sowohl interne Variablen als auch Klassenvariablen) und Funktionen werden vererbt. Um die Funktion einer Oberklasse zu rufen, wird der Funktionsname mit dem Klassennamen qualifiziert:

Call *Klassenname.Funktionsname*()

Enthält eine Fensterklasse Kindfenster, so werden diese mitvererbt. Auch die Message Actions werden vererbt. Beim Eintreffen einer entsprechenden Nachricht wird nur die Message Action der abgeleiteten Klasse aufgerufen. Mit Hilfe der Funktionen SalSendClassMessage() und SalSendClassMessageNamed() können Nachrichten an die Message Actions der Elternklassen weitergereicht werden.

Die Vererbungsbeziehungen werden im Abschnitt Derived From der Klassendefinition angegeben. Als Beispiel soll ein Ausschnitt aus dem Programm `INHERIT.APP` dienen:

- Functional Class: fcA
  - Description:
  - Derived From
  - Class Variables
  - Instance Variables
  - Functions
- Functional Class: fcB
  - Description:
  - Derived From
    - Class: fcA
  - Class Variables
  - Instance Variables
  - Functions

Die funktionale Klasse fcB erbt von der funktionalen Klasse fcA.

4.3.6.3 Vererbung von Klassenvariablen

Sehr ungewöhnlich ist die Umsetzung des Konzepts der Klassenvariablen in SQLWindows. Klassenvariablen werden nur ein einziges Mal je Klasse angelegt und von allen Objekten dieser Klasse gemeinsam genutzt. Leitet man aber von einer Klasse, die eine Klassenvariable besitzt, eine weitere Klasse ab, und instantiiert sowohl ein Objekt von der Basisklasse als auch ein Objekt von der abgeleiteten Klasse, so benutzen beide Objekte dieselbe Klassenvariable. Ein Beispiel (`CLASSVAR.APP`) soll diesen Umstand verdeutlichen:

- Functional Class: fcBase
  - Description:
  - Derived From
  - Class Variables
    - String: m_strClassName
  - Instance Variables
  - Functions
    - Function: Init
      - Description:
      - Returns
      - Parameters
      - Static Variables
      - Local variables
      - Actions
        - Call SetClassName('fcBase')
    - Function: SetClassName
      - Description: Stores the class name in the class variable m_strClassName
      - Returns
      - Parameters
        - String: strClassName
      - Static Variables
      - Local variables
      - Actions
        - Set m_strClassName = strClassName

- ♦ Function: GetClassName
  - ◊ Description: Returns the class name stored in the class variable m_strClassName
  - ♦ Returns
    - ◊ String:
  - ◊ Parameters
  - ◊ Static Variables
  - ◊ Local variables
  - ♦ Actions
    - ◊ Return m_strClassName
- ♦ Functional Class: fcA
  - ◊ Description:
  - ♦ Derived From
    - ◊ Class: fcBase
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ♦ Function: Init
      - ◊ Description:
      - ◊ Returns
      - ◊ Parameters
      - ◊ Static Variables
      - ◊ Local variables
      - ♦ Actions
        - ◊ Call SetClassName('fcA')
- ♦ Functional Class: fcB
  - ◊ Description:
  - ♦ Derived From
    - ◊ Class: fcA
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ♦ Function: Init
      - ◊ Description:
      - ◊ Returns
      - ◊ Parameters
      - ◊ Static Variables
      - ◊ Local variables
      - ♦ Actions
        - ◊ Call SetClassName('fcB')

Die Basisklasse fcBase speichert den Klassennamen in einer Klassenvariablen m_strClassName. Dieser kann mittels SetClassName() gesetzt werden. Die Klassen fcA (abgeleitet von fcBase) und fcB (abgeleitet von fcA) tun dies in ihrer Init()-Funktion.

Nun erzeugt man je eine Instanz der Klassen fcA und fcB und initialisiert diese, wobei die Instanz von fcA zweimal initialisiert wird:

◊ fcA: udvA
◊ fcB: udvB
...
◊ Call udvA.Init()
◊ Call udvB.Init()
◊ Call udvA.Init()

Durch Aufruf der Init()-Funktion werden die Klassennamen gesetzt. Die folgende Tabelle gibt an, welchen Wert die Funktion GetClassName() der Klassen fcA und fcB jeweils nach dem Aufruf von Init() zurückliefert:

Nach dem Aufruf von...	...liefert udvA.GetClassName()	...liefert udvB.GetClassName()
Call udvA.Init()	fcA	fcA
Call udvB.Init()	fcB	fcB
Call udvA.Init()	fcA	fcA

Tabelle 4.2: Werte der Klassenvariablen m_strClassName

Das Setzen des Klassennamens in udvB.Init() überschreibt den in udvA.Init() gesetzten Klassennamen, und umgekehrt (beim zweiten Aufruf von udvA.Init()). Beide Klassen benutzen also dieselbe Klassenvariable, obwohl sie verschiedene Klassen sind – die allerdings von derselben Klasse erben. Dieser Eigenart muß man sich bewußt sein, wenn man das Klassenvariablen-Konzept nutzen möchte.

4.3.6.4 Objektableitung

Das wohl sonderbarste Konzept in SQLWindows (neben der Mehrfach-Instantiierung von Objekten) wird von Niemeier als *Objektableitung* (*object derivation*) bezeichnet (vgl. [Nie]): Ein Objekt einer Fensterklasse kann zur Entwicklungszeit direkt modifiziert werden, ohne eine neue Klasse ableiten zu müssen. Gegeben sei z.B. eine Fensterklasse wcMain (vgl. Beispielprogramm `OBJDERIV.APP`). Diese repräsentiert ein Fenster mit einigen Kontrollelementen. Wird nun eine Instanz frmMain dieser Klasse angelegt, so ist es möglich, dieser Instanz weitere Kontrollelemente hinzuzufügen:

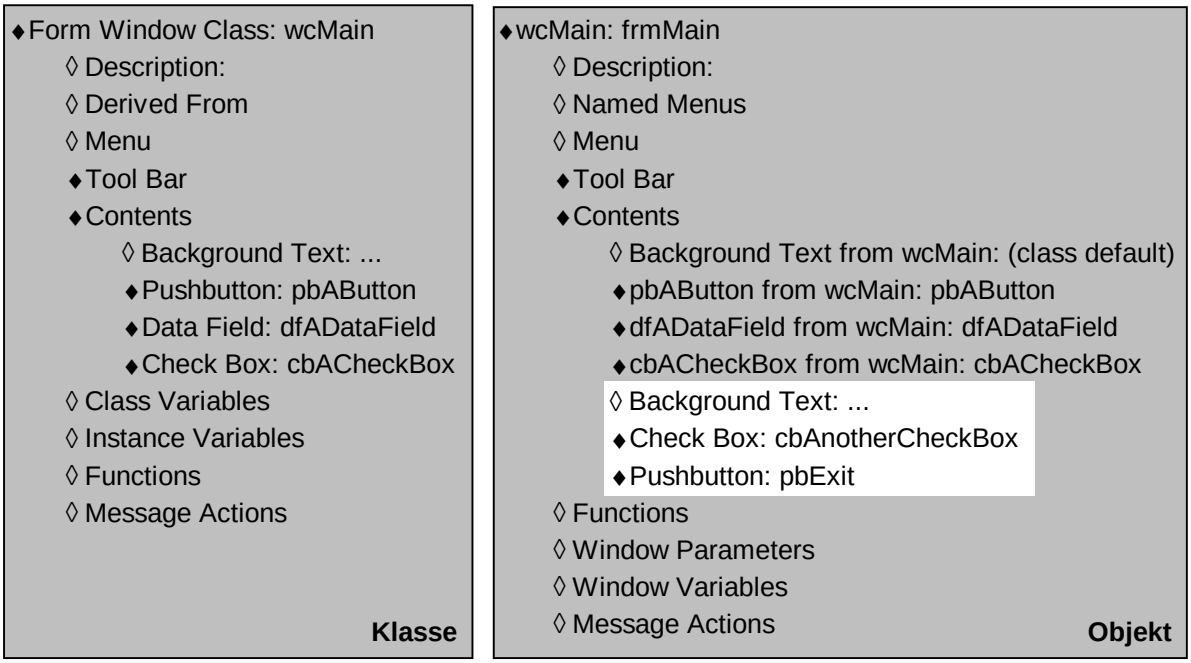


Abbildung 4.19: Objektableitung

frmMain hat immer noch wcMain als Typ, bietet aber eine gegenüber der Klasse erweiterte Funktionalität. Es ist jedoch nicht möglich, Kontrollelemente , die in wcMain definiert sind, aus frmMain zu entfernen.

Das Konzept der Objektableitung macht SQLWindows zu einer Art prototypischen Sprache. Der Unterschied zur Definition von prototypischen Sprachen nach Wegner (vgl. Abschnitt 2.2.5) besteht darin, daß hier eine Klasse und nicht ein Objekt den Prototypen bildet, von dem weitere Objekte abgeleitet werden können, die dann nur durch ihre Unterschiede gegenüber dem Prototypen repräsentiert werden.

Die Implementierung der einfachen Vererbung in SQLWindows entspricht der Definition im Dimensionskatalog. Die Tatsache, daß in Fensterklassen ein Abschnitt Message Actions für die Bearbeitung von Nachrichten für die Klasse existiert, verlangt nach einem Mechanismus, um die Nachricht an die Elternklassen weiterzuleiten. Ein solcher ist in SQLWindows mit der Funktionen SalSendClassMessage() und SalSendClassMessageNamed() vorhanden. Das Konzept der Objektableitung weicht die Vererbungs- und Instantiierungsregeln dahingehend auf, daß es erlaubt, instantiierte Fensterobjekte auf der Objektebene zu erweitern. Die Objektableitung macht SQLWindows zu einer prototypischen Sprache im Sinne Wegners.

4.3.7 Polymorphismus und dynamisches Binden

SQLWindows unterstützt das späte (dynamische) Binden von (Klassen-)Funktionen und ermöglicht somit die Nutzung von Polymorphismus. Spät gebundene Funktionen habe zwei Punkte '..' als Präfix. Ein Beispiel, dem Programm `POLYCALL.APP` entnommen:

- ♦ Pushbutton Class: wcBaseButton
  - ◊ Description:
  - ◊ Derived From
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ♦ Function: ClickAction
      - ◊ Description:
      - ◊ Returns
      - ◊ Parameters
      - ◊ Static Variables
      - ◊ Local variables
      - ♦ Actions
        - ◊ Call SalMessageBox('wcBaseButton::ClickAction()', 'SAM_Click Message Action', MB_Ok)
  - ♦ Message Actions
    - ♦ On SAM_Click
      - ◊ Call ..ClickAction()
- ♦ Pushbutton Class: wcDerivedButton
  - ◊ Description:
  - ♦ Derived From
    - ♦ Class: wcBaseButton
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ♦ Function: ClickAction
      - ◊ Description:
      - ◊ Returns
      - ◊ Parameters
      - ◊ Static Variables
      - ◊ Local variables
      - ♦ Actions
        - ◊ Call SalMessageBox('wcDerivedButton::ClickAction()', 'SAM_Click Message Action', MB_Ok)
  - ◊ Message Actions

Die Klasse wcBaseButton reagiert auf die Nachricht SAM_Click mit einem Aufruf der Funktion ClickAction(). Die Klasse wcDerivedButton erbt von wcBaseButton. Sie überschreibt die Funktion ClickAction(), reagiert jedoch nicht auf die Nachricht SAM_Click. Wird nun eine Schaltfläche vom Typ wcDerivedButton angeklickt, so wird ClickAction() aufgerufen, wie in den Message Actions der Basisklasse wcBaseButton festgelegt. Da ClickAction() dort polymorph gerufen wird (Call

..ClickAction()), wird in diesem Fall die in wcDerivedButton überschriebene Version von ClickAction() aufgerufen.

Im obigen Beispiel findet der polymorphe Aufruf innerhalb einer Klasse statt. Es ist in SQLWindows jedoch nicht möglich, eine Funktion einer Klasse polymorph von außerhalb des Objektes zu rufen. Zur Veranschaulichung dient wieder ein Ausschnitt aus dem Beispielprogramm `POLYCALL.APP`:

```
♦Functional Class: fcBaseClass
    ◊ Description: A base class
    ◊ Derived From
    ◊ Class Variables
    ◊ Instance Variables
    ♦Functions
        ♦Function: GetName
            ◊ Description: Returns the class name
            ♦Returns
                ◊ String:
            ◊ Parameters
            ◊ Static Variables
            ◊ Local variables
            ♦Actions
                ◊ Return "fcBaseClass"
        ♦Function: _GetName
            ◊ Description:
            ♦Returns
                ◊ String:
            ◊ Parameters
            ◊ Static Variables
            ◊ Local variables
            ♦Actions
                ◊ Return ..GetName()
♦Functional Class: fcDerivedClass
    ◊ Description: A class derived from fcBaseClass
    ♦Derived From
        ◊ Class: fcBaseClass
    ◊ Class Variables
    ◊ Instance Variables
    ♦Functions

        ♦Function: GetName
            ◊ Description: Returns the class name
            ♦Returns
                ◊ String:
            ◊ Parameters
            ◊ Static Variables
            ◊ Local variables
            ♦Actions
                ◊ Return "fcDerivedClass"
```

Die folgende Klasse fcAnotherClass definiert eine Funktion GetName(), die eine Referenz auf ein Objekt vom Typ fcBaseClass übergeben bekommt. Für diese Funktion werden drei Möglichkeiten für den Rückgabewert angegeben:

- Functional Class: fcAnotherClass
  - Description:
  - Derived From
  - Class Variables
  - Instance Variables
  - Functions
    - Function: GetName
      - Description: Returns the class name of an object passed as parameter
      - Returns
      - Parameters
        - fcBaseClass: udvObject
      - Static Variables
      - Local variables
      - Actions
        - Return udvObject.GetName() ①
        - Return udvObject..GetName() ②
        - Return udvObject._GetName() ③

Wird nun ein Objekt vom Typ fcDerivedClass an GetName() übergeben, so führen die drei angegebenen Möglichkeiten für den Rückgabewert zu unterschiedlichen Ergebnissen:

Variante ① liefert wie erwartet den Namen der Basisklasse. Variante ② führt zu einer Fehlermeldung des SQLWindows-Compilers. Als Abhilfe kann für alle polymorph zu rufenden Funktionen eine zusätzliche Funktion definiert werden, die denselben Namen – mit einem Unterstrich '_' als Suffix[12] – trägt, und welche die eigentliche Funktion polymorph aufruft (vgl. Variante ③).

David Burke beschreibt in [Hol95] eine zweite Form des Polymorphismus in SQLWindows: Mit Hilfe von Nachrichten. Durch das Versenden einer Nachricht kann beim empfangenden Objekt eine bestimmte Reaktion ausgelöst werden, ohne daß der Typ des Empfängers bekannt sein muß. Viele der in SQLWindows definierten Nachrichten der Form SAM_* werden polymorph benutzt. So zeigt z.B. die Nachricht SAM_Click an, daß das empfangende Objekt angeklickt wurde – unabhängig von dessen Typ. Es gibt keine Spezialisierungen von SAM_Click wie SAM_ButtonClick, SAM_ColumnClick, SAM_TableClick oder SAM_ListClick. Da Nachrichten nur von Fensterklassen empfangen werden können, nicht aber von funktionalen Klassen, ist dieses Konzept nicht umfassend genug und soll daher hier nicht weiter betrachtet werden.

Das Polymorphismus-Konzept entfaltet seine Möglichkeiten erst mit dem Referenzkonzept in Verbindung mit der Vererbung: Wenn nämlich Verweise auf Objekte definiert werden können, bei denen der statische und der dynamische Typ unterschiedlich sind, dann kann ein Objekt einer abgeleiteten Klasse unter der Schnittstelle seiner Oberklasse bekanntgemacht werden, wobei der polymorphe Aufruf einer Funktion der Oberklassen-Schnittstelle in einem Aufruf der in der abgeleiteten Klasse überschriebenen Funktion resultieren kann. Da das Referenzkonzept von

[12] Diese Konvention wurde von den Klassen des E.I.S. Application Builder übernommen (vgl. [EIS96]).

SQLWindows sehr beschränkt ist, kann auch nur ein Teil der Möglichkeiten genutzt werden, die das Polymorphismus-Konzept bietet.

SQLWindows unterstützt spätes (dynamisches) Binden von Funktionen und stellt somit einen Mechanismus zur Verfügung, mit dem Polymorphismus realisiert werden kann. Leider wurde dieses Konzept nur unzureichend implementiert: So können polymorphe Aufrufe nicht von außerhalb eines Objektes erfolgen, d.h. ein Funktionsaufruf der Art Call *Objekt..Funktion*() ist nicht erlaubt. Polymorphismus kann in SQLWindows lediglich dazu genutzt werden, um in einer Klasse polymorph eine Funktion aufzurufen, die in einer abgeleiteten Klasse überschrieben wurde.

4.3.8 Mehrfachvererbung

Mit Mehrfachvererbung kann eine Klasse mehr als eine direkte Basisklasse haben:

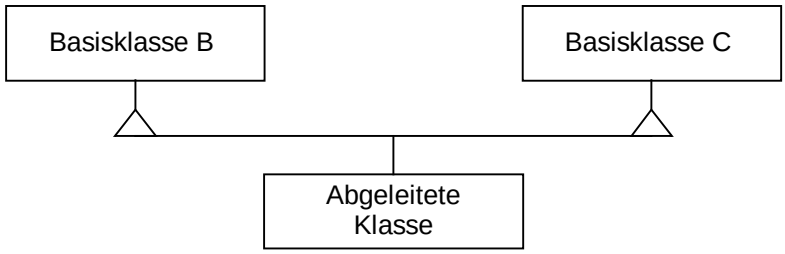


Abbildung 4.20: Mehrfachvererbung

Was aber passiert, wenn die beiden Basisklassen B und C eine Klasse A als Elternteil haben? Diesen Fall bezeichnet Meyer in [Mey90] als *wiederholtes Erben*. Es gibt nun zwei Möglichkeiten:

Man betrachtet die beiden „Instanzen“ der Basisklasse A getrennt voneinander:

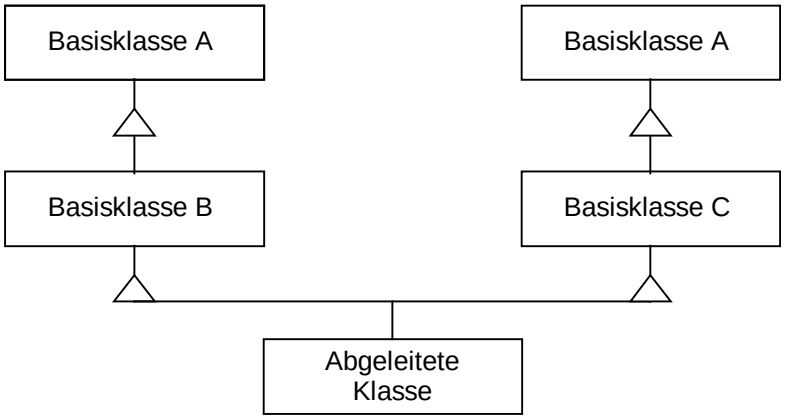


Abbildung 4.21: Wiederholtes Erben – Variante 1

Oder man kommt zu einer Konstruktion, die *diamantenförmige* oder *rautenförmige Vererbung* genannt wird:

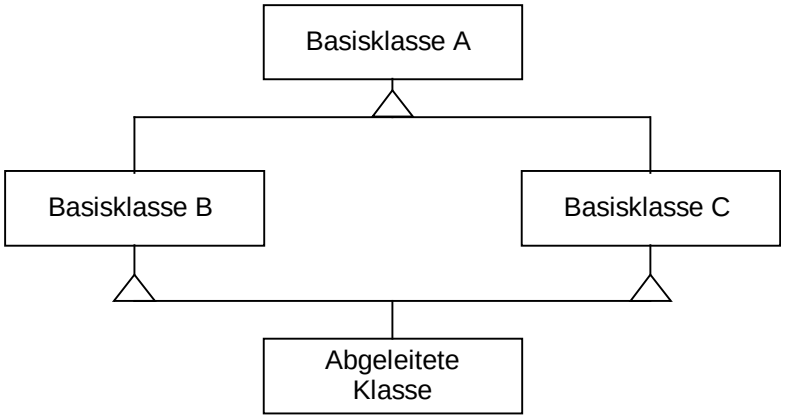


Abbildung 4.22: Wiederholtes Erben – Variante 2

An einem Beispielprogramm (`INHERIT.APP`) soll untersucht werden, welche der beiden Möglichkeiten für wiederholtes Erben von SQLWindows unterstützt wird.

- ♦ Functional Class: fcA
  - ◊ Description:
  - ◊ Derived From
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ♦ Function: GetClassAName
      - ◊ Description:
      - ♦ Returns
        - ◊ String:
      - ◊ Parameters
      - ◊ Static Variables
      - ◊ Local variables
      - ♦ Actions
        - ◊ Return 'Class A'
- ♦ Functional Class: fcB
  - ◊ Description:
  - ♦ Derived From
    - ◊ Class: fcA
  - ◊ Class Variables
  - ◊ Instance Variables
  - ◊ Functions
- ♦ Functional Class: fcC
  - ◊ Description:
  - ♦ Derived From
    - ◊ Class: fcA

◊ Class Variables
◊ Instance Variables
◊ Functions
♦ Form Window Class: wcD
◊ Description:
♦ Derived From
◊ Class: fcB
◊ Class: fcC
◊ Menu
♦ Tool Bar
◊ Contents
◊ Class Variables
♦ Instance Variables
◊ String: sClassAString
◊ Functions
♦ Message Actions
♦ On SAM_Create
◊ Set sClassAString = GetClassAName()

Die Vererbungsbeziehungen dieser Klassen sollen zunächst noch einmal grafisch veranschaulicht werden. Die strichlierte Lösung stellt die diamantenförmige Vererbung dar:

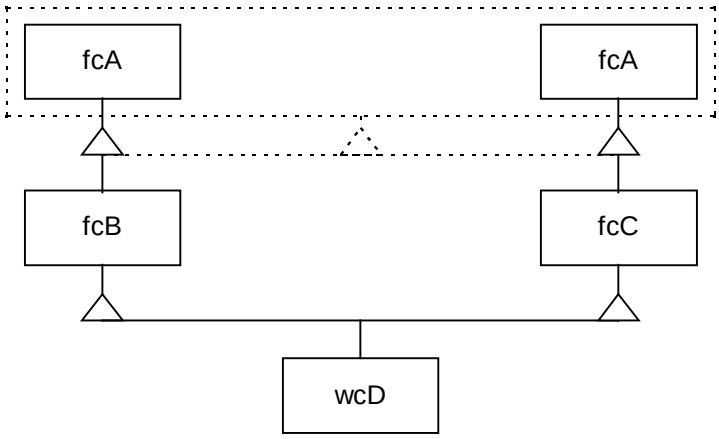


Abbildung 4.23: Klassendiagramm des Beispielprogramms `INHERIT.APP`

Beim Versuch, ein Programm mit einer Instanz der Klasse wcD auszuführen, liefert der SQLWindows-Compiler die Fehlermeldung, daß die Funktion GetClassAName() redundant von Klasse fcA über die Klasse fcB und fcC geerbt wird. Ruft man GetClassAName() klassenqualifiziert auf (z.B. Set sClassName = fcB.GetClassAName()), so läßt sich das Programm anstandslos übersetzen. SQLWindows benutzt also nicht standardmäßig die diamantenförmige Vererbung, bietet aber auch kein sprachliches Konstrukt, mit dem diese Form der Mehrfachvererbung festgelegt werden kann (vergleichbar dem Schlüsselwort virtual zur Definition virtueller Basisklassen in C++). Damit sind die Möglichkeiten der Mehrfachvererbung in SQLWindows eingeschränkt.

Die Implementierung der Mehrfachvererbung in SQLWindows entspricht der Definition im Dimensionskatalog. Besitzt eine Klasse mehrere Basisklassen, so dürfen diese nicht von einer gemeinsamen Klasse erben, wenn auch die abgeleitete Klasse Funktionen dieser gemeinsamen Basisklasse nutzen soll. SQLWindows unterstützt nämlich keine diamantenförmigen Vererbungsstrukturen. Diese Tatsache beschränkt die Möglichkeiten der Mehrfachvererbung in SQLWindows.

4.3.9 Nebenläufigkeit

SQLWindows ist eng mit dem Betriebssystem Microsoft Windows 3.x verbunden, einem nichtpräemptiven Multitasking-Betriebssystem, das die Unterteilung einer Applikation in mehrere *Prozesse* (*threads*) nicht unterstützt. Diese Eigenschaft, *Multithreading* genannt, verbunden mit einem preemptiven Multitasking, bieten erst die Betriebssysteme Windows NT und Windows 95. Da das Betriebssystem kein Multithreading unterstützt, kann es auch von SQLWindows nicht unterstützt werden. SQLWindows stellt jedoch Funktionen bereit, um den Kontrollfluß an ein anderes Objekt zu übergeben, und das sowohl innerhalb einer Applikation als auch zwischen Applikationen.

4.3.9.1 Kontrollflußweitergabe innerhalb einer Applikation

Solange sich eine SQLWindows-Applikation in einem Actions-Abschnitt eines Objektes befindet, kann kein anderes Objekt die Systemkontrolle übernehmen: Alle Anweisungen des Abschnitts werden nacheinander abgearbeitet (synchrone Verarbeitung). Mit Hilfe der Funktion SalYield-StartMessages() kann eine asynchrone Verarbeitung erreicht werden. Das Beispielprogramm `YIELD.APP` definiert in seinem Hauptfenster zwei Schaltflächen pbStart und pbStop, die asynchron arbeiten:

```
♦Data Field: dfCount     ! The type of this data field is Number.
    ◊ Message Actions
♦Pushbutton: pbStart
    ♦Message Actions
        ♦On SAM_Click
            ◊ Call SalYieldStartMessages(pbStop)
            ◊ Call SalDisableWindow(pbStart)
            ◊ Call SalEnableWindow(pbStop)
            ◊ Set bStopped = FALSE
            ◊ Set dfCount = 0
            ♦While dfCount < 1000
                ◊ Set dfCount = dfCount + 1
                ♦If bStopped = TRUE
                    ◊ Break
♦Pushbutton: pbStop
    ♦Message Actions
        ♦On SAM_Create
            ◊ Call SalDisableWindow(pbStop)
        ♦On SAM_Click
            ◊ Set bStopped = TRUE
            ◊ Call SalYieldStopMessages()
```

◊ Call SalDisableWindow(pbStop)
◊ Call SalEnableWindow(pbStart)

Nach einem Klick auf die Schaltfläche pbStart wird die Nachricht SAM_Click an dieses Schaltflächenobjekt gesandt. pbStart ruft daraufhin SalYieldStartMessages() für die Schaltfläche pbStop auf. Damit kann pbStop der Schaltfläche pbStart die Systemkontrolle entziehen. Anschließend tritt das Programm in eine Schleife ein. Wird jetzt die Schaltfläche pbStop angeklickt, dann wird die Schleifenausführung unterbrochen, und die Kontrolle geht an den Programmabschnitt über, der bei der Schaltfläche pbStop für die Nachricht SAM_Click zuständig ist. Hier wird die boolesche Variable bStopped auf den Wert TRUE gesetzt. Nach Abarbeitung der restlichen Anweisungen wird mit der Ausführung der Schleife für pbStart fortgefahren. Da bStopped jetzt den Wert TRUE hat, wird die Schleife abgebrochen.

4.3.9.2 Kontrollflußweitergabe zwischen Applikationen

Da Microsoft Windows 3.x ein nicht-preemptives Multitasking-Betriebssystem ist, müssen alle laufenden Applikationen von Zeit zu Zeit die Kontrolle an Windows zurückgeben, damit dieses wiederum die Kontrolle an eine der anderen Applikationen weitergeben kann. Im Windows API gibt es Funktionen wie PeekMessage(), die genau dies sicherstellen. SQLWindows übernimmt diese Aufgabe automatisch, ohne daß sich der SQLWindows-Programmierer darum kümmern müßte. Er kann dieses Verhalten aber verhindern. Durch Aufruf von SalYieldEnable(FALSE) wird eben diese automatische Abgabe der Kontrolle unterbunden. Als Beispiel dient ein weiterer Ausschnitt aus dem Programm `YIELD.APP`:

```
♦Pushbutton: pbNoYield
    ♦Message Actions
        ♦On SAM_Click
            ◊ Call SalDisableWindow(pbNoYield)
            ◊ Call SalYieldEnable(FALSE)
            ◊ Set dfCount = 0
            ♦While dfCount < 250
                ◊ Set dfCount = dfCount + 1
            ◊ Call SalYieldEnable(TRUE)
            ◊ Call SalEnableWindow(pbNoYield)
```

Nach einem Klick auf die Schaltfläche pbNoYield wird die Nachricht SAM_Click an dieses Schaltflächenobjekt gesandt. pbNoYield ruft daraufhin SalYieldEnable(FALSE) auf. Nun wird die Kontrolle nicht mehr an andere laufende Applikationen abgegeben – das Betriebssystem „steht“, bis die Schleife durchlaufen und SalYieldEnable(TRUE) aufgerufen wurde.

4.3.9.3 Nebenläufigkeit in Datenbanken

Beim Einsatz von Datenbanken geht man in der Regel von einer Mehrbenutzerumgebung aus. Hier wird *Nebenläufigkeit* (*concurrency*) definiert als Fähigkeit, mit mehreren Benutzern und mehreren Applikationen gleichzeitig auf dieselben Daten einer Datenbank zugreifen zu können. Die meisten Datenbanken bieten hierfür Locking-Mechanismen an. Ein hoher Grad an Nebenläufigkeit (high concurrency) wird definiert als schneller, effizienter Zugriff auf dieselben Daten durch mehrere Benutzer.

SQLWindows ist ein Datenbank-Entwicklungssystem, das Schnittstellen zu den gängigsten Datenbanksystemen anbietet. SQLWindows bietet eine allgemeine SQL-Schnittstelle mit Fehlerbehandlung an. Die SQL-Befehle werden an die Datenbank weitergereicht, welche dann einen oder mehrere Datensätze an SQLWindows zurückliefert, die einzeln angefordert werden („FetchNext"-Mechanismus). Es liegt in der Verantwortung des SQLWindows-Programmierers, seine Applikation so zu entwerfen, daß diese auch mit den speziellen Anforderungen von Mehrbenutzer-Datenbanken (z.B. Zugriff auf Datensätze, die von anderen Benutzern gesperrt wurden) umgehen kann.

SQLWindows unterstützt keine Nebenläufigkeit, da das Betriebssystem Microsoft Windows 3.x, für das SQLWindows entwickelt wurde, kein Multithreading anbietet. Allerdings bietet SQLWindows Funktionen an, mit denen der Programmierer die Kontrollflußweitergabe sowohl innerhalb der Applikation als auch zwischen Applikationen steuern kann. Dies hat aber nichts mit Nebenläufigkeit im Sinne des Dimensionskataloges zu tun. Für die Unterstützung von nebenläufigen Mehrbenutzer-Datenbanken hat der SQLWindows-Programmierer Sorge zu tragen – SQLWindows stellt lediglich eine allgemeine SQL-Schnittstelle mit Fehlerbehandlung zur Verfügung.

4.3.10 Persistenz

Mit seiner SQL-Sprachunterstützung ist SQLWindows für die Arbeit mit bzw. auf relationalen Datenbanken ausgelegt. Auch die Standard-Datentypen orientieren sich an den in Datenbanken üblichen Datentypen. Das bedeutet, daß alle SQLWindows-Klassen auch als Tabellen in Datenbanken repräsentiert werden können. Ein Beispiel: Gegeben sei eine Klasse fcEmployee:

- Functional Class: fcEmployee
  - Description:
  - Derived From
  - Class Variables
  - Instance Variables
    - String: sEmpID
    - Date/Time: dtBirthday
    - Number: nSalary
    - Date/Time: dtHireDate
  - Functions

Eine passende SQL-Tabellendefinition sähe wie folgt aus:

```
CREATE TABLE EMPLOYEE
  (EmpID              CHAR(6) NOT NULL,
   Birthday           DATE,
   Salary             DECIMAL(10,2) NOT NULL,
   HireDate           DATE,
   PRIMARY KEY (EmpID))
CREATE UNIQUE INDEX EMPLOYEE_IDX
   ON EMPLOYEE (EmpId)
```

Gibt es von dieser Klasse abgeleitete Klassen, so muß die SQL-Tabellendefinition erweitert werden. Leider bietet SQLWindows keine Möglichkeit zur Anbindung von objektorientierten Datenbanken.

Dieses Thema soll hier nicht weiter vertieft werden. Karim Attia hat in seiner Studienarbeit ([Att96]) die Möglichkeiten der Abbildung von Objektstrukturen auf relationale Strukturen eingehend untersucht.

In seiner Eigenschaft als Client/Server-Datenbank-Entwicklungssystem ist SQLWindows für die Arbeit mit bzw. auf relationalen Datenbanken konzipiert. Alle Attribute von SQLWindows-Objekten lassen sich in solchen Datenbanken speichern – eine für den SQLWindows-Programmierer relativ einfache und – abhängig vom verwendeten Datenbanksystem – effiziente Lösung des Persistenzproblems.

4.3.11 Zusammenfassung der Dimensionsidentifikation

Die folgende Tabelle faßt die Identifikation objektorientierter Dimensionen in SAL in Form einer Tabelle zusammen. Ein Häkchen ✓ symbolisiert eine erfüllte Dimension. Ist das Häkchen in Klammern eingefaßt (✓), so wird die entsprechende Dimension nur zum Teil erfüllt. Die Bemerkungen geben Aufschluß über die Einschränkungen. Mit einem Kreuz ✗ versehene Dimensionen werden von SQLWindows nicht erfüllt.

Die erweiternden Dimensionen sind grau unterlegt.

Dimension		Bemerkungen
Objekte	✓	Mehrfachinstantiierung muß verboten werden.
Werte	(✓)	Der Wertcharakter von Fachwert-Klassen kann nicht sichergestellt werden.
Abstraktion	✗	
Typen und Klassen	(✓)	Klassentypen zerstören das einfache Klassen- und Typkonzept.
Referenzen	(✓)	Lediglich call by reference für Basis-Datentypen und benutzerdefinierte Variablen; das SQLWindows Object System (SOS) implementiert einen eingeschränkten Referenzmechanismus (vgl. Kapitel 6).
Einfache Vererbung	✓	Einschränkungen bzgl. des Klassentyps, Objektableitung weicht die Vererbungs- und Instantiierungsregeln auf.
Polymorphismus und dynamisches Binden	(✓)	Kein polymorpher Aufruf von außerhalb des Objektes möglich.
Mehrfachvererbung	(✓)	Keine diamantenförmige Struktur bei wiederholtem Erben möglich.
Nebenläufigkeit	✗	
Persistenz	✓	Mit den SQL-Funktionen von SQLWindows einfach und effizient zu implementieren.

Tabelle 4.3: Identifikation objektorientierter Dimensionen in SAL

4.4 Fazit

Unter der strengen Voraussetzung, daß eine Sprache nur dann als objektorientiert zu bezeichnen ist, wenn sie mindestens alle Kerndimensionen des Dimensionskataloges erfüllt, zählt SAL nicht zu den objektorientierten Sprachen, da sie keine Möglichkeiten zur Abstraktion bietet. Dies und das Fehlen eines durchgängigen Referenzkonzeptes sind das größte Manko von SAL. Die Klassentypen machen das objektorientierte Konzept von SAL unnötig kompliziert, auch wenn man berücksichtigen muß, daß diese wohl eine Altlast aus prozeduralen Tagen darstellen.

Nun könnte man den Vorwurf erheben, daß der Dimensionskatalog zu restriktiv ist, als daß ihn eine Programmiersprache in allen Punkten erfüllen kann. Betrachtet man jedoch die Programmiersprachen C++, Eiffel und Java hinsichtlich des Dimensionskataloges, so erfüllen diese Sprachen zumindest die Kerndimensionen in ihrem vollen Umfang. Objektorientierte Programmiersprachen im Sinne des Dimensionskataloges sind also durchaus möglich und existent.

Sieht man einmal von der Abstraktion ab, so werden alle übrigen Kerndimensionen zumindest in gewisser Weise von SAL erfüllt – wenn auch nicht in ihrem vollen Umfang. Abgesehen vom

fehlenden Referenzkonzept lassen sich all diese Einschränkungen mit Hilfe der Sprachmittel von SAL umgehen. So gesehen, kann man bei SQLWindows doch von einer objektorientierten Sprache sprechen.

Betrachtet man schließlich die in SQLWindows übliche Definition von Klassen aus Objekten heraus und das Konzept der Objektableitung, so kann man SAL auch als prototypische Sprache verstehen.

SAL ist meiner Meinung nach eine Programmiersprache der vierten Generation mit objektorientierten und prototypischen Elementen, wobei die Sprache weder rein objektorientiert (im Sinne des Dimensionskataloges) noch rein prototypisch ist. SAL enthält die Kernkonzepte des objektorientierten Paradigmas: Objekte, Klassen und Vererbung. Zur objektorientierten Programmiersprache im engeren Sinne fehlen ihr aber wichtige Eigenschaften, wobei an erster Stelle die Abstraktion und ein Referenzmechanismus zu nennen sind.

4.5 Zusammenfassung

Inhalt dieses Kapitels ist zum einen die Beschreibung des SQLWindows-Entwicklungssystems und der Programmiersprache SAL (SQLWindows Application Language), und zum anderen die Klärung der Frage, ob SAL eine objektorientierte Programmiersprache im Sinne des Dimensionskataloges aus Kapitel 3 ist.

Zur einfacheren Orientierung sind jeweils die Nummern der Abschnitte angegeben, auf die sich die Zusammenfassung bezieht.

Verzahnung von Sprache und Entwicklungsumgebung (4.1; 4.2)

SAL ist keine eigenständige Programmiersprache – sie ist sehr eng an das Entwicklungssystem SQLWindows gebunden. So wird die Struktur eines SQLWindows-Programms nicht durch die Programmiersprache SAL festgelegt, sondern durch den sogenannten *Outline*, das Editor-Werkzeug von SQLWindows. Die Entwicklungsumgebung von SQLWindows hat einen großen Einfluß auf das Vorgehen bei der Softwareentwicklung. Das proprietäre Dateiformat der SQLWindows-Applikationsdateien, in denen neben dem Programm auch Projektinformationen abgelegt sind, erhöht die Bindung an das Entwicklungssystem.

Datentypen und prozedurale Sprachelemente (4.2)

Neben den Standard-Datentypen definiert SAL zusätzliche spezielle Datentypen, die aus den Bereichen der Datenbanken (z.B. Date/Time) und der Windows-Programmierung (Window Handle) stammen.

SAL enthält die üblichen prozeduralen Sprachkonstrukte, wobei die Syntax einiger Konstrukte unglücklich gewählt ist und die Lesbarkeit der Programme unnötig erschwert.

SAL besitzt Nachrichten (Messages) als zusätzliches Sprachmittel, das jedoch auf dem Windows-Nachrichtenmechanismus beruht und die Sprache eng an das Betriebssystem koppelt.

Objekte (4.3.1; 4.3.4)

Alle drei SAL-Objekttypen (Standard-Fensterobjekte, benutzerdefinierte Fensterobjekte und benutzerdefinierte Variablen) besitzen Funktionen und lokale Variablen. SQLWindows erlaubt die Mehrfachinstantiierung von Hauptfenster-Objekten und verletzt damit das Objektkonzept des Dimensionskataloges. Diese Mehrfachinstantiierung kann jedoch programmtechnisch unterbunden werden.

Werte (4.3.2)

Die „eingebauten“ SQLWindows-Datentypen haben alle Wertcharakter. Das fehlende Abstraktionskonzept erlaubt es jedoch nicht, Fachwerte als Klassen zu realisieren, die einen Wertcharakter der Fachwert-Objekte garantieren.

Abstraktion (4.3.3)

SQLWindows bietet keine Möglichkeit, um Kapselung zu erzwingen: Auf die Variablen und Funktionen eines jeden Objektes kann von jeder Stelle des Programms aus zugegriffen werden. Das Verbergen von Informationen kann in SQLWindows durch die Konvention erreicht werden, auf ein Objekt nur über Funktionen zuzugreifen. Nun macht aber i.d.R. nicht die Gesamtheit der Klassenfunktionen die Schnittstelle der Klasse aus, sondern nur eine Teilmenge der Funktionen. Hier kann nur eine ausführliche Klassendokumentation helfen, die diejenigen Klassenfunktionen nennt, welche die Schnittstelle der Klasse bilden.

Typen und Klassen (4.3.4)

Der Klassenbegriff von SQLWindows muß als eingeschränkt bezeichnet werden:

Nimmt man die Definition des abstrakten Datentyps sehr genau, dann entsprechen die SQLWindows-Klassen nicht dem Klassenbegriff des Dimensionskataloges, allein schon aus dem Grund, daß auf alle Attribute zugegriffen werden kann. Neben Klassen kennt SQLWindows noch Klassentypen, die auf den Standard-Fenstertypen basieren. Diese Klassentypen, die man auch als Basisklassen hätte definieren können, machen das Typkonzept von SQLWindows unnötig kompliziert.

Betrachtet man eine Klasse – abweichend von der Definition im Dimensionskatalog – unter dem Gesichtspunkt einer Schablone für Objekte, die Dienste und einen internen Zustand durch lokale Variablen definiert, dann findet man diese Betrachtungsweise auch in den SQLWindows-Klassen wieder.

Referenzen (4.3.5)

Die Übergabe von Objektreferenzen als Parameter und das Speichern von Objektreferenzen als Attribute sind eine Grundvoraussetzung für die objektorientierte Programmierung. SQLWindows unterstützt lediglich die Übergabe von Referenzen auf Basis-Datentypen und benutzerdefinierte Variablen, nicht aber das Speichern von Objektreferenzen als Attribute. Mit dem in Kapitel 6 beschriebenen SQLWindows Object System (SOS) wurde ein eingeschränkter Referenzmechanismus für SQLWindows geschaffen.

Einfache Vererbung (4.3.6)

Die Implementierung der einfachen Vererbung in SQLWindows entspricht der Definition im Dimensionskatalog. Die Tatsache, daß in Fensterklassen ein Abschnitt Message Actions für die Bearbeitung von Nachrichten für die Klasse existiert, verlangt nach einem Mechanismus, um die Nachricht an die Elternklassen weiterzuleiten. Ein solcher ist in SQLWindows mit den Funktionen SalSendClassMessage() und SalSendClassMessageNamed() vorhanden. Das Konzept der Objektableitung weicht die Vererbungs- und Instantiierungsregeln dahingehend auf, daß es erlaubt, instantiierte Fensterobjekte auf der Objektebene zu erweitern. Die Objektableitung macht SQLWindows zu einer prototypischen Sprache im Sinne Wegners.

Polymorphismus und dynamisches Binden (4.3.7)

SQLWindows unterstützt spätes (dynamisches) Binden von Funktionen und stellt somit einen Mechanismus zur Verfügung, mit dem Polymorphismus realisiert werden kann. Leider wurde dieses Konzept nur unzureichend implementiert: So können polymorphe Aufrufe nicht von außerhalb eines Objektes erfolgen, d.h. ein Funktionsaufruf der Art Call *Objekt..Funktion*() ist nicht erlaubt. Polymorphismus kann in SQLWindows lediglich dazu genutzt werden, um in einer Klasse polymorph eine Funktion aufzurufen, die in einer abgeleiteten Klasse überschrieben wurde.

Mehrfachvererbung (4.3.8)

Die Implementierung der Mehrfachvererbung in SQLWindows entspricht der Definition im Dimensionskatalog. Besitzt eine Klasse mehrere Basisklassen, so dürfen diese nicht von einer gemeinsamen Klasse erben, wenn auch die abgeleitete Klasse Funktionen dieser gemeinsamen Basisklasse nutzen soll. SQLWindows unterstützt nämlich keine diamantenförmigen Vererbungsstrukturen. Diese Tatsache beschränkt die Möglichkeiten der Mehrfachvererbung in SQLWindows.

Nebenläufigkeit (4.3.9)

SQLWindows unterstützt keine Nebenläufigkeit, da das Betriebssystem Microsoft Windows 3.x, für das SQLWindows entwickelt wurde, kein Multithreading anbietet. Allerdings bietet SQLWindows Funktionen an, mit denen der Programmierer die Kontrollflußweitergabe sowohl innerhalb der Applikation als auch zwischen Applikationen steuern kann. Für die Unterstützung von nebenläufigen Mehrbenutzer-Datenbanken hat der SQLWindows-Programmierer Sorge zu tragen – SQLWindows stellt lediglich eine allgemeine SQL-Schnittstelle mit Fehlerbehandlung zur Verfügung.

Persistenz (4.3.10)

In seiner Eigenschaft als Client/Server-Datenbank-Entwicklungssystem ist SQLWindows für die Arbeit mit bzw. auf relationalen Datenbanken konzipiert. Alle Attribute von SQLWindows-Objekten lassen sich in solchen Datenbanken speichern – eine für den SQLWindows-Programmierer relativ einfache und – abhängig vom verwendeten Datenbanksystem – effiziente Lösung des Persistenzproblems.

Ist SAL eine objektorientierte Programmiersprache? (4.4)

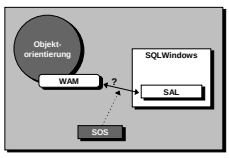
SAL ist meiner Meinung nach eine Programmiersprache der vierten Generation mit objektorientierten und prototypischen Elementen, wobei die Sprache weder rein objektorientiert (im Sinne des Dimensionskataloges) noch rein prototypisch ist. SAL enthält die Kernkonzepte des objektorientierten Paradigmas: Objekte, Klassen und Vererbung. Zur objektorientierten Programmiersprache im engeren Sinne fehlen ihr aber wichtige Eigenschaften, wobei vor allem die Abstraktion und ein Referenzmechanismus zu nennen sind.

5 Entwicklung eines WAM-Frameworks für SQLWindows

Das vorangegangene Kapitel hat gezeigt, daß SQLWindows als Programmiersprache mit objektorientierten Elementen betrachtet werden kann. Gegenstand dieses Kapitels ist die Untersuchung, inwieweit man ein Framework für die Werkzeug-Material-Metapher (WAM-Metapher) in SQLWindows implementieren kann. Ein Teil der WAM-Metapher wird in [RZ95] in Form von Entwurfsmustern beschrieben und wurde bereits als Framework in C++ (BibV*xx*-Framework des Arbeitsbereichs Softwaretechnik) und in Smalltalk (vgl. [RS95]) implementiert. Im Rahmen dieser Diplomarbeit wird versucht, ein vergleichbares Framework in SQLWindows zu entwickeln.

Abschnitt 5.1 beschreibt zunächst kurz die Werkzeug-Material-Metapher. Anschließend wird in Abschnitt 5.2 – ausgehend von den Metaphern und den daraus entwickelten Entwurfsmustern – die Entwicklung des WAM-Frameworks beschrieben. Hier zeigt sich, ob die SAL-Sprachelemente mächtig genug sind, um alle Entwurfsmuster implementieren zu können, und ob eventuelle SAL-Spracherweiterungen der Entwurfsmuster-Implementierung dienlich sein können. Abschnitt 5.3 erläutert die Benutzung des Frameworks anhand eines Beispiels. Dieser Abschnitt klärt auch die Frage, warum das WAM-Framework wirklich ein Framework ist, und keine Klassenbibliothek. Außerdem werden die Programmier- und Dokumentationsrichtlinien beschrieben, die bei der Entwicklung und der Benutzung des Frameworks verwendet werden. Abschnitt 5.4 gibt eine Zusammenfassung dieses Kapitels.

Die Klassen des TANDEM-Frameworks werden im Detail in Anhang A beschrieben.



Dieses Kapitel versucht, die WAM-Methodik einerseits und SQLWindows andererseits in Form eines Frameworks zusammenzubringen. Neben dem eigentlichen Framework werden auch die Konventionen und Programmierrichtlinien für die Entwicklung und die Benutzung des Frameworks beschrieben.

5.1 Die Werkzeug-Material-Metapher

Die Werkzeug-Material-Metapher, kurz: WAM-Metapher, ist eine Sammlung von Metaphern, die dem Softwareentwickler für alle Bereiche der Entwicklung interaktiver Softwaresysteme (Analyse des Anwendungsgebietes, fachlicher und technischer Entwurf, Implementierung) Unterstützung bieten. Dadurch kann der durchaus übliche Bruch beim Übergang von der informellen Beschreibung der Problemdomäne zum formalen Softwaresystem vermieden werden. Die Dokumente für Analyse und Entwurf (Szenarios, Glossare und Visionen) sollen hier nicht weiter betrachtet werden (eine ausführliche Beschreibung findet man z.B. in [KGZ94]). Diese Arbeit konzentriert sich auf die Metaphern, die das Bindeglied zwischen fachlichem und technischem Entwurf darstellen. Namentlich sind dies Werkzeuge, Materialien und die Umgebung.

5.1.1 Kontext der Metaphern

Die von der WAM-Metapher adressierten Anwendungsgebiete sind Büroumgebungen, allgemeiner: Arbeitsplätze für weitgehend selbstbestimmte menschliche Tätigkeit, d.h. der Benutzer hat die Verantwortung für ein Aufgabengebiet, ohne daß eine Reihenfolge für die Erledigung der ihm übertragenen Aufgaben vorgegeben ist. Diese Wahlfreiheit in der Reihenfolge der Bearbeitung ist es, was das Wesen (idealer) interaktiver Softwaresysteme ausmacht. Dieses Verständnis der Fähigkeiten und Verantwortlichkeiten der Benutzer wird als das *Leitbild* der Metapher bezeichnet.

Der durch die Metapher unterstützte Softwareentwicklungsprozeß selbst ist evolutionär und partizipativ: Die Teilaufgaben sind zeitlich nicht fest geordnet, sondern werden zyklisch, z.T. parallel oder verschränkt bearbeitet (das gilt auch für die Dokumentation). Während der gesamten Entwicklung wird dem Dialog mit dem späteren Benutzer eine hohe Bedeutung beigemessen. Dieser ist aktiv am Softwareentwicklungsprozeß beteiligt und soll dabei helfen, ein möglichst gutes Abbild (Modell) des bestehenden Anwendungsgebietes zu erstellen – Grundvoraussetzung für eine hohe Akzeptanz des Softwaresystems. Der Softwareentwicklungsprozeß soll hier nicht näher erläutert werden (vgl. z.B. [KGZ94], [Flo93]).

5.1.2 Die Metaphern

Bei ihrer täglichen Arbeit benutzen Menschen Werkzeuge, um auf Materialien zu arbeiten. Dies gilt sowohl im handwerklichen und produzierenden Bereich, als auch bei der Büroarbeit. Bei letzterer spräche man wohl eher von Arbeitsmitteln und -gegenständen, doch zur allgemeinen, anwendungsgebietsübergreifenden Beschreibung menschlichen Arbeitshandelns sollen die Begriffe Werkzeug und Material als Metaphern verwendet werden. Die folgende Beschreibung der Metaphern orientiert sich an [KGZ94] und [Zül94].

Materialien

Dinge des Anwendungsgebietes, die im Rahmen einer Aufgabenerledigung den Charakter eines Arbeitsgegenstandes aufweisen, werden den *Materialien* (*materials*) zugeordnet. Materialien sind die Zwischen- und Endprodukte von Arbeitsprozessen. Materialien lassen sich in bestimmter Weise bearbeiten. Dabei wird unterschieden zwischen dem Verändern und dem Sondieren eines Materials. Materialien selbst sind passiv. Die Bearbeitung wird mit Hilfe von Werkzeugen durchgeführt.

Als Beispiel aus dem Bürobereich mag der bereits in Kapitel 2 eingeführte Ordner dienen. Dieser ist selbst ein Material im obigen Sinne, das jedoch weitere Gegenstände oder Materialien, wie ein Inhaltsverzeichnis, Dokumente und Zwischenblätter, enthalten kann. Abbildung 5.1 stellt diese Materialien dar.

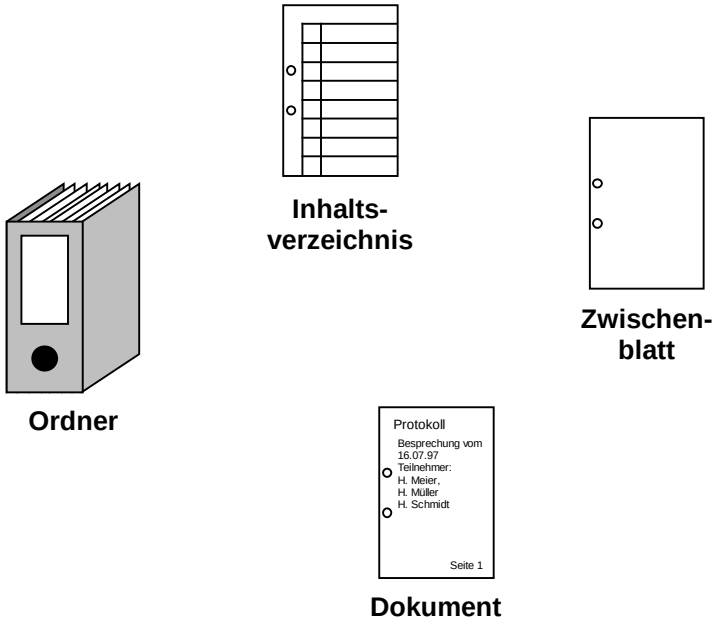


Abbildung 5.1: Ein Ordner und Gegenstände, die ein Ordner enthalten kann

Wie bereits gesagt, lassen sich an einem Material verändernde und sondierende Tätigkeiten durchführen. So kann ein Ordner beschriftet werden, was ihn verändert (zu beachten ist, daß es immer noch derselbe Ordner ist wie vor der Beschriftung – die Veränderung ändert nichts an der Identität des Ordners). Wenn man festlegt, daß ein Ordner immer ein Inhaltsverzeichnis besitzt, so kann man dieses durch Sondieren des Ordners (genauer: seines Inhaltes) erhalten und führen. Des weiteren kann man in einen Ordner Dokumente und Zwischenblätter einfügen bzw. aus diesem entnehmen. Inhaltsverzeichnis, Dokumente und Zwischenblätter sind wiederum selber Materialien mit bestimmten Umgangsformen.

Da Materialien passiv sind, können sie nur indirekt mit Hilfe von Werkzeugen bearbeitet werden.

Werkzeuge

Dinge des Anwendungsgebietes, die im Rahmen einer Aufgabenerledigung den Charakter eines Arbeitsmittels aufweisen, werden den *Werkzeugen* (*tools*) zugeordnet. Werkzeuge helfen bei der Herstellung des Arbeitsergebnisses. Wir hantieren mit Werkzeugen, wenn wir Material bearbeiten. Ein Werkzeug kann für unterschiedliche Materialien geeignet sein.

Zu den allgemeinen im Bürobereich verwendeten Materialien wie Zwischenblatt oder Dokument aus obigem Beispiel lassen sich leicht Werkzeuge wie Locher, Hefter oder Schreiber finden, mit denen diese Materialien bearbeitet werden können. Für einen Ordner lassen sich nicht ohne weiteres geeignete Werkzeuge finden. Bei der Modellierung des Anwendungsbereiches in Form eines Softwaresystems müssen oftmals Werkzeuge entworfen werden, die in der realen Welt in dieser

Form nicht existieren. So läßt sich der Inhalt von Ordnern mit einem speziellen Werkzeug, genannt „Ordner-Browser“, durchsuchen. Eine andere softwaretechnisch motivierte Werkzeugklasse ist die der Editoren, mit denen Werkzeuge bearbeitet werden können.

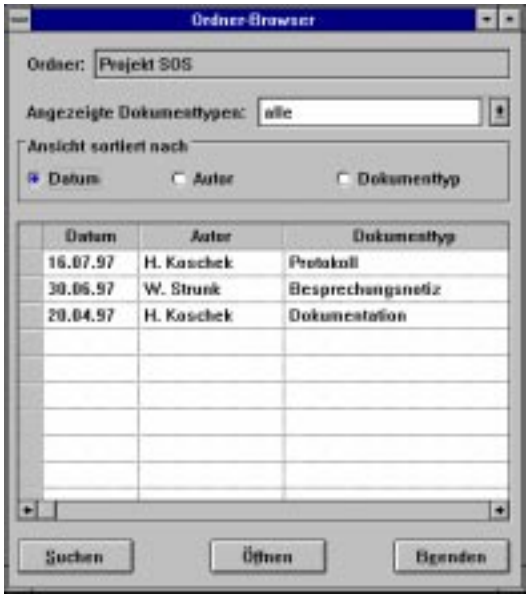


Abbildung 5.2: Ein interaktives Browser-Werkzeug für Ordner

Abbildung 5.2 stellt ein Browser-Werkzeug für Ordner dar. An diesem Beispiel lassen sich die Eigenschaften von Software-Werkzeugen aufzeigen, wie sie in [KGZ94] beschrieben sind:

Ein Software-Werkzeug

- hat einen Namen und kann über ein graphisches Symbol aktiviert werden.
- zeigt immer eine Sicht auf das gerade bearbeitete Material (in diesem Fall der Inhalt eines Ordners).
- bleibt bei der Benutzung selbst „im Blick“, d.h. bei der Bearbeitung des Materials ist immer erkennbar, mit welchem Werkzeug es bearbeitet wird.
- bietet mögliche Handhabungen an, um das Material unterschiedlich zu bearbeiten (z.B. „Suchen“).
- zeigt jederzeit seine Einstellungen, die die Materialsicht bestimmen (z.B. die Sortierung der Elemente des Ordners nach dem Datum).

Als „Bindeglied“ zwischen Materialien und Werkzeugen dienen die Aspekte.

Aspekte

Bei der Bearbeitung eines Materials bedient sich ein Werkzeug eines bestimmten Aspektes dieses Materials. Ein *Aspekt* (*aspect*) repräsentiert eine mögliche Bearbeitungsform für ein Material, die zur Durchführung einer bestimmten Aufgabe benötigt wird. Ein Aspekt ist eine aufgabenspezifische Sicht eines Werkzeuges auf ein Material. Aspekte drücken das „Zueinanderpassen“ von Werkzeugen und Materialien aus.

Softwaretechnisch werden Aspekte in Form von Aspektklassen implementiert, die Werkzeugen eine Schnittstelle anbieten. Die Materialklassen erben von solchen Aspektklassen, so daß sie einem Werkzeug unter der Aspekt-Schnittstelle bekanntgemacht werden können. Abbildung 5.3 macht diesen Zusammenhang am Beispiel des Ordners deutlich. Die Ordner-Klasse erbt von der Aspektklasse Auflistbar. Die Werkzeugklasse Browser benutzt den Aspekt Auflistbar, um den Inhalt von Materialien wie dem Ordner anzuzeigen.

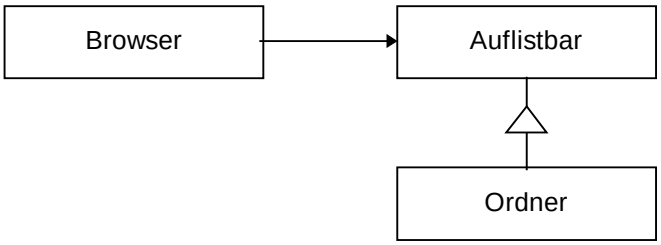


Abbildung 5.3: Zusammenhang zwischen dem Werkzeug „Browser“, dem Aspekt „Auflistbar“ und dem Material „Ordner“

Diese Beziehungen zwischen den Metaphern Werkzeug, Aspekt und Material machen die Kernidee des WAM-Ansatzes aus.

Bei der Analyse des Anwendungsbereiches müssen zunächst die Gegenstände des Anwendungsbereiches identifiziert werden. Dabei ist zu unterscheiden, welche dieser Gegenstände als Werkzeuge betrachtet werden können, und welche den Materialien zuzuordnen sind. Es kann dabei durchaus vorkommen, daß ein Gegenstand sowohl Werkzeug als auch Material ist. Die Werkzeuge werden anschließend auf ihren Umgang mit Materialien untersucht. Ziel ist es, die Hauptmerkmale in Form von Aspekten zu beschreiben.

Zur umfassenden Interpretation des Anwendungsbereiches sind weitere Metaphern vonnöten, wie z.B. Archive, Automaten oder Arbeitsumgebungen. Hier sei auf die oben erwähnte Literatur verwiesen. Die Umgebungs-Metapher soll jedoch an dieser Stelle kurz beschrieben werden.

Umgebung

Wir führen alle Tätigkeiten in einer Arbeitsumgebung aus, beispielsweise in einer Werkstatt oder an einem Schreibtisch. Innerhalb dieses Raumes organisieren wir unsere Werkzeuge und Materialien, sowohl in geistiger als auch in räumlicher Hinsicht. Dieser Raum wird *Umgebung* (*environment*) genannt.

Die obige Beschreibung der Metaphern ist hauptsächlich fachlicher Natur. Mit Hilfe dieser Metaphern läßt sich ein fachlicher Entwurf des Anwendungsbereiches erstellen. Bei vielen Methoden zur Softwareentwicklung kommt es jedoch beim Übergang vom fachlichen zum technischen Entwurf zu einem Bruch, wenn die fachlichen Anforderungen auf die softwaretechnischen Methoden und Mittel zur Realisierung projiziert werden. Die Stärke des WAM-Ansatzes liegt nun darin, diese „semantische Lücke“ zwischen fachlichem und technischem Entwurf dadurch so klein wie möglich halten zu können, daß die Metaphern auch für den technischen Entwurf existieren – in Form von Werkzeug-, Aspekt- und Materialklassen. Die Charakteristika dieser Klassen

lassen sich als Entwurfsmuster im Sinne von Gamma et. al. (vgl. [GHJV95]) beschreiben. [RZ95] beschreibt die wesentlichen WAM-Metaphern als Entwurfsmuster. Es lag also nahe, diesen Musterkatalog als Grundlage für ein WAM-Framework für SQLWindows zu nehmen.

5.2 Implementierung der WAM-Metaphern und -Entwurfsmuster

Dieser Abschnitt beschreibt den Versuch, ein WAM-Framework für SQLWindows auf Basis der in [RZ95] beschriebenen WAM-Entwurfsmuster zu entwickeln. Eine ausführliche Beschreibung der Entwurfsmuster und der ihnen zugrunde liegenden Metaphern wird man hier vergeblich suchen – der interessierte Leser sei auf [RZ95] und [KGZ94] verwiesen.

Bevor jedoch das erste WAM-Entwurfsmuster implementiert werden konnte, mußte ein Mangel von SQLWindows behoben werden, der auch einigen „echten“ objektorientierten Programmiersprachen anhaftet: Es gibt keinen technischen Objektbegriff in Form einer Basisklasse, die Grundinformationen über Objekte und Klassen zur Verfügung stellt, also ein (einfaches) Meta-Objekt-Protokoll implementiert. Abschnitt 5.2.1 beschreibt die Basisklasse fcObject, die ein solches Objektmodell implementiert. Die Klassen- und Objektinformationen werden vor allem für das SQLWindows Object System (SOS) benötigt, das einen Ersatz für den fehlenden Referenzmechanismus bietet (vgl. Kapitel 6).

Anschließend werden in den Abschnitten 5.2.2 und 5.2.3 die technischen Aspekte der Metaphern „Werkzeug“ und „Material“ beschrieben, deren fachliche Aspekte bereits Gegenstand des Abschnitts 5.1.2 waren.

In den Abschnitten 5.2.4 bis 5.2.12 werden schließlich die WAM-Entwurfsmuster eingeführt und auf ihre Implementierbarkeit in SAL hin untersucht.

5.2.1 Die Basisklasse fcObject

Die Basisklasse fcObject verwaltet Informationen über den Objekttyp und die Objekt-ID. Da SQLWindows keine Klassenkonstruktoren und -destruktoren kennt, enthält diese Klasse außerdem Funktionen, die beim Erzeugen bzw. Zerstören eines Objektes aufgerufen werden sollen.

Eine ausführliche Beschreibung der Klassenfunktionen findet man in Abschnitt A.2.7 (Anhang A). Die Idee dieser Klasse geht zum großen Teil auf die Klasse clsObject des E.I.S. Application Builder zurück (vgl. [EIS96]).

5.2.1.1 Typ- und Objektinformationen

Jede Klasse, die von fcObject erbt, setzt mit der Funktion SetObjectType() ihren Klassennamen (den *Objekttyp*). Dieser kann dann mit GetObjectType() abgefragt werden. Das manuelle Setzen des Klassennamens ist notwendig, da keine Möglichkeit besteht, den Klassennamen automatisch zu bestimmen (d.h. es gibt kein Meta-Objekt-Protokoll). Das Setzen des Klassennamens sollte in der Funktion Init() erfolgen (siehe auch den folgenden Abschnitt).

5.2.1.2 Objekt-Lebenszyklus

fcObject definiert zwei Funktionen, die den Lebenszyklus eines Objektes festlegen: Init() initialisiert das Objekt. Damit beginnt seine Lebenszeit. Die Funktion Done() beendet schließlich die Lebenszeit des Objektes. Vor dem Aufruf von Init() und nach dem Aufruf von Done() ist keine Arbeit mit dem Objekt erlaubt. Aus Sicht der Klasse kann man Init() und Done() als Konstruktor bzw. Destruktor betrachten.

Die Funktion Reset() dient dazu, das Objekt auf einen definierten Zustand zurückzusetzen. Sie darf jederzeit während der Lebenszeit des Objektes aufgerufen werden. Init() ruft Reset() polymorph auf, um die Klassenattribute in einen definierten Zustand zu versetzen. Diese Art, in einer Basisklassenfunktion einen Algorithmus zu definieren, dessen konkrete Implementierung in Teilen an abgeleitete Klassen übertragen werden kann, beschreiben Gamma et. al. mit dem Entwurfsmuster „Template Method“ (vgl. [GHJV95]).

Ein Objekt ist immer in einem von drei Zuständen, die in der folgenden Tabelle aufgeführt sind. Mit der Funktion GetObjectState() kann der momentane Objektzustand ermittelt werden. Die Funktion liefert eine der drei symbolischen Konstanten zurück, die ebenfalls in der Tabelle aufgeführt sind.

Zustand	GetObjectState() liefert...
Das Objekt wurde noch nicht initialisiert (Init() wurde noch nicht aufgerufen)	OBJECT_NOT_INITIALIZED
Das Objekt wurde initialisiert (Init() wurde aufgerufen)	OBJECT_INITIALIZED
Das Objekt ist nicht mehr gültig (Done() wurde aufgerufen)	OBJECT_INVALID

Tabelle 5.1: Zustände eines Objektes

Die Funktion Alive() liefert TRUE, wenn das Objekt „lebendig“ ist (d.h. Init() wurde aufgerufen, aber Done() wurde noch nicht aufgerufen), sonst FALSE.

Um eine korrekte Initialisierung des Objektes gewährleisten zu können, muß bei einer von fcObject abgeleiteten Klasse in der Funktion Init() als erstes die (überschriebene) Init()-Funktion der Basisklasse aufgerufen werden. Nur so werden die geerbten Attribute korrekt initialisiert. Dies gilt analog für die Funktion Reset(). Damit die Aufräumarbeiten nach dem „Ableben“ des Objektes vollständig durchgeführt werden, wird in der Funktion Done() nach den eigentlichen Aufräumarbeiten die Funktion Done() der Basisklasse aufgerufen.

Der interne Ablauf bei der Initialisierung soll am Beispiel zweier Klassen fcA und fcB deutlich gemacht werden, die miteinander in folgender Vererbungsbeziehung stehen:

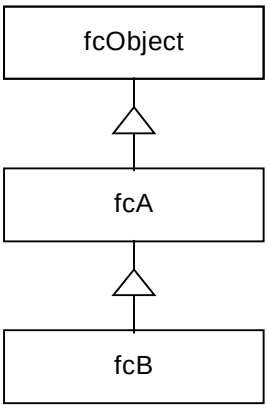


Abbildung 5.4: Vererbungsbeziehungen der Klassen fcA und fcB

Beide Klassen überschreiben die Funktionen Init() und Reset(). Nun werde ein Objekt vom Typ fcB erzeugt:

- fcB: udvB

Ein Aufruf von udvB.Init() löst folgende Funktionsaufrufe bei den überschriebenen Funktionen der Oberklassen aus:

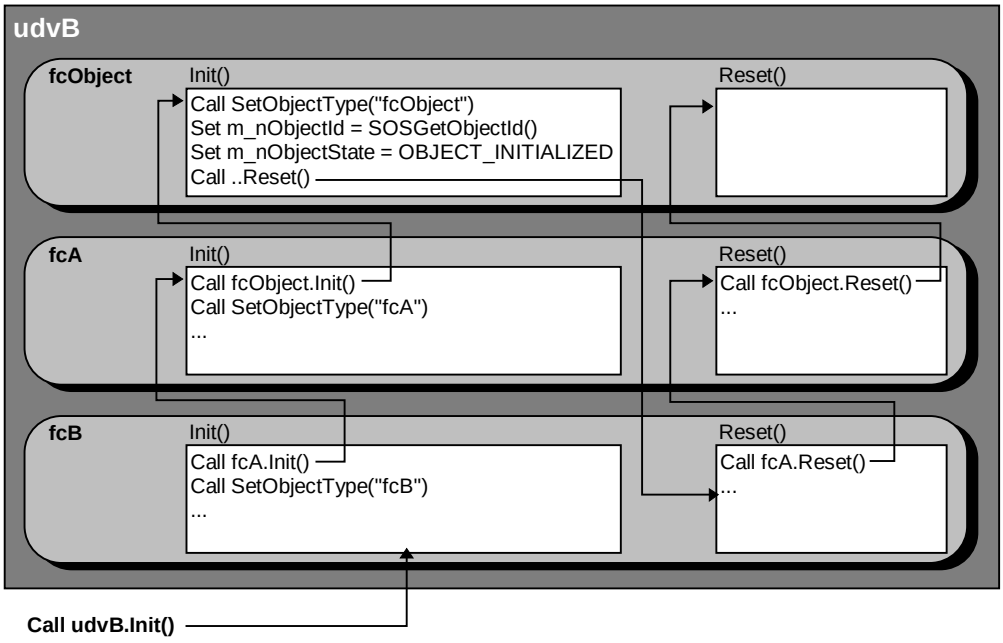


Abbildung 5.5: Initialisierung einer Instanz der Klasse fcB

Wie aus der Abbildung ersichtlich ist, wird fcB.Reset() bereits vor allen Anweisungen in fcB.Init() aufgerufen. Dies muß bei der Implementierung der Init()-Funktion beachtet werden.

5.2.2 Technische Aspekte der Material-Metapher

Die Material-Metapher wurde bereits kurz in Abschnitt 5.1.2 beschrieben. Bei verschiedenen Materialien kann man i.d.R. keine gemeinsamen Eigenschaften ausmachen, die zu einer Oberklasse Material führen. Materialklassen werden also immer von Grund auf neu entwickelt. Interessanterweise gibt es jedoch sowohl im Smalltalk-Framework von Dirk Riehle und Martin Schnyder (vgl. [RS95]) als auch in diesem Framework eine Material-Basisklasse. Riehle und Schnyder kapseln in dieser Klasse Informationen über die Aspekte, unter denen ein bestimmtes Material betrachtet werden kann, und können dadurch die Aspekt-Metapher in Smalltalk implementieren, was aufgrund der fehlenden Mehrfachvererbung in dieser Sprache zunächst unmöglich schien. Das im Rahmen dieser Arbeit entwickelte Framework definiert eine Basisklasse fcMaterialBase, die Funktionen enthält, welche beim Schreiben von Attributwerten überprüfen, ob sich der Wert geändert hat, und entsprechend ein Modifikations-Flag setzen. Auf diese Weise werden nur diejenigen Materialien in die Datenbank geschrieben, deren Zustand sich geändert hat, was zu Performanzgewinnen führen kann. Die Speicherung von Materialien in Datenbanktabellen ist der natürlichste Weg, wenn man bedenkt, daß SQLWindows für die Arbeit mit bzw. auf Datenbanken konzipiert wurde. Allerdings werden die Datenbankzugriffe komplett in den Klassen zur Materialverwaltung (vgl. Abschnitt 5.2.11) gekapselt. Die Verwendung von Materialklassen und entsprechenden Materialverwaltungsklassen erlaubt somit eine Abstraktion vom konkreten Speicherkonzept der Daten, oder, mit anderen Worten, eine Trennung von Datenbankmodell und Benutzungsmodell:

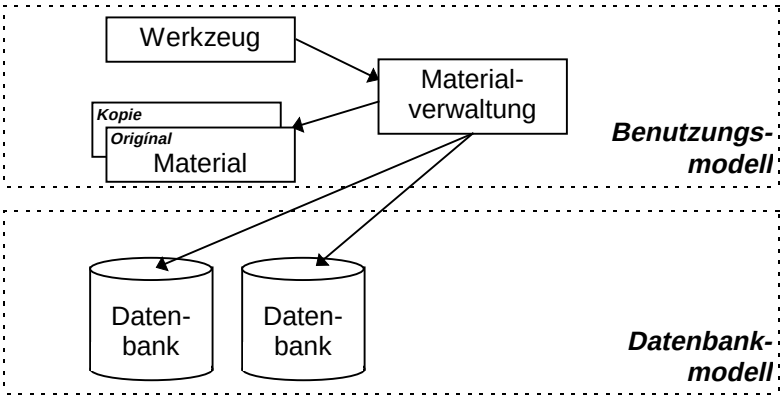


Abbildung 5.6: Trennung von Datenbankmodell und Benutzungsmodell

Durch diese Trennung wird zusätzlich zur Datenbank-Schicht ein zweiter Persistenzraum geöffnet, in dem sich die Materialien bewegen. Das bedeutet nicht nur zusätzlichen Speicherbedarf, sondern auch neuen Verwaltungsaufwand, da Integritätsprobleme und – bei Mehrbenutzerbetrieb – auch Synchronisationsprobleme auftreten. Diese Probleme werden von den Datenbank-Verwaltungssystemen bereits gelöst; für das Benutzungsmodell muß nun zusätzlich eine Lösung gefunden werden. Da man aber von den ausgereiften Methoden und Verfahren der Datenbank-Systeme profitieren kann, stellt dies einen kalkulierbaren Aufwand dar, durch den man schließlich die Unabhängigkeit vom konkreten Speicherkonzept gewinnt.

Eng verbunden mit dem Materialbegriff ist die in der WAM-Methodik vertretene Idee von Original und Kopie. Sieht man die in der Datenbank gespeicherten Werte als Original an, so können

mehrere Material-Objekte existieren, die dieselben Werte repräsentieren – das Material wurde kopiert. Der Umgang mit solchen Kopien, insbesondere die Frage der Zugriffsrechte (lesend und/oder schreibend), muß genau geregelt werden. Dazu kennt die Materialverwaltung bestimmte Strategien für die Materialhaltung. Die vorliegende Implementierung der Materialklasse und der Materialverwaltung bietet keine konzeptuelle Unterstützung für die Original/Kopie-Problematik.

Beim Entwurf von Materialien werden Repräsentations- und Interaktionsaspekte nicht berücksichtigt. Es ist Aufgabe der entsprechenden Werkzeuge, Visualisierungen für die Materialien zu finden und einen Kontext für die grafische und logische Bearbeitung der Materialien zur Verfügung zu stellen.

Eine ausführliche Beschreibung der Klasse fcMaterialBase findet man in Abschnitt A.2.5.

5.2.3 Technische Aspekte der Werkzeug-Metapher

Die allgemeine Beschreibung der Werkzeug-Metapher in Abschnitt 5.1.2 gibt keinerlei Hinweise darauf, wie ein Werkzeug aufgebaut sein soll und wie es Materialien präsentieren kann bzw. als Hilfsmittel zur Bearbeitung von Materialien dienen kann.

Werkzeuge bieten eine Sicht auf Materialien und geben dem Benutzer Rückmeldung über deren Zustandsveränderungen. Dabei haben Werkzeuge einen eigenen Zustand, der von der Arbeitssituation abhängig ist.

Gut entworfene Werkzeuge treten bei der Benutzung in den Hintergrund und lassen den Eindruck entstehen, man arbeite direkt auf den Materialien. Trotzdem sind sie immer dann zur Hand, wenn man sie benötigt.

Oftmals können bei einer bestimmten Aufgabe mehrere Teilaufgaben unterschieden werden, die voneinander unabhängig ausgeführt werden können. Eine ähnliche Aufteilung kann auch bei Werkzeugen geschehen, indem man ein Werkzeug aus mehreren Sub-Werkzeugen zusammensetzt.

Für die Konstruktion von Werkzeugen und deren Beziehungen zu Materialien sowie die Integration von Werkzeugen mit anderen Werkzeugen und Materialien in eine Umgebung führen Riehle und Züllighoven in [RZ95] eine Reihe von Entwurfsmustern an. Der Versuch, diese in SAL zu implementieren, ist Gegenstand der nun folgenden Abschnitte.

Ein Waagensymbol am Seitenrand leitet die Diskussion ein, ob und inwieweit das jeweilige Entwurfsmuster in SQLWindows implementiert werden kann.

5.2.4 Tool and Material Coupling

Dieses Entwurfsmuster besagt, daß Werkzeuge und Materialien über Aspektklassen gekoppelt werden sollten. Dadurch werden die Werkzeuge unabhängig vom konkreten Material, das sie bearbeiten, und bleiben offen für neue Materialien.

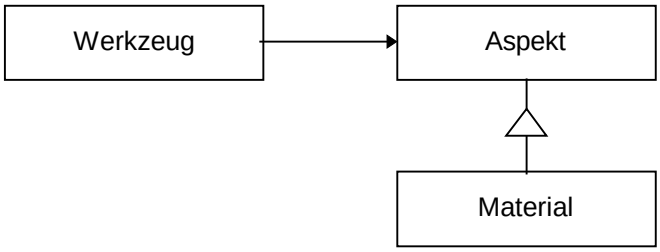


Abbildung 5.7: Werkzeuge und Materialien sind durch Aspekte miteinander verbunden

Da SAL Mehrfachvererbung unterstützt, ist die Implementierung von Aspektklassen möglich. Allerdings kann ein Material dem Werkzeug nicht unter dem Namen des Aspektes bekannt gemacht werden. Der Grund ist das fehlende Referenzkonzept: Werkzeuge können keinen Referenzen auf Materialien halten – sie können lediglich auf ein globales Materialobjekt zugreifen. Dieses globale Materialobjekt ist technisch gesehen vom Material-Klassentyp. Es birgt zwar den Aspekt und somit auch dessen Schnittstelle, aber dem Werkzeug ist es nicht möglich, das Material nur unter dieser Aspekt-Schnittstelle anzusprechen.

Um die globale Deklaration des Materialobjektes zu verhindern, sollte der Zugriff auf das Material-Objekt mittels einer verwaltenden Klasse beschränkt werden. Diese Aufgabe sollte eine Materialverwaltungsklasse übernehmen:

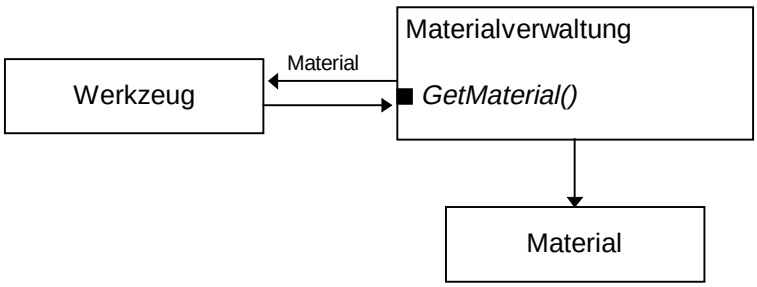


Abbildung 5.8: Materialverwaltung

Diese Idee scheiterte wiederum an den sprachlichen Beschränkungen von SAL. Es ist nämlich nicht möglich, ein Objekt (oder eine Referenz auf ein Objekt) als Rückgabewert einer Funktion zu definieren. Als Rückgabewerte sind nur die Basis-Datentypen erlaubt.

Mit der Einführung des SQLWindows Object System (SOS) war es möglich, Referenzen auf Objekte zu übergeben. Dazu speichert die Materialverwaltung das Materialobjekt im SOS und übergibt dem Werkzeug die Objekt-ID, anhand derer das Werkzeug das Materialobjekt aus dem SOS lesen kann:

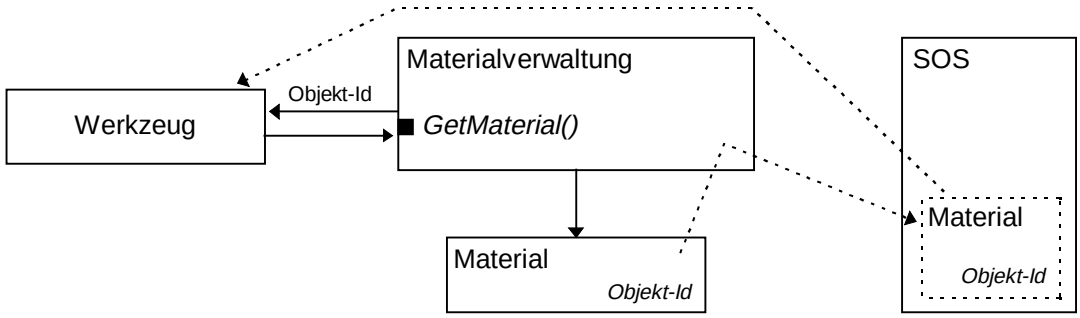


Abbildung 5.9: Zusammenspiel von Materialverwaltung und SOS

Mit der Speicherung im SOS wird implizit ein Kopie-Begriff für Materialien eingeführt, indem diese nun „im Original“ als SQLWindows-Objekt und „in Kopie“ als SOS-Objekt vorliegen. Da das SOS keine Kenntnis über Änderungen am Original-Objekt besitzt, kann es passieren, daß Original und Kopie inkonsistent werden. Nur für den Zeitpunkt direkt nach der Speicherung eines Objektes im SOS kann die Konsistenz von Original und Kopie garantiert werden. Dieser Tatsache muß man sich bewußt sein, wenn man sicher mit dem SOS arbeiten möchte.

Aspektklassen lassen sich in SAL zwar implementieren, aber nicht nutzen, da keine Objektreferenzen gespeichert werden können. Mit Hilfe des SOS kann eine Materialverwaltung Referenzen auf Materialobjekte an Werkzeuge übergeben.

5.2.5 Tool Composition

Wie bereits erwähnt, kann man – analog zur Unterteilung einer Aufgabe in mehrere Teilaufgaben – ein Werkzeug aus mehreren Sub-Werkzeugen zusammensetzen, die jeweils eine spezifische (Teil-)Aufgabe erfüllen.

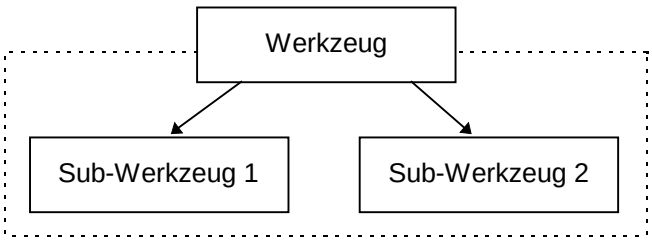


Abbildung 5.10: Zusammensetzen von Werkzeugen aus Sub-Werkzeugen

Oftmals können solche Sub-Werkzeuge auch in anderen Werkzeugen verwendet werden. So benötigen viele Werkzeuge einen Auflister oder einen Editor. Werden solche Sub-Werkzeuge möglichst allgemein und flexibel entworfen, so fördert das die Wiederverwendbarkeit dieser Komponenten.

Werkzeuge können also entweder einfach oder zusammengesetzt sein. Riehle und Züllighoven bilden diese Eigenschaften in [RZ95] in den Klassen Tool, Simple Tool und Compound Tool ab, deren Klassenbeziehungen aus Abbildung 5.11 ersichtlich sind.

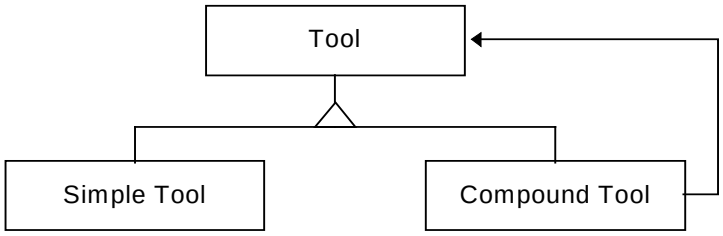


Abbildung 5.11: Klassenbeziehungen der Werkzeug-Komposition

Ein Compound Tool-Objekt kann eines oder mehrere Tool-Objekte „enthalten“ (benutzen), wobei diese Tool-Objekte wiederum Compound Tool-Objekte sein können. Da SQLWindows keinen Referenzmechanismus besitzt, wäre eine solche Benutztbeziehung zwischen (Sub-)Werkzeugen nur sehr unelegant über das Speichern von Fensterbezügen zu realisieren gewesen. Das SOS als Ersatz für einen Referenzmechanismus wäre mit der Aufgabe überfordert. Da i.d.R. reger Gebrauch von den Tool-Objektreferenzen gemacht wird, bedeutete dies, daß ständig Objekte aus dem SOS ausgelesen werden müßten. Dadurch käme es zu Einbußen bei der Performanz des Softwaresystems. Im Rahmen des hier realisierten Frameworks wurde deshalb auf die Möglichkeit rekursiv verschachtelter (Sub-)Werkzeuge verzichtet.

Die Beschreibung des Entwurfsmusters „IP/FP Plug In“ in Abschnitt 5.2.8 zeigt, wie Sub-Werkzeuge in SQLWindows realisiert werden können.

Die Aufteilung eines Werkzeuges in mehrere Sub-Werkzeuge ist auch in SQLWindows-Programmen möglich. Da jedes Fenster (sowohl Haupt- als auch Kindfenster) ein eigenes Objekt darstellt, kann man es als eine Art Sub-Werkzeug betrachten. Die Möglichkeit rekursiv verschachtelter (Sub-)Werkzeuge wurde in diesem Framework nicht realisiert, da SQLWindows keinen Referenzmechanismus besitzt.

5.2.6 Separation of Powers

Dieses Entwurfsmuster beschreibt die Aufteilung eines Werkzeugs in zwei Komponenten:

- Die *Interaktionskomponente* (*interaction part*, *IP*) dient der Präsentation und Handhabung des Materials. Sie definiert und erzeugt die Benutzerschnittstelle des Werkzeugs und nimmt Eingaben vom Benutzer entgegen.
- Die *Funktionskomponente* (*functional part*, *FP*) kapselt die Funktionalität des Werkzeugs. Sie liefert die gewünschten Informationen an die Interaktionskomponente oder verändert das Material wie gefordert.

Die Interaktionskomponente sollte austauschbar sein, ohne daß dadurch die Funktionskomponente geändert werden müßte. So lassen sich leicht unterschiedliche Benutzeroberflächen für ein Werkzeug implementieren, die je nach Einsatzkontext verwendet werden können.

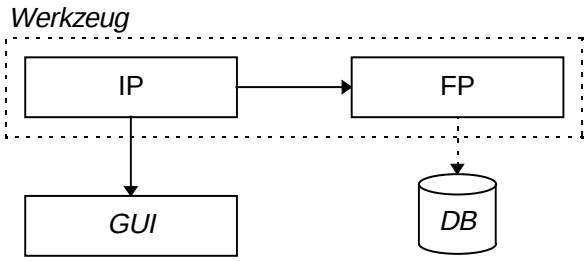


Abbildung 5.12: Zusammenspiel von Funktions- und Interaktionskomponente

Obige Abbildung zeigt die Benutztbeziehungen zwischen Funktions- und Interaktionskomponente. Letztere hat zusätzlich Zugriff auf eine Benutzeroberflächen-Bibliothek (Graphical User Interface, GUI). Im Falle einer Datenbankanwendung hätte die Funktionskomponente den Zugriff auf einen Materialverwalter (in obiger Abbildung nicht dargestellt), der die Daten aus der Datenbank in Form von Materialien zur Verfügung stellt. Die daraus entstehende Problematik des zweiten Persistenzraumes wurde bereits in Abschnitt 5.2.2 behandelt.

Wie aus der Abbildung ersichtlich, benutzt die Interaktionskomponente die Funktionskomponente, aber nicht umgekehrt. Der Grund dafür ist die Forderung nach Austauschbarkeit der Interaktionskomponente: Dies kann nur erfüllt werden, wenn die Funktionskomponente keine Schnittstellenfunktionen der Interaktionskomponente aufruft. Wie kann nun die Funktionskomponente mit der Interaktionskomponente kommunizieren, ohne deren Schnittstelle kennen zu müssen? Diese Frage beantwortet das Entwurfsmuster „Event Mechanism“, das im nächsten Abschnitt vorgestellt wird.

Es ist grundsätzlich auch in SAL möglich, ein Werkzeug in Funktions- und Interaktionskomponente aufzuteilen. Allerdings kapselt in der Implementierung des WAM-Frameworks die Interaktionskomponente auch die gesamte Benutzeroberfläche des Werkzeugs (die Interaktionskomponente wurde als Fensterklasse implementiert).

Bei der Aufteilung in Funktions- und Interaktionskomponenten bekommt die SAL-eigene Unterscheidung von funktionalen Klassen und Fensterklassen einen gewissen Sinn: Basisklasse der Funktionskomponenten ist die Klasse fcFPBase, eine funktionale Klasse. Die Fensterklasse wclPBase bildet die Basis für Interaktionskomponenten. Da wclPBase eine General Window Class ist, können von dieser Klasse keine Instanzen direkt erzeugt werden – die Klasse ist abstrakt. Man erzeugt also zunächst eine neue Klasse, die vom Typ einer der Standard-Fensterklassen (Form Window Class, Dialog Box Class etc.) ist. Diese Klasse erbt dann von wclPBase.

Die Klasse fcFPBase erbt von fcObservable, die Klasse wclPBase von fcObserver. Diese Basisklassen implementieren den Beobachtermechanismus, der im nächsten Abschnitt vorgestellt wird.

Um die Interaktionskomponenten austauschbar zu halten (bei Beibehaltung der Funktionskomponente), ist es üblich, in der Interaktionskomponente eine Referenz auf die Funktionskomponente zu haben. Da wclPBase keine Referenz auf ein fcFPBase-Objekt halten kann (auch hier wirkt sich der fehlende Referenzmechanismus hinderlich aus), müssen Funktions- und Interaktionskomponente als globale Objekte angelegt werden; die Interaktionskomponente greift dann auf das globale Funktionskomponenten-Objekt zu. Damit sind natürlich das Modularitätskonzept und das

Geheimnisprinzip zerstört, aber gleichzeitig auch ein Merkmal typischer SQLWindows-Programme aufgedeckt, in denen man immer eine große Anzahl globaler Variablen und Objekte finden wird – eine Tatsache, die nicht gerade zur Verständlichkeit und Sicherheit der Programme beiträgt.

Bei der Initialisierung eines Fensters wird diesem die Nachricht SAM_Create gesandt, beim Löschen des Fensters die Nachricht SAM_Destroy. Alle von wcIPBase abgeleiteten Klassen sollten die Funktionen OnCreate() und OnDestroy() überschreiben, anstatt in ihren Message Actions auf diese Nachrichten zu reagieren. wcIPBase ruft in seinen Message Actions die Funktionen OnCreate() und OnDestroy() polymorph auf. Ein Vorteil dieser funktionsbasierten Fensterinitialisierung und -zerstörung ist der einfachere Aufruf der Funktionen in der Basisklasse. Um nämlich eine an das Fenster gesandte Nachricht an die Basisklasse weiterzuleiten, muß die Funktion SalSendClassMessage() bzw. SalSendClassMessageNamed() aufgerufen werden. Beide benötigen als Parameter den Nachrichtennamen und zwei Parameter, die typisch für die Fensternachrichten von Microsoft Windows sind. Hier zeigt sich wieder die enge Bindung von SQLWindows an das Betriebssystem Windows. Mit Hilfe der On...()-Funktionen kann man diese Windows-Nachrichten im Framework kapseln, so daß auf dem Framework aufbauende Programme ein wenig an Portabilität gewinnen[13].

Die Trennung von Funktions- und Interaktionskomponente ist auch in SQLWindows möglich, wobei die Interaktionskomponente auch die Benutzeroberflächenelemente enthält. Aufgrund des fehlenden Referenzkonzeptes müssen Funktions- und Interaktionskomponenten-Objekte global angelegt werden, was eine schwerwiegende Verletzung des Modularitäts- und Geheimnisprinzips darstellt.

5.2.7 Event Mechanism

Der *Beobachtermechanismus* (*Event Mechanism*) ermöglicht es der Funktionskomponente, alle sie benutzenden Interaktionskomponenten über Änderungen zu informieren, ohne daß sie diese kennen muß (im Sinne einer Benutztbeziehung). Dazu müssen sich die Interaktionskomponenten als *Beobachter* (*Observer*) bei der *beobachteten Klasse* (*Observable*) registrieren. Bei einer Zustandsänderung verschickt die beobachtete Klasse eine entsprechende Nachricht an alle registrierten Beobachter. Das folgende Diagramm zeigt die Beziehungen zwischen Observer und Observable und deren Funktionen, die den Beobachtermechanismus implementieren. Die SQLWindows-Implementierung des Beobachtermechanismus ist einfacher als die in [RZ95] beschriebene. Sie kommt im Unterschied zu letzterer ohne Event-Klasse aus.

[13] Gemeint ist hier vor allem die Portierung in eine andere Programmiersprache, die diesen Nachrichtenmechanismus nicht kennt.

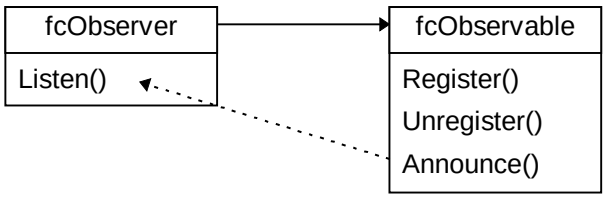


Abbildung 5.13: SQLWindows-Implementierung des Beobachtermechanismus

Mit Register() kann sich ein beobachtendes Objekt bei dem zu beobachtenden Objekt registrieren. Unregister() macht diese Registrierung wieder rückgängig. Die Funktion Announce() erwartet eine Nachricht als Parameter. Diese Nachricht wird an die Funktion Listen() aller registrierten Observer geschickt. Dort werden – abhängig von der konkreten Nachricht – entsprechende Aktionen veranlaßt.

Der Entwurf eines Beobachtermechanismus in SQLWindows zeigt deutlich auf, wie sehr man auf Workaround-Strategien und Spracherweiterungen angewiesen ist, wenn man die Vorteile der objektorientierten Methode auch in SQLWindows nutzen möchte. Aus diesem Grund wird im folgenden die Entwicklungsgeschichte der SQLWindows-Version des Beobachtermechanismus nachgezeichnet. Diese wird anhand mehrerer Programmbeispiele nachgezeichnet. Ein Interaktionsdiagramm für den Beobachtermechanismus soll beim Verstehen der Programmbeispiele helfen. Die einzelnen Phasen im Interaktionsdiagramm sind mit Nummern ❶-❹ gekennzeichnet; eine entsprechende Kennzeichnung findet man in den Programmbeispielen wieder. Hier zunächst das Interaktionsdiagramm:

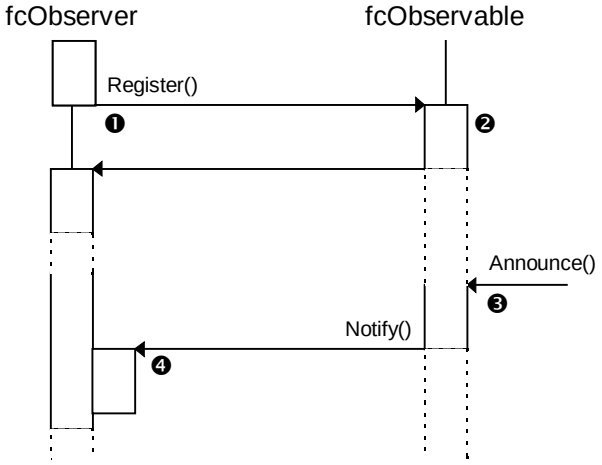


Abbildung 5.14: Beobachtermechanismus – Registrierung und Benachrichtigung

Die ursprüngliche Idee war, der Klasse fcObservable eine Liste von fcObserver-Objekten (bzw. Zeigern auf diese) als Attribut mitzugeben. In der Funktion Register() sollte dann das zu registrierende fcObserver-Objekt in diesem Array gespeichert werden (vgl. Beispielprogramm FC-TEST.APP):

```
♦Function: Register ❷
    ◊ Description: Registers an observer to this observable
    ♦Returns
        ◊ ! TRUE if the observer has been registered successfully, FALSE otherwise
        ◊ Boolean: bOk
    ♦Parameters
        ◊ fcObserver: Observer
    ◊ Static Variables
    ♦Local variables
        ◊ Boolean: bOk
        ◊ Number: nNumObservers
    ♦Actions
        ♦If SalArrayIsEmpty(aObservers)
            ◊ ! This is the first observer to be registered to this observable
            ◊ Set aObservers[0] = Observer     💣
        ♦Else
            ◊ ! Retrieve the number of observers already registered
            ◊ Set bOk = SalArrayGetUpperBound(aObservers, 1, nNumObservers)
            ♦If bOk = FALSE
                ◊ Return FALSE
            ♦Else
                ◊ Set aObservers[nNumObservers + 1] = Observer     💣
        ◊ Return TRUE
```

SAL hat ein sehr dynamisches Array-Konzept, aber leider können nur Basis-Datentypen als Array-Elemente gespeichert werden. Deshalb resultiert die Zuweisung eines fcObserver-Objektes zu einem Array-Element (vgl. die mit 💣 gekennzeichneten Zeilen) in einem Compilerfehler.

Als erste Lösung dieses Problems wurde die Speicherung von Fensterbezügen gewählt. Das mit SQLWindows mitgelieferte Beispielprogramm `OOP.APP`, das in die objektorientierte Programmierung einführen soll, verwendet den Fensterbezug als Objektreferenz selbst für eine allgemeine Mengenklasse. Der Nachteil dieser Methode ist klar: Nur Fensterobjekte können auf diese Art und Weise gespeichert werden. Ein weiterer Nachteil entsteht durch die Typlosigkeit der gespeicherten „Objektreferenzen“: Ein Fensterbezug gibt keinen Aufschluß darüber, von welchem Typ das mit diesem Fensterbezug referenzierte Fensterobjekt ist. Somit hat man nie die Gewißheit, ob eine Funktion, die man an einem (nur über den Fensterbezug referenzierten) Objekt aufruft, auch tatsächlich in der dem Objekt zugrundeliegenden Klasse definiert ist.

In einem neuen Entwurf sollte eine Referenz auf das fcObserver-Objekt an die Register()-Funktion der Klasse fcObservable übergeben werden. Dieses fcObserver-Objekt konnte man nach seinem Fensterbezug fragen und diesen intern in einem Array speichern. Da der Fensterbezug nun nachweislich zu einem fcObserver-Objekt gehörte, hätte man ihn „typsicher“ verwenden können. Dieser Entwurf hätte von seiten der Klasse fcObservable keine Probleme bereitet, da Referenzen auf benutzerdefinierte Variablen als Parameter einer Funktion angegeben werden können:

```
♦Functional Class: fcObservable
    ...
    ♦Functions
```

♦Function: Register ❷
◊ Description: Registers an observer to this observable.
♦Returns
◊ ! TRUE if the observer has been registered successfully, FALSE otherwise
◊ Boolean: bOk
♦Parameters
◊ fcObserver: udvObserver
...

Die Probleme ergaben sich erst bei den Beobachtern – Instanzen der Klasse wcIPBase, einem Erben der Klasse fcObserver. wcIPBase sollte eine Funktion RegisterToFP() haben, die als Parameter eine (Referenz auf eine) Funktionskomponente erhält und die Interaktionskomponente bei dieser registriert. Dazu muß RegisterToFP() die Funktion Register() der übergebenen Funktionskomponente aufrufen und sich selbst als Parameter übergeben (❶). SAL hält aber keinen entsprechenden sprachlichen Ausdruck für eine Selbstreferenz bereit, wie ihn z.B. die Programmiersprache C++ mit dem Schlüsselwort this bietet. Es gibt die Systemvariable MyValue, die aber nur den Fensterbezug der aktuellen Klasseninstanz hält. Ein Beispiel:

♦Data Field: dfName
♦Message Actions
♦On SAM_Create
◊ Set MyValue = 'Holger'

Ein dem Schlüsselwort this ähnliches Sprachkonstrukt fehlt in SAL. So konnte diese typsichere Lösung nicht implementiert werden.

Erst mit der Einführung des SQLWindows Object System (SOS; vgl. Kapitel 6) war es möglich, Objektreferenzen von der beobachtenden an die beobachtete Klasse zu übergeben. Streng genommen sind auch mit dem SOS keine Referenzen auf Objekte möglich. Man kann aber ein Objekt an das SOS übergeben und dort speichern. Unter Angabe von Objekt-ID und Objekttyp kann dieses Objekt (mit dem Zustand, den es zur Zeit der Speicherung innehatte) wieder ausgelesen werden. Diesen Umstand macht sich die vorliegende Implementierung des Beobachtermechanismus zunutze. Die beobachtende Klasse speichert sich selbst im SOS und übergibt beim Aufruf der Funktion Register() (❶) ihren Objekttyp und ihre Objekt-ID. Die beobachtete Klasse kann nun mit Hilfe dieser Angaben das Objekt aus dem SOS holen. Da der Fensterbezug mit dem Objekt gespeichert wird, kann sie diesen intern in einem Array speichern. Die Zustandsbehaftetheit des SOS (siehe Abschnitt 6.3.3) stellt für den Beobachtermechanismus kein Problem dar, da der Fensterbezug ein Attribut ist, das seinen Wert während der gesamten Lebenszeit des Beobachter-Objektes beibehält.

Die beiden Klassen fcObserver und fcObservable werden im folgenden in voller Länge angegeben, da sie einige interessante Details enthalten und einen guten Einblick in das WAM-Framework und die SQLWindows-Programmierung geben. Die interessanten Stellen sind im Quelltext mit Nummern versehen und werden anschließend erklärt.

♦Functional Class: fcObserver
◊ Description: Observes a fcObservable object. If the state of the fcObservable object changes, it sends a message to all registered observers by calling their Listen() function with the message as parameter.

Messages are global constant numbers with a value greater than SAM_User.
Abstract class.
fcObserver inherits from fcAtomizable (defined in SOS.APL), which itself inherits from fcObject.

♦Derived From
◊ Class: fcAtomizable
◊ Class Variables
♦Instance Variables
◊ ! This observer's window handle. It can be obtained by calling the
! GetWindowHandle() function.
◊ Window Handle: m_hWndThis
♦Functions
◊ ! Internal ①

♦Function: WriteAttributes
◊ Description: Writes the observer's attributes to the SOS.
♦Returns
◊ ! TRUE if successful, else FALSE:
◊ Boolean:
◊ Parameters
◊ Static Variables
♦Local variables
◊ Boolean: bOk
♦Actions
◊ Set bOk = SOSWriteWindowHandle ②
(GetObjectId(),'m_hWndThis',m_hWndThis)
◊ Return bOk
♦Function: ReadAttributes
◊ Description: Reads the observer's attributes from the SOS.
♦Returns
◊ ! TRUE if successful, else FALSE:
◊ Boolean:
◊ Parameters
◊ Static Variables
♦Local variables
◊ Boolean: bOk
♦Actions
◊ Set bOk = SOSReadWindowHandle
(GetObjectId(), 'm_hWndThis',m_hWndThis)
◊ Return bOk
◊ ! Public
♦Function: Reset
◊ Description: Resets the object's attributes.
◊ Returns
◊ Parameters
◊ Static Variables
◊ Local variables
♦Actions

- ◊ Set m_hWndThis = hWndNULL
- ♦ Function: SetWindowHandle
  - ◊ Description: Sets the observer's window handle
  - ◊ Returns
  - ♦ Parameters
    - ◊ Window Handle: hWndThis
  - ◊ Static Variables
  - ◊ Local variables
  - ♦ Actions
    - ◊ Set m_hWndThis = hWndThis
- ♦ Function: GetWindowHandle
  - ◊ Description: Returns the observer's window handle
  - ♦ Returns
    - ◊ Window Handle:
  - ◊ Parameters
  - ◊ Static Variables
  - ◊ Local variables
  - ♦ Actions
    - ◊ Return m_hWndThis
- ◊ ! To be overridden [Public]
- ♦ Function: Listen ❹
  - ♦ Description: This function is called by a fcObservable object where this observer is registered in case of a state change.
  - ◊ Returns
  - ♦ Parameters
    - ◊ ! Message describing the fcObservable object's state change
    - ◊ Number: nMessage
  - ◊ Static Variables
  - ◊ Local variables
  - ♦ Actions
    - ◊ Call FunctionNotOverridden(GetObjectType(), 'Listen') ③

♦ Functional Class: fcObservable

◊ Description: fcObservable objects can be observed by fcObserver objects. If the state of the fcObservable object changes, all registered observers can be notified by calling the Announce() function with a message as parameter. Messages are global constant numbers with a value greater than SAM_User. Abstract class.

- ♦ Derived From
  - ◊ Class: fcAtomizable
- ◊ Class Variables
- ♦ Instance Variables
  - ◊ ! Dynamic array of observer window handles
  - ◊ Window Handle: m_ahWndObservers[*] ④
  - ◊ ! The number of connected observers
  - ◊ Number: m_nNumObservers
- ♦ Functions

◊ ! Internal
♦ Function: Announce ❸
◊ Description: Notifies all registered observers of a status change
◊ Returns
♦ Parameters
◊ ! Message describing the status change.
! Messages are global constant numbers with a value greater than
! SAM_User.
◊ Number: nMessage
◊ Static Variables
♦ Local variables
◊ Number: nIndex

♦ Actions
◊ Set nIndex = 0
♦ While nIndex < m_nNumObservers
◊ ! Call the Listen() function of every registered observer
◊ Call m_ahWndObservers[nIndex].fcObserver..Listen(nMessage) ⑤
◊ Set nIndex = nIndex + 1
◊ ! Public
♦ Function: Register ❷
◊ Description: Registers an observer to this observable
♦ Returns
◊ ! TRUE if the observer has been registered successfully, FALSE otherwise
◊ ! Boolean:
♦ Parameters
◊ ! Object type of the observer to be registered
◊ String: strObjectType
◊ ! Object Id of the observer to be registered
◊ Number: nObjectId
◊ Static Variables
♦ Local variables
◊ fcObserver: udvObserver
◊ Window Handle: hWndObserver
♦ Actions
◊ ! Read object from SOS and retrieve its window handle
◊ Call udvObserver.Init() ⑥
◊ ! Set correct type and ID for the local observer object
◊ Call udvObserver.SetObjectType(strObjectType) ⑦
◊ Call udvObserver.SetObjectId(nObjectId)
♦ If NOT udvObserver.ReadObject()
◊ Call udvObserver.Done() ⑥
◊ Return FALSE
◊ Set hWndObserver = udvObserver.GetWindowHandle()
◊ ! Register observer
♦ If SalArrayIsEmpty(m_ahWndObservers)
◊ ! This is the first observer to be registered to this observable

```
                ◊ Set m_ahWndObservers[0] = hWndObserver
                ◊ Set m_nNumObservers = 1
            ♦ Else
                ◊ Set m_nNumObservers = m_nNumObservers + 1
                ◊ Set m_ahWndObservers[m_nNumObservers] = hWndObserver
            ◊ Call udvObserver.Done()     ⑥
            ◊ Return TRUE
♦ Function: Unregister
    ◊ Description: Unregisters an observer from this observable
    ♦ Returns
        ◊ ! TRUE if the observer has been unregistered successfully, FALSE
          ! otherwise
        ◊ Boolean:

    ♦ Parameters
        ◊ ! Object type of the observer to be registered
        ◊ String: strObjectType
        ◊ ! Object Id of the observer to be unregistered
        ◊ Number: nObjectId
    ◊ Static Variables
    ♦ Local variables
        ◊ fcObserver: udvObserver
        ◊ Window Handle: hWndObserver
        ◊ Number: nIndex
        ◊ Boolean: bFound
    ♦ Actions
        ◊ ! Read object from SOS and retrieve its window handle
        ◊ Call udvObserver.Init()
        ◊ ! Set correct type and ID for the local observer object
        ◊ Call udvObserver.SetObjectType(strObjectType)
        ◊ Call udvObserver.SetObjectId(nObjectId)
        ♦ If NOT udvObserver.ReadObject()
            ◊ Call udvObserver.Done()
            ◊ Return FALSE
        ◊ Set hWndObserver = udvObserver.GetWindowHandle()
        ◊ ! Register observer
        ♦ If NOT SalArrayIsEmpty(m_ahWndObservers)
            ◊ ! Search the observer window handle array for the observer to be
              ! unregistered
            ◊ Set nIndex = 0
            ◊ Set bFound = FALSE
            ♦ While nIndex < m_nNumObservers
                ♦ If m_ahWndObservers[nIndex] = hWndObserver
                    ◊ Set bFound = TRUE
                ♦ If bFound = TRUE AND nIndex < m_nNumObservers - 1
                    ◊ ! Move all following observers to preceding array
                      ! position to 'close the gap' created by removing the
                      ! unregistered observer
```

```
                    ◊ Set m_ahWndObservers[nIndex] =
                      m_ahWndObservers[nIndex + 1]
                ◊ Set nIndex = nIndex + 1
        ♦ If bFound = TRUE
                ◊ ! Erase the last observer's duplicate
                ◊ Set m_ahWndObservers[m_nNumObservers] = hWndNULL
    ◊ Call udvObserver.Done()
    ◊ Return TRUE
```

① Der Kommentar ! Internal schreibt vor, daß die nachfolgenden Funktionen nur innerhalb der Klasse benutzt werden dürfen, in der sie definiert sind, sowie innerhalb der von dieser Klasse abgeleiteten Klassen. Kommentare dieser Art simulieren einen Zugriffsschutz, der in SQLWindows fehlt. Der Zugriffsschutz ist in Abschnitt 5.3.3.3 beschrieben.

② Mit Hilfe der Funktion SOSWriteWindowHandle() wird ein Fensterbezug in das SOS geschrieben. Die Klasse fcObserver ist ein gutes Beispiel für die Benutzung des SOS und der Klasse fcAtomizable. Ein ausführlicheres Beispiel findet man in Abschnitt 6.5.

③ Die globale Funktion FunctionNotOverridden() sollte immer in Funktionen aufgerufen werden, die in abgeleiteten Klassen überschrieben werden müssen („virtuelle Methoden“ im C++ - Sprachgebrauch). Eine Ausnahme bilden Funktionen, deren überschriebene Versionen von den abgeleiteten Klassen weiterhin aufgerufen werden sollen, wie z.B. die Funktion Init() der Klasse fcObject (siehe Abschnitt 5.2.1 bzw. A.2.7). Da die Funktion auch in ihrer Basisklassenversion aufgerufen wird, würde in diesem Falle fälschlicherweise FunctionNotOverridden() aufgerufen.

SQLWindows bietet keine Möglichkeit, virtuelle Methoden zu definieren. Auch abstrakte Klassen sind – mit Ausnahme der General Window Class – nicht möglich. Strenge Dokumentationsrichtlinien (vgl. Abschnitt 5.3.3) und die Funktion FunctionNotOverridden() sollen eine Verwendung virtueller Methoden erlauben, die so sicher wie möglich ist.

④ Die Klasse fcObservable hält mit m_ahWndObservers[*] ein dynamisches Array von Fensterbezügen. Die Verwaltung von Arrays ist eine der Stärken von SQLWindows. Arrays können dynamisch oder statisch mit beliebigen skalaren Unter- und Obergrenzen sein. Es steht eine Reihe von Systemfunktionen zur Verfügung, mit Hilfe derer Arrays komfortabel sondiert und manipuliert werden können. Die *Visual Toolchest*, eine Klassen- und Funktionsbibliothek für SQLWindows, die zum Lieferumfang des Entwicklungssystems gehört, enthält weitere Array-Funktionen.

In SAL-Arrays können neben Werten der Standard-Datentypen auch benutzerdefinierte Variablen gespeichert werden, jedoch keine Referenzen auf benutzerdefinierte Variablen. Das schränkt die Mächtigkeit dieses Sprachmittels erheblich ein.

⑤ Die Funktion Announce() verwendet beim Aufruf der Funktion Listen() bei allen angemeldeten Beobachtern einen voll qualifizierten polymorphen Funktionsaufruf:

```
m_ahWndObservers[nIndex].fcObserver..Listen(nMessage)
```

Der Funktionsaufruf ist voll qualifiziert (sowohl der Fensterbezug (m_ahWndObservers[nIndex]) als auch der Klassenname (fcObserver) sind zur Qualifikation aufgeführt).

⑥ Die benutzerdefinierte Variable udvObserver ist eine lokale Variable der Funktion Register(). Somit ist die Funktion selbst für die korrekte Initialisierung und das Aufräumen nach Benutzung von udvObserver zuständig. Sie kommt dieser Pflicht durch Aufruf von udvObserver.Init() vor der Benutzung und udvObserver.Done() nach dem letzten Zugriff auf udvObserver nach.

⑦ Das SOS hat eine eingebaute Typprüfung. Um ein Objekt aus dem SOS auslesen zu können, müssen Objekttyp und Objekt-ID des Zielobjektes mit denen des im SOS gespeicherten Objektes identisch sein. Deshalb wird der Typ und die ID von udvObserver auf die an die Funktion Register() übergebenen Werte gesetzt. Erst dann führt ein Aufruf von udvObserver.ReadObject() zum gewünschten Erfolg.

Diese Implementierung des Beobachtermechanismus nutzt nicht die Message Actions der Fensterklassen zum Versenden der Nachrichten, sondern ruft bei den Empfängern der Nachricht die Funktion Listen() auf. Dadurch wird eine weitestgehende Unabhängigkeit vom Betriebssystem Microsoft Windows erreicht, auf dessen Nachrichtensystem die Message Actions von SQLWindows aufbauen. So fällt es leichter, SQLWindows-Programme in andere Programmiersprachen zu portieren, die i.d.R. keine Nachrichtenbehandlung als sprachliches Konzept aufweisen. Das folgende Beispiel einer Listen()-Funktion stammt aus PSP_00.APP:

- Function: Listen
  - Description: This function is called by a fcObservable object where this observer is registered in case of a state change.
  - Returns
  - Parameters
    - ! Message describing the fcObservable object's state change
    - Number: nMessage
  - Static Variables
  - Local variables
  - Actions
    - Select Case nMessage
      - Case INSTRUCTOR_SELECTED
        - Call DisplaySelectedInstructor()

Das WAM-Framework implementiert einen einfachen Beobachtermechanismus, der ohne eigene Event-Klassen auskommt und sich nicht auf die Message Actions stützt. Zur typsicheren Objekt(referenz)übergabe von der beobachtenden an die beobachtete Klasse wird das SQLWindows Object System (SOS) eingesetzt.

5.2.8 IP/FP Plug In

Die Komposition von Werkzeugen aus Sub-Werkzeugen (vgl. Abschnitt 5.2.5) wird dahingehend verfeinert, daß man unter Berücksichtigung des Entwurfsmusters „Separation of Powers“ (vgl. Abschnitt 5.2.6) Sub-Interaktionskomponenten in Kontext-Interaktionskomponenten steckt, und ebenso Sub-Funktionskomponenten in Kontext-Funktionskomponenten.

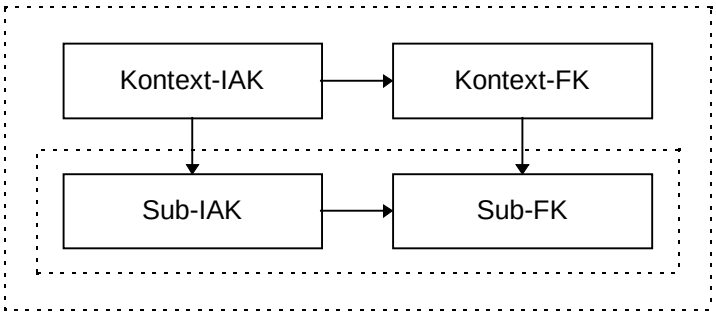


Abbildung 5.15: Sub-Funktions- und -Interaktionskomponenten

Bei der Realisierung in SAL verhinderte wieder einmal das Fehlen eines Referenzmechanismus eine saubere und sichere Implementierung. Alle Funktionskomponenten werden als globale benutzerdefinierte Variablen angelegt. Die Sub-Interaktionskomponenten existieren als Kindfenster des Hauptfensters, das gleichzeitig die Kontext-Interaktionskomponente ist. Während die Interaktionskomponenten frei auf ihre korrespondierende Funktionskomponente zugreifen können, bedienen sich die Funktionskomponenten des Beobachtermechanismus, um die angemeldeten Beobachter (i.d.R. Interaktionskomponenten) über Änderungen zu informieren. Es war geplant, den Beobachtermechanismus auch für die Kommunikation zwischen Sub-Werkzeug und Kontext-Werkzeug einzusetzen. Dazu hätte aber die Klasse wcSubIPBase sowohl von fcObserver (in ihrer Funktion als Interaktionskomponente) als auch von fcObservable (zur Benachrichtigung der Kontext-Interaktionskomponente) erben müssen. Da sowohl fcObserver als auch fcObservable von fcObject erben, SAL aber keine diamantenförmige Vererbung unterstützt (vgl. Abschnitt 4.3.8), käme es zu einem Basisklassen-Konflikt. Hinzu kommt, daß der Beobachtermechanismus intern mit einem Fensterbezug zur Objektidentifikation arbeitet. Funktionskomponenten sind aber Instanzen funktionaler Klassen und haben keinen Fensterbezug. Die Kommunikation zwischen Kontext- und Sub-Funktionskomponente konnte demnach auch nicht über den Beobachtermechanismus erfolgen. Deshalb besitzen die Basisklassen wcSubIPBase und fcSubFPBase gegenüber ihren Basisklassen wcIPBase bzw. fcFPBase lediglich eine zusätzliche (virtuelle) Funktion NotifyContextIP() bzw. NotifyContextFP(). Über diese zu überschreibenden Funktionen soll die Kommunikation mit der Kontext-Interaktions- bzw. Funktionskomponente geschehen, indem deren Listen()-Funktion aufgerufen wird. Das Kontext-Interaktions- bzw. Funktionskomponenten-Objekt wird direkt über die entsprechende globale Variable referenziert. Als Beispiel sei ein Ausschnitt aus der Klasse wcInstructorListerSubIP des Programms PSP_00.APP dienen:

- Function: NotifyContextIP
  - Description: Notifies the context IP of a status change.
    This function should call the Listen() function of the context IP.
  - Returns
  - Parameters
    - ! Message describing the status change.
    - ! Messages are global constant numbers with a value greater than SAM_User.
    - Number: nMessage
  - Static Variables
  - Local variables

♦ Actions
◊ Call frmMain.Listen(nMessage)

frmMain ist das globale Interaktionskomponenten-Objekt, in diesem Falle das Hauptfenster. Wie man an diesem Beispiel bereits erkennen kann, hat das Fehlen eines Referenzmechanismus und die damit verbundene Verwendung globaler Variablen einen großen Einfluß auf die Modularität von SQLWindows-Programmen: Die Klasse wcInstructorListerSubIP ist auf die Existenz eines globalen Objektes mit einem bestimmten Namen angewiesen. Wird der Name geändert, so kann das weitreichende Änderungen im Programm nach sich ziehen.

Die Implementierung der Sub-Interaktions- bzw. -Funktionskomponenten wird negativ durch das Fehlen eines Referenzmechanismus beeinflußt. Neben der als sehr problematisch zu betrachtenden Notwendigkeit der Globalität der Funktions- und Interaktionsobjekte kann die Kommunikation zwischen Kontext- und Sub-Interaktions- bzw. Funktionskomponenten nicht über den Beobachtermechanismus erfolgen, sondern nur durch direkten Aufruf der globalen Objekte.

5.2.9 Material Container

Ein Material-Container gruppiert und koordiniert voneinander abhängige Materialien und garantiert, daß Integritätsbedingungen und Beschränkungen, die zwischen diesen Materialien existieren, immer erfüllt sind.

Verschiedene Materialien hängen oftmals voneinander ab oder sind logisch miteinander verbunden. Diese Abhängigkeiten sollten nicht in einem Werkzeug modelliert werden, da sie eventuell auch in einem anderen Kontext benötigt werden könnten. Man braucht also eine unabhängige Instanz, die diese Abhängigkeiten modelliert und die universell eingesetzt werden kann. Ein Material-Container ist eine solche Instanz.

Diese Aufgabe ist sehr speziell und hängt vom konkreten Anwendungskontext ab. Es konnten (auch mangels konkreter Beispiele) keine sinnvollen Abstraktionen gefunden werden, die man in einer Klasse des Frameworks hätte kapseln können.

5.2.10 Tool Coordinator

Ein Werkzeug-Koordinator informiert ein Werkzeug über von außen herbeigeführte Änderungen an den Materialien, die dieses Werkzeug bearbeitet.

Wenn eine der in Abschnitt 5.2.9 angesprochenen Integritätsbedingungen und Beschränkungen von Materialien zu einer Änderung des Materialzustands führt, dann müssen der Zustand aller Werkzeuge, die dieses Material bearbeiten, sowie die visuelle Präsentation des Materials aktualisiert werden. Die Aufgabe der Benachrichtigung bei einer Änderung des Materialzustands übernimmt ein Werkzeug-Koordinator, von dem es je einen für jeden Material-Container gibt. Der Werkzeug-Koordinator weiß, welche Werkzeuge (genauer: welche Funktionskomponenten von Werkzeugen) auf den in „seinem“ Material-Container enthaltenen Materialien arbeiten, so daß er diese bei einer Änderung des Materialzustands gezielt benachrichtigen kann.

Diese Aufgabe ist sehr speziell und zudem abhängig von den Integritätsbedingungen und Beschränkungen der Materialien sowie der Art, in der die Materialien gespeichert sind. So konnte für dieses Entwurfsmuster keine Abstraktion gefunden werden, die in das Framework hätte aufgenommen werden können.

5.2.11 Material Administration

Aufgabe einer Materialverwaltung ist das Lesen und Speichern von Materialien, die Zugriffskontrolle auf Originale und Kopien sowie die Gruppierung von Materialien in Material-Containern (vgl. Abschnitt 5.2.9).

Die Basisklasse fcMaterialAdministrationBase des WAM-Frameworks erfüllt nur einen Teil dieser Aufgaben, namentlich das Lesen und Speichern der Materialien. Die Materialverwaltung bezieht die Materialien i.d.R. aus einer Datenbank oder aus einem Material-Container (vgl. Abschnitt 5.2.9).

- ♦ Functional Class: fcMaterialAdministrationBase
  - ◊ Description: Material Administration base class
  - ♦ Derived From
    - ◊ Class: fcObject
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ◊ ! To be overridden [Public]
    - ♦ Function: GetFirst
    - ♦ Function: GetLast
    - ♦ Function: GetNext
    - ♦ Function: GetPrevious
    - ♦ Function: MakePersistent

Mit Hilfe der Get...()-Funktionen kann durch die verfügbaren Materialien navigiert werden. In einer ersten Implementierung stellte fcMaterialAdministrationBase immer ein aktuelles Material in Form einer globalen Variablen bereit, ähnlich dem Konzept der Kommunikationsrekords und Aktualitätsanzeiger in Netzwerk- und hierarchischen Datenbanken (vgl. [LS87]). Mit der Einführung des SQLWindows Object System (SOS; vgl. Kapitel 6) konnten die Funktionen das gesuchte Material im SOS speichern und eine Objekt-Id (oder TANDEM_ERROR im Fehlerfall) liefern, über die das Material aus dem SOS ausgelesen werden kann.

Um nicht bei jeder Objektanforderung das Material in das SOS speichern zu müssen, kann sich die Materialverwaltung der Funktion GetModify() des Materials bedienen, um zu überprüfen, ob das Material geändert wurde. Sind keine Änderungen erfolgt, so muß das Material u.U. nicht noch einmal in das SOS gespeichert werden.

Die Funktion MakePersistent() macht das aktuelle Material mit allen getätigten Änderungen persistent.

Das WAM-Framework besitzt einen gegenüber dem Entwurfsmuster „Material Administration“ eingeschränkten Begriff der Materialverwaltung. Diese ist lediglich für das Lesen und Speichern der Materialien, i.d.R. aus bzw. in eine Datenbank, zuständig. Eine Übergabe der Materialobjekte an das Werkzeug per Referenz wurde erst mit der Einführung des SOS möglich.

5.2.12 Environment

Ein Umgebungsobjekt soll als Aufsetzpunkt für das gesamte Programmsystem dienen. Es zeigt die verfügbaren Werkzeuge und Materialien, erzeugt zu jedem Material-Container einen Werkzeug-Koordinator und trägt Sorge für die einwandfreie Initialisierung des Systems, der angeschlossenen Datenbanken etc.

Da es in dieser Implementierung des WAM-Frameworks keine Material-Container gibt, entfällt die oben genannte zweite Aufgabe der Umgebung. Lediglich die dritte Aufgabe könnte von einer Umgebung übernommen werden. Die Klasse clsProgram des E.I.S. Application Builder enthält Funktionen zur Abfrage des Programmnamens und der Kommandozeilenparameter (vgl. [EIS96]). All diese Aufgaben sind in die Klasse fcEnvironmentBase des WAM-Frameworks eingeflossen.

- ♦ Functional Class: fcEnvironmentBase
  - ◊ Description: Environment base class
  - ♦ Derived From
    - ◊ Class: fcObject
  - ♦ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ◊ ! Public
    - ♦ Function: Init
    - ♦ Function: Reset
    - ♦ Function: Done
    - ♦ Function: GetProgramName
    - ♦ Function: GetModuleName
    - ♦ Function: GetPath
    - ♦ Function: GetArgument
    - ◊ ! To be overridden
    - ♦ Function: ConnectToDB
    - ♦ Function: DisconnectFromDB

Die Attribute der Klasse sind als Klassenvariablen deklariert, d.h. alle Instanzen der Klasse benutzen dieselben Variablen. Das ist sinnvoll, da zur Laufzeit nur eine einzige Umgebungsinstanz existieren soll.

Die Klasse fcEnvironmentBase basiert auf einem Umgebungsbegriff, der nur zu einem geringen Teil dem in [RZ95] beschriebenen Umgebungs-Objekt entspricht – namentlich in seiner Aufgabe, die Datenbankdienste zu initialisieren. Die übrigen Aufgaben beziehen sich auf das Verfügbarma-

chen von allgemeinen Programminformationen wie Programmname, Ausführungspfad und Kommandozeilenargumente.

5.3 Benutzung des Frameworks

Ein Framework ist keine Klassenbibliothek – dieser Tatsache sollte sich der Benutzer des WAM-Frameworks immer bewußt sein. Der folgende Abschnitt zeigt die Unterschiede zwischen einem Framework und einer Klassenbibliothek auf. Der Einsatz des Frameworks in eigenen SQLWindows-Programmen wird anhand eines Beispiels in Abschnitt 5.3.2 ausführlich erläutert.

Zur Benutzung eines Frameworks kann auch dessen Erweiterung gehören. Damit dies in einer konsistenten Art und Weise geschieht, zeigt Abschnitt 5.3.3 die Programmier- und Dokumentationsrichtlinien auf, auf deren Grundlage das Framework implementiert wurde.

5.3.1 Klassenbibliothek vs. Framework

In diesem Kapitel ist die ganze Zeit von einem Framework die Rede. Was genau macht denn ein solches Framework aus, und wie unterscheidet es sich von einer Klassenbibliothek, als die man die oben aufgeführten Klassen auch betrachten könnte?

Eine Definition des Begriffs „Framework“ stammt von Cotter und Potel ([CP95]):

> „Ein *Framework* verkörpert einen allgemeinen Entwurf, bestehend aus einer Menge kooperierender Klassen, die an die konkreten Probleme innerhalb einer Domäne angepaßt werden können.“

Laut Taligent ([Tal95]) sind Frameworks Aggregate von Klassen, in gleicher Weise, wie Klassen Aggregate von Funktionen und Daten sind. Darüber hinaus haben Frameworks jedoch eine Architektur, d.h. sie liefern eine Struktur, die einen Entwurf widerspiegelt. Ein wichtiger Begriff in obiger Definition ist der Domänenbegriff. Eine Domäne sei hier in Anlehnung an [FKP96] definiert als eine Menge von Merkmalen, welche eine Problemklasse, für die Softwarelösungen gesucht werden, umfassend und genau beschreibt.

Von Klassenbibliotheken zu Frameworks

Wiederverwendbare Klassenbibliotheken sind zwar ein probates Mittel zur objektorientierten Code-Wiederverwendung, aber sie weisen einige Beschränkungen auf:

- Komplexitätsprobleme: Große Bibliotheken mit untereinander stark verknüpften Klassen sind nur schwer zu durchschauen und zu erlernen.
- Kein inhärenter Kontrollfluß: Der Kontrollfluß liegt in der Verantwortung des Applikationsentwicklers als Benutzer der Klassenbibliothek.
- Keine Design-Wiederverwendung: Zwei Programmierer könnten dieselbe Klassenbibliothek für die Entwicklung zweier Programme einsetzen, welche exakt dieselbe Funktionalität haben, deren Entwürfe jedoch völlig unterschiedlich sein können.

Frameworks können diesen Problemen begegnen. Ihre innere Architektur ist normalerweise so aufgebaut, daß nie unüberschaubar viele Klassen miteinander verknüpft sind. Komplexe Frame-

works lassen sich i.d.R. in Einheiten unterteilen, die über klar definierte Schnittstellen miteinander verbunden sind. So bleiben auch große Frameworks leicht überschaubar. Der Kontrollfluß liegt zum großen Teil im Framework selbst. Nach dem Motto „Don't call us, we'll call you" ruft das Framework Funktionen auf, die vom Framework-Benutzer über Vererbung und Überschreiben von Funktionen dem Framework hinzugefügt wurden. Frameworks beinhalten neben dem Verhalten auch ein Protokoll, d.h. Regeln, die Verhaltenskombinationen beschreiben und Verantwortlichkeiten von Framework und Applikationsentwickler festlegen, wie z.B. die Festlegung, welche Teile des Frameworks vom Applikationsentwickler überschrieben und ergänzt werden müssen.

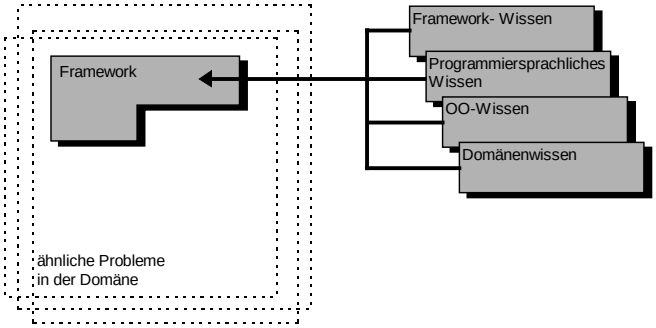


Abbildung 5.16: Ein Framework ist mehr als eine Klassenbibliothek

Der größte Vorteil eines Frameworks ist jedoch, daß es einen Entwurf repräsentiert, der mit der Benutzung des Frameworks wiederverwendet wird, während Klassenbibliotheken lediglich Code-Wiederverwendung ermöglichen. Abbildung 5.16 (nach [Tal95]) zeigt, welches Wissen in ein Framework eingeht, um eine möglichst allgemeine Lösung zu erreichen.

Haben wir es wirklich mit einem Framework zu tun?

Die Antwort auf diese Frage lautet: Ja. Betrachten wir z.B. die Klasse fcObject. In deren Init()-Funktion wird polymorph die Funktion Reset() aufgerufen, die in abgeleiteten Klassen überschrieben werden kann. Dieses Verhalten ist innerhalb des Frameworks beschrieben, wirkt sich aber auf Klassen aus, die vom Benutzer des Frameworks erstellt werden. Ähnlich verhält es sich mit dem Beobachtermechanismus, der in den Klassen fcObserver und fcObservable implementiert ist. Durch Aufruf der Funktion Announce(), beispielsweise in einer Funktionskomponente, werden Objekte benachrichtigt, die Instanzen von Klassen sind, die der Applikationsprogrammierer erstellt hat. Diese benachrichtigenden und benachrichtigten Klassen sind jedoch von Klassen des Frameworks abgeleitet, die wiederum ein bestimmtes Kooperationsprotokoll besitzen, das im Framework verankert ist und an die abgeleiteten Klassen weitergegeben wird. Die Klassen erben nicht nur ein Verhalten, sondern eine spezifische Beziehung zwischen Objekten dieser Klassen. Eben dieses über die reine Verhaltensbeschreibung hinausgehende Wissen ist es, was ein Framework ausmacht.

5.3.2 Der Einsatz des WAM-Frameworks am Beispiel des „Break Supervision Planner“

Der Einsatz des WAM-Frameworks soll anhand eines Beispielprogramms beschrieben werden. Die Grundlage für dieses Beispielprogramm bildete der Pausenaufsichtsplaner der OO-DACH-Gruppe (vgl. [Syl96]), der in die Welt von „Star Trek – The Next Generation“ versetzt wurde und hier als „Break Supervision Planner“ für die Sternenflotten-Akademie seinen Dienst tut.

5.3.2.1 Programmbeschreibung

Abbildung 5.17: Break Supervision Planner

Mit Hilfe des Break Supervision Planner können Pausenaufsichten verwaltet werden. Jeder Instruktor der Sternenflotten-Akademie hat eine gewisse Anzahl an Pausen, die zu beaufsichtigen er verpflichtet ist.

Das Programm zeigt links eine Liste der verfügbaren Instruktoren. Wählt man einen Instruktor aus, so werden in der Statuszeile am unteren Fensterrand folgende Informationen angezeigt:

- Name des Instruktors
- Anzahl der insgesamt von ihm zu beaufsichtigenden Pausen
- Anzahl der Pausen, für die er zur Beaufsichtigung eingeteilt wurde

Hat man einen Instruktor ausgewählt, so kann man ihn per Drag-and-Drop auf den Pausenaufsichtsplan ziehen. Alternativ kann man auch das Namenskürzel des Instruktors in den Pausenaufsichtsplan eintragen (ungültige Namenskürzel werden vom Programm abgewiesen). In beiden Fällen wird geprüft, ob der Instruktor eine weitere Pause beaufsichtigen kann.

Ein Klick auf die Schaltfläche „Clear Plan“ löscht alle Einträge (d.h. alle Instruktoren) aus dem Pausenaufsichtsplan. Die Schaltfläche „Print Plan“ dient dem Ausdrucken des Pausenaufsichtsplanes. Mit der Schaltfläche „Remove“ kann ein ausgewählter Instruktor aus dem Plan entfernt werden.

5.3.2.2 Datenbankmodell

Der Break Supervision Planner bezieht seine Daten aus einer einfachen relationalen Datenbank ACADEMY mit drei Tabellen, die in der folgenden Abbildung dargestellt ist:

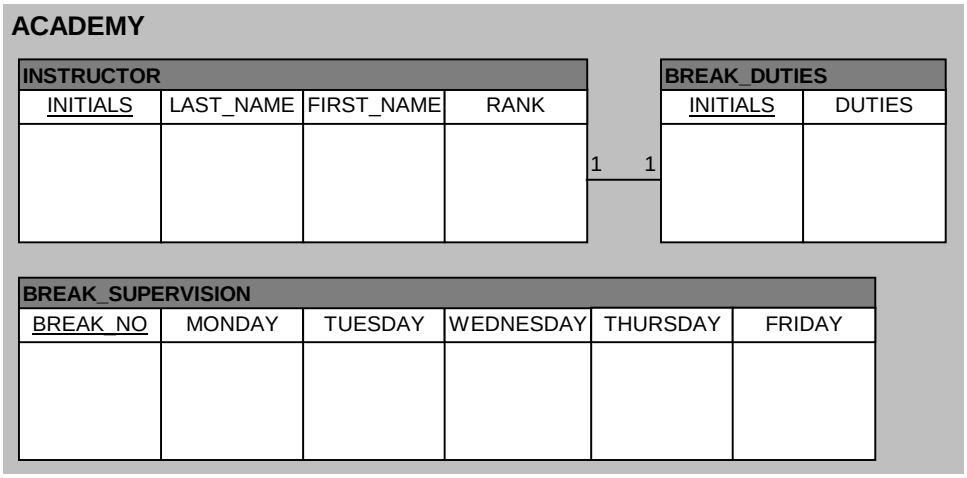


Abbildung 5.18: Datenmodell der Pausenaufsicht

Die Pausenaufsichtspflichten hätte man auch in der Tabelle INSTRUCTOR speichern können. Die Trennung dient in diesem Beispiel dazu, um eine Materialklasse (fcInstructor) zu modellieren, die ihre Daten aus mehr als einer Datenbanktabelle bezieht.

5.3.2.3 Klassendiagramm

Bevor ausgewählte Teile des Entwurfs näher beschrieben werden, soll ein Klassendiagramm den inneren Aufbau des Break Supervision Planner illustrieren. Die Klassen außerhalb des grau unterlegten Bereiches sind Basisklassen aus dem WAM-Framework.

Die konkreten Applikationsklassen sollen hier nicht näher erläutert werden, da sie lediglich Implementierungen der abstrakten Klassen des WAM-Frameworks darstellen, wie z.B. die Klasse fcEnvironment. Bei den abstrakten Klassen ist jedoch eine kurze Erklärung angebracht. Sie werden in den beiden folgenden Abschnitten vorgestellt.

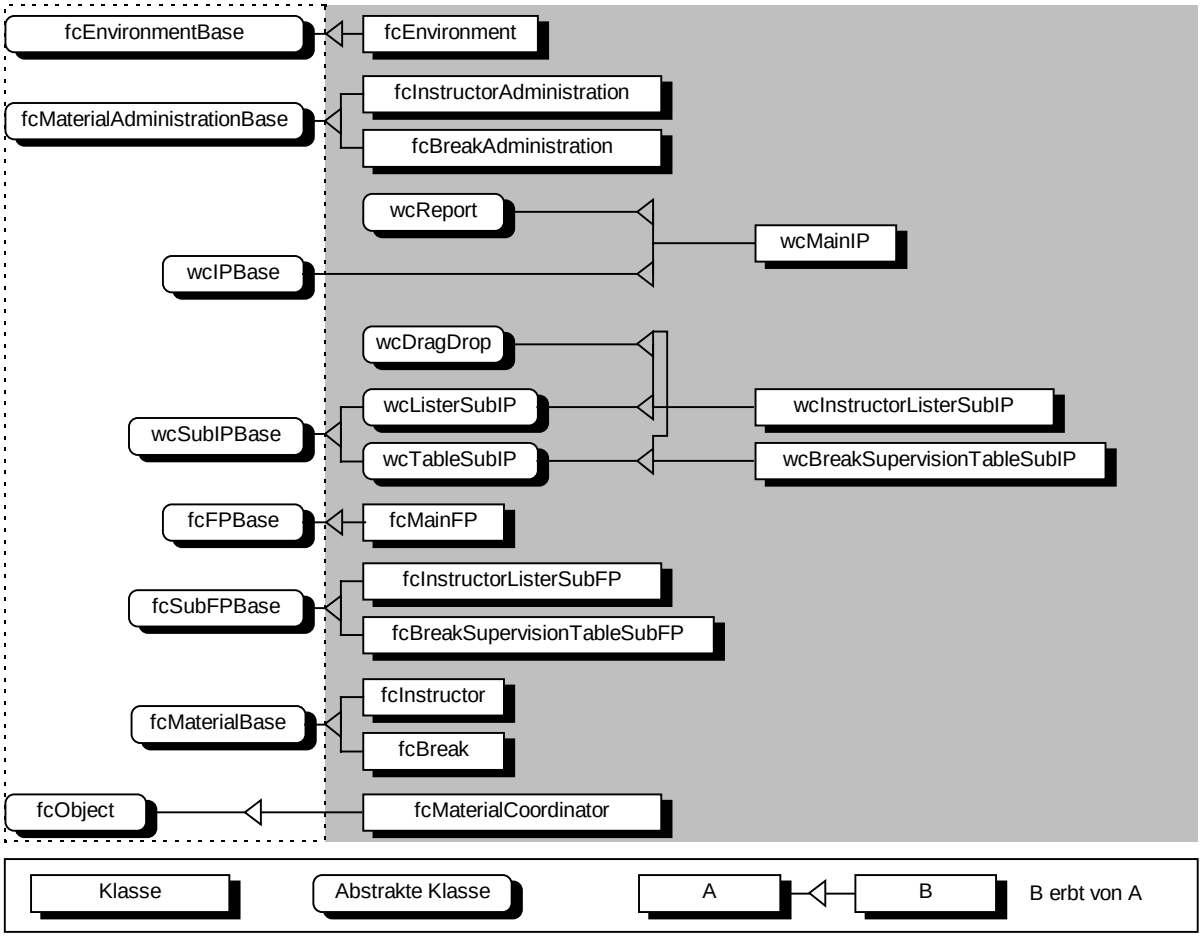


Abbildung 5.19: Break Supervision Planner – Klassendiagramm

5.3.2.4 „Nachrichten-Aspektklassen“

Die Klassen wcDragDrop und wcReport könnte man als „Nachrichten-Aspektklassen“ bezeichnen: Sie empfangen eine bestimmte Kategorie von SAM-Nachrichten (SQLWindows Application Messages) und bieten für jede dieser Nachrichten eine entsprechende Funktion an, die in abgeleiteten Klassen überschrieben werden kann und von der Aspektklasse polymorph gerufen wird. Ein Ausschnitt aus der Klasse wcDragDrop mag hier als Beispiel dienen:

- Functions
  - Function: OnDragDrop
    - Description: Called when the SAM_DragDrop message arrives
    - Returns
    - Parameters
      - ! Handle of the source window
      - Window Handle: hWndSource
    - Static Variables
    - Local variables
    - Actions

```
    ...
♦Message Actions
    ...
    ♦On SAM_DragDrop
        ◊ Call ..OnDragDrop(SalNumberToWindowHandle(wParam))
```

Beim Eintreffen der Nachricht SAM_DragDrop wird polymorph die Funktion OnDragDrop() gerufen. Da SAM_DragDrop den Nachrichtenparameter wParam zur Übergabe eines Fensterbezuges nutzt, wird dieser in den korrekten Typ konvertiert und als Parameter an OnDragDrop() übergeben. Dieses Konzept vereinfacht den Nachrichtenempfang aus Sicht des Applikationsprogrammierers. Bisher mußte er bei der Auswertung einer Nachricht genau wissen, was für eine Bedeutung die Nachrichtenparameter wParam und lParam im Kontext dieser Nachricht haben, um sie dann eventuell noch in den erwarteten Typ zu konvertieren. Durch Benutzung der On*()-Funktionen bekommt er die von der entsprechenden Nachricht benutzten Parameter mit sprechenden Namen und korrektem Typ zur Verfügung gestellt.

wcDragDrop kapselt die Nachrichten des Drag-and-Drop-Mechanismus, mit dem man Zeichenketten und grafische Objekte mit der Maus von einem Kontrollelement in ein anderes kopieren kann.

Die in wcReport gekapselten Nachrichten dienen der Kommunikation mit *ReportWindows*, einem Werkzeug des SQLWindows-Entwicklungssystems, das dem Erstellen und Drucken von Reports dient. Abschnitt 5.3.2.6 geht genauer auf die Report-Programmierung ein.

5.3.2.5 Sub-Interaktionskomponenten für Listen und Tabellen

Für die Realisierung des Pausenaufsichtsplaners wurde ein Auflister und eine Tabelle benötigt. Anstatt diese applikationsspezifisch zu entwickeln, wurden mit den Klassen wcListerSubIP und wcTableSubIP zwei allgemeine Sub-Interaktionskomponenten geschaffen, welche die wesentliche Funktionalität von Auflistern und Tabellen kapseln. Von diesen wurden dann die konkreten Auflister- und Tabellenklassen des Break Supervision Planner abgeleitet.

Die Tabellenfenster von SQLWindows können sehr komfortabel mit Daten gefüllt werden, indem man sie unter Angabe einer SQL-Anweisung automatisch mit den entsprechenden Werten aus der Datenbank füllen läßt. Auf diese Möglichkeit wurde in der allgemeinen Tabellenklasse wcTableSubIP verzichtet, da die Datenbankanbindung von den Materialverwaltungsklassen gekapselt wird und innerhalb der Applikation nur auf den Materialobjekten gearbeitet wird. Das bringt einen Performanzverlust und den in Abschnitt 5.2.2 erwähnten Overhead durch doppelte Datenhaltung (in der Datenbank und in den Materialklassen) mit sich, macht den Break Supervision Planner aber weitestgehend unabhängig von der konkreten Art der Datenhaltung (in diesem Falle durch eine Datenbank).

Die in den letzten beiden Abschnitten vorgestellten abstrakten Klassen gehören eigentlich nicht in die Applikation, sondern in eine Klassenbibliothek. Während man die Sub-Interaktionskomponenten-Klassen durchaus dem WAM-Framework zuordnen könnte, sollte für die Nachrichten-Aspektklassen eine neue Bibliothek angelegt werden. Dies geschähe erst in einem nächsten Schritt, nachdem sich die Klassen in der Beispielapplikation bewährt haben, mit der Beispielapplikation gewachsen sind und schließlich eine Allgemeingültigkeit erreicht haben, die sie zu designierten wiederverwendbaren Klassen macht. Denn grundsätzlich gilt, daß die primäre

Voraussetzung für die Wiederverwendbarkeit von Klassen ihre nachgewiesene Verwendbarkeit sein sollte (vgl. [KGZ94]).

5.3.2.6 Erstellen des Pausenplans über einen Report

In den meisten Programmiersprachen und Softwareentwicklungsumgebungen stellt die Ausgabe von Ergebnissen auf einem Drucker einen erheblichen Implementierungsaufwand dar. Das Windows-Betriebssystem erleichtert diese Aufgabe bereits erheblich, indem es eine einheitliche Beschreibung der Druckausgabe erlaubt, die nahezu unabhängig vom verwendeten Ausgabegerät ist. SQLWindows erleichtert die Druckausgabe noch weiter, indem es eine Schnittstelle zu *ReportWindows* zur Verfügung stellt. ReportWindows ist ein Werkzeug des SQLWindows-Entwicklungssystems, das dem Erstellen und Drucken von Datenbank-Reports dient. Über diese Schnittstelle ist es nun möglich, als Datenquelle nicht etwa eine Datenbank, sondern eine SQLWindows-Applikation anzugeben, die auf Anfrage einen Datensatz nach dem anderen an ReportWindows sendet.

Bevor die oben beschriebene Kommunikation implementiert werden kann, muß zunächst ein entsprechender Report in SQLWindows erstellt werden. Die folgende Abbildung zeigt den Report für einen Pausenplan:

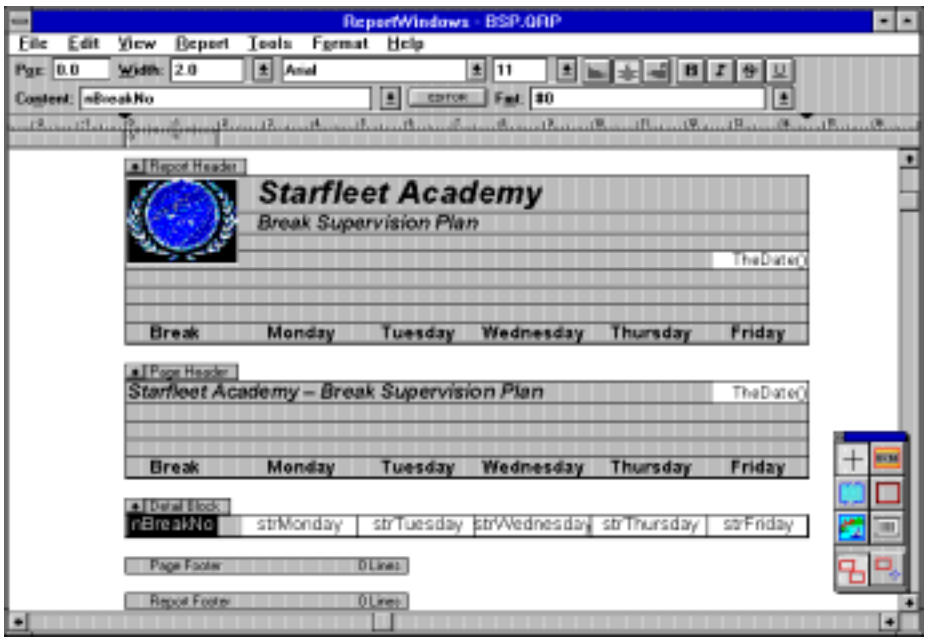


Abbildung 5.20: ReportWindows-Report für einen Pausenplan

Ein Report ist in mehrere Abschnitte unterteilt: Es gibt Kopf- und Fußbereiche für den gesamten Report und für jede Seite und die Datenbereiche (*Detail Blocks*), in denen Eingabeelemente (*Input Items*) definiert werden, die bei der Ausführung des Reports die Daten aufnehmen.

Neben statischen Text- und Grafikelementen können Formeln und die bereits erwähnten Eingabeelemente definiert werden. Alle grau hinterlegten Elemente im obigen Beispiel sind statische Elemente. TheDate() ist eine Formel, die das aktuelle Datum ausgibt. Zur Definition von Formeln enthält ReportWindows einen Formeleditor. nBreakNo und strMonday...strFriday sind Eingabee-

lemente. Sie haben einen Typ (erkennbar am Präfix, das den Namenskonventionen von SQLWindows entspricht; vgl. Abschnitt 5.3.3.1) und können auch in Formeln verwendet werden.

Um von einer SQLWindows-Applikation aus einen Report zu generieren, muß eine der SalReport*()-Funktionen aufgerufen werden: SalReportPrint() druckt den Report auf einem Ausgabegerät aus, SalReportPrintToFile() schreibt einen Report im RTF- oder ASCII-Format in eine Datei, und SalReportView() zeigt den Report auf dem Bildschirm an, wobei die Möglichkeit zum anschließenden Ausdruck besteht. Die letzte Funktion bietet mit ihrer Anzeige- und Druckmöglichkeit die größte Flexibilität, weshalb sie auch für den Break Supervision Planner ausgewählt wurde. Der konkrete Aufruf sieht wie folgt aus:

```
◊ Set m_hWndReport = SalReportView(
    GetWindowHandle(),
    hWndNULL,
    'bsp.qrp',
    'm_nBreakNo, m_strMonday, m_strTuesday, m_strWednesday, m_strThursday, m_strFriday',
    'nBreakNo, strMonday, strTuesday, strWednesday, strThursday, strFriday',
    nError)
```

Der erste Parameter gibt das Fensterobjekt an, das als Datenlieferant dienen soll. Da der zweite Parameter den Wert hWndNULL (ungültiger Fensterbezug) hat, wird das von ReportWindows vordefinierte Ausgabefenster gewählt. `BSP.QRP` ist der Name der (oben abgebildeten) Report-Datei. Die folgende Zeichenkette ('m_nBreakNo,...) enthält die Namen von SQLWindows-Variablen, welche die Daten halten, die zur Übergabe an ReportWindows bestimmt sind. Empfangen werden die Daten von den Eingabeelementen, die in einer zweiten Zeichenkette ('nBreakNo,...) angegeben sind. Dabei ist die erste SQLWindows-Variable in der ersten Zeichenkette dem ersten Eingabeelement der zweiten Zeichenkette zugeordnet usw. Der letzte Parameter empfängt im Fehlerfalle einen Fehlercode.

ReportWindows sendet nun Nachrichten an das als Datenlieferant spezifizierte Fenster, die jeweils einen neuen Datensatz anfordern. Der Datenlieferant weist daraufhin den oben angegebenen SQLWindows-Variablen die entsprechenden Werte zu bzw. bricht die Datenlieferung ab, wenn keine weiteren Datensätze existieren. Diese Nachrichten werden von der Klasse wcReport in SQLWindows-Funktionen gekapselt (vgl. Abschnitt 5.3.2.4).

Die folgende Abbildung zeigt die Bildschirmausgabe eines vom Break Supervision Planner generierten Reports:

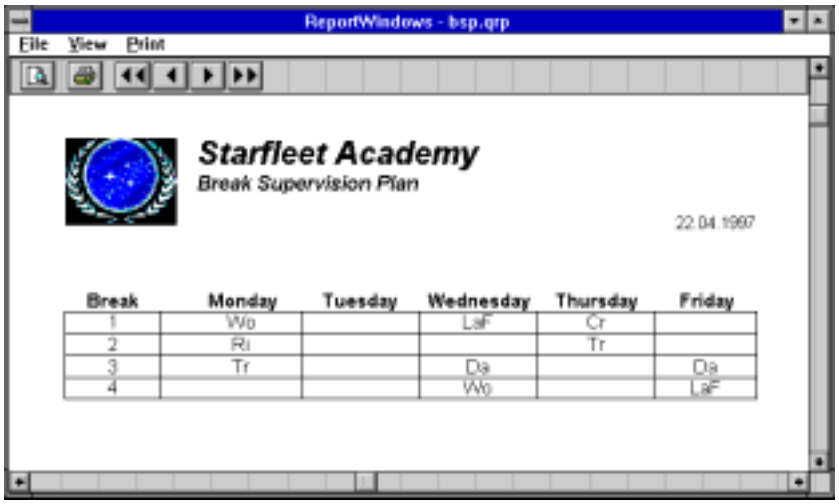


Abbildung 5.21: Mit *ReportWindows* generierter Report

Mit der Report-Schnittstelle stellt SQLWindows ein mächtiges Werkzeug zur Druckausgabe zur Verfügung.

5.3.3 Programmier- und Dokumentationsrichtlinien

Die Untersuchung der Programmiersprache SAL hat gezeigt, daß dieser Sprache mit dem Abstraktionsmechanismus ein wichtiges Konzept objektorientierter Programmiersprachen fehlt. Dies und das Fehlen eines Referenzkonzeptes machen das Deklarieren globaler Objekte oft unumgänglich. Auf diese globalen Objekte kann nun von jeder Stelle des Programms aus zugegriffen werden, wobei deren komplette Implementierung (alle Attribute und Funktionen) für den Zugriff offensteht. Es liegt nun in der Hand des SQLWindows-Programmierers, ob er sich an gewisse Richtlinien hält, die einen objektorientierten Programmierstil auch in SAL erlauben. Ein solcher Stil kann durch Programmier- und Dokumentationsrichtlinien unterstützt, wenn auch leider nicht erzwungen werden. Bei der Untersuchung von SAL und im Verlaufe des Framework-Entwurfs entstanden einige solcher Richtlinien, die im folgenden vorgestellt werden sollen.

5.3.3.1 Namenskonventionen

Alle Klassen, Objekte und Variablen des Frameworks halten sich an bestimmte Namenskonventionen, die zum großen Teil von Gupta vorgeschlagen (vgl. z.B. [Gup94a]) und um eigene Konventionen erweitert wurden.

Klassen, Objekte und Variablen beginnen mit einem Präfix, der Aussage über ihren Typ gibt, gefolgt von einem Namen, der mit einem Großbuchstaben beginnt. Die maximale Länge für SAL-Bezeichner beträgt 32 Zeichen.

Klassen	
wc	Window class
fc	Functional class
Objekte	
udv	User-defined variable
	Window class-Objekte bekommen ein Präfix entsprechend des zugrunde liegenden Fenstertyps (siehe unten)
Datentypen	
b	Boolean
dt	Date / Time
ls	Long String
str	String
n	Number
hWnd	Window Handle
hFile	File Handle
hSql	Sql Handle
a	Array (als Präfix für alle Datentypen; Beispiel: adt)
c	Constant (als Präfix für alle Datentypen; Beispiel: cdt)
r	Receive data type (als Präfix für alle Datentypen; Beispiel: rdt)
m_	Member variable (als Präfix für class / instance / window variables)
Hauptfenster	
frm	Form Window
tbl	Table Window
dlg	Dialog Box
mdi	MDI Window
qst	Quest Window
Kindfenster	
df	Data Field
ml	Multiline Text Field
pb	Push Button
rb	Radio Button
cb	Check Box
ob	Option Button
lb	List Box
cmb	Combo Box
ctbl	Child Table Window
cqst	Quest Child Window
pic	Picture
sb	Scroll Bar (Horizontal/Vertical)
cc	Custom Control

Tabelle 5.2: SQLWindows-Namenskonventionen

5.3.3.2 Kommentare im Quellcode

SAL-Kommentare beginnen mit einem Ausrufungszeichen '!'. Sie können sich über eine ganze Zeile erstrecken oder innerhalb der Zeile beginnen, reichen aber immer bis zum Ende der Zeile. Kommentare können die hierarchische Strukturierung des Outline nutzen, einschließlich der Möglichkeit des Ausblendens von Hierarchieebenen. Damit wird dem SQLWindows-Programmierer ein gutes Werkzeug zur Quellcode-Dokumentation an die Hand gegeben – von dem ausgiebig Gebrauch gemacht werden sollte. Das Fehlen eines Abstraktionskonzeptes z.B. macht es notwendig, die Zugriffsbeschränkungen auf Objekte in Form von Kommentaren anzugeben. Ohne gute Kommentierung des Quellcodes wird ein SQLWindows-Programm vor allem für Außenstehende leicht undurchschaubar und nur schwer nachvollziehbar. Kommentare im Quellcode ersetzen natürlich nicht Dokumente wie die Entwurfsdokumentation, die Systemspezifikation oder die Benutzerhandbücher. Sie helfen aber beim schnellen Verstehen von Details und zeichnen den Entwurf im Quellcode nach. Ohne Kommentare wäre die objektorientierte Programmierung vor dem Hintergrund der Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit ein hoffnungsloses Unterfangen.

5.3.3.3 Schutzmechanismen

Wie bereits beschrieben, verletzt SQLWindows das Geheimnisprinzip. So bietet es auch keine Schutzmechanismen für Klassenelemente. Die Programmiersprache C++ kennt drei Zugriffskontrollklassen für Klassenelemente (vgl. [Str92]):

- Ist ein Element private, so kann sein Name nur in Element-Funktionen und friends der Klasse verwendet werden, in der es deklariert ist.
- Ist ein Element protected, so kann sein Name nur in Element-Funktionen und friends der Klasse verwendet werden, in der es deklariert ist, oder in Element-Funktionen und friends einer von dieser Klasse abgeleiteten Klasse.
- Ist ein Element public, so kann sein Name in beliebigen Funktionen verwendet werden.

Außerdem kennt sie das Schlüsselwort virtual, das denjenigen Funktionen vorangestellt wird, die in abgeleiteten Klassen überschrieben werden können (virtuelle Funktionen). Rein virtuelle Funktionen erhalten den Zusatz = 0. Diese Funktionen müssen in abgeleiteten Klassen überschrieben werden.

In Anlehnung an diese Schutzmechanismen wurden die Funktionen einer SQLWindows-Klasse in mehrere Kategorien eingeteilt. Die einzelnen Kategorien werden im Abschnitt Functions einer Klasse durch eine der folgenden Kommentarzeilen eingeleitet:

! Internal	Die Funktion darf nur innerhalb derjenigen Klasse verwendet werden, in der sie deklariert ist, sowie innerhalb der von dieser Klasse abgeleiteten Klassen (≈ private und protected).
! Public	Die Funktion darf von beliebigen Funktionen verwendet werden (≈ public).
! To be overridden	Die Funktion kann bzw. muß in abgeleiteten Klassen überschrieben werden (≈ virtuelle bzw. rein virtuelle Methoden).
! Late-bound [*Kategorie*]	Die Funktion ist das spät gebundene Äquivalent einer zu überschreibenden Funktion. Ihr Name beginnt mit einem Unterstrich '_' und ist ansonsten identisch mit dem der zu überschreibenden Funktion (zur Notwendigkeit dieser Funktionen siehe Abschnitt 4.3.7). In der abgeleiteten Klasse wird die überschriebene Funktion der Kategorie zugeteilt, die in den eckigen Klammern angegeben ist.

Tabelle 5.3: Schlüsselworte für Zugriffsbeschränkungen

Die Attribute einer Klasse (interne Funktionen und Klassenfunktionen) sind immer Internal. Auf sie darf nur über entsprechende Get()- bzw. Set()-Funktionen lesend bzw. schreibend zugegriffen werden.

Es sei nochmals darauf hingewiesen, daß es sich lediglich um Kommentare und Konventionen handelt – es liegt im Ermessen des Programmierers, diese im Sinne eines sauberen Entwurfs zu verwenden und strikt einzuhalten.

5.4 Zusammenfassung

In diesem Kapitel wird die Entwicklung eines WAM-Frameworks für SQLWindows beschrieben. Als Basis für ein solches Framework dienen die WAM-Entwurfsmuster von Riehle und Züllighoven (vgl. [RZ95]).

Bei dem Versuch, diese Entwurfsmuster in SQLWindows zu implementieren, zeigte sich deutlich, daß vielfach das Fehlen eines Referenzmechanismus in SQLWindows eine sichere Implementierung unmöglich machte. Auch das SQLWindows Object System (SOS), das einen eingeschränkten Referenzmechanismus implementiert, konnte nicht in allen Fällen Abhilfe schaffen. Grundsätzlich aber ermöglicht das im Rahmen dieser Arbeit entwickelte TANDEM-Framework eine Softwareentwicklung nach der WAM-Methodik mit SQLWindows.

Dieses Kapitel zeigt ferner, daß die hier entwickelten SQLWindows-Klassen ein Framework bilden, und keine Klassenbibliothek, da die Klassen in einer logischen Beziehung stehen, die unabhängig von den Vererbungsbeziehungen ist. Diese logische Beziehung faßt die Klassen zu einem Entwurf zusammen. Diese Kapselung von Entwurfswissen ist das Hauptmerkmal von Frameworks gegenüber Klassenbibliotheken.

Wie oben angesprochen, war bei einigen Entwurfsmustern eine sichere Implementierung unmöglich. Hier konnten nur strenge Programmier- und Dokumentationsrichtlinien helfen, um die Pro-

grammierung mit SQLWindows sicherer und verständlicher zu machen. Dabei liegt die Verantwortung für einen sauberen Entwurf stets beim Programmierer.

Das Fehlen eines Referenzmechanismus in SQLWindows wurde bereits mehrfach als gravierendes Problem identifiziert. Das nun folgende Kapitel stellt das SQLWindows Object System (SOS) vor, das dieses Problem zu lösen versucht, indem es SQLWindows um einen Referenzmechanismus erweitert.

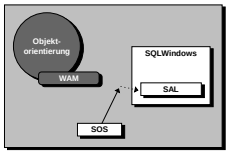
6 Das SQLWindows Object System (SOS)

Das Fehlen eines vollständigen Referenzkonzeptes in SQLWindows ist einer der größten Schwachpunkte dieses Entwicklungssystems. Wie bereits in Abschnitt 2.3.1 erläutert und aus dem Dimensionskatalog ersichtlich, sind Referenzen auf Objekte ein wichtiges Konzept objektorientierter Programmierung.

Die Entwickler der Firma E.I.S. haben mit ihrem Packet-Mechanismus eine Lösung zur Umgehung dieses Problems gefunden. Aufbauend auf dieser Idee entstand das *SQLWindows Object System* (*SOS*), dessen Funktionsweise und Entstehungsgeschichte im folgenden beschrieben werden soll.

Abschnitt 6.1 enthält eine Kurzbeschreibung des SOS. In Abschnitt 6.2 wird die Entstehungsgeschichte des SOS nachgezeichnet, aus dem die in Abschnitt 6.3 beschriebenen Aufgaben des SOS abgeleitet werden können. Dieser Abschnitt nennt auch die Grenzen des SOS im Vergleich zu einem „echten“, allgemeinen Referenzmechanismus. Abschnitt 6.4 zeigt die Komponenten des SOS, so wie sie der Anwendungsentwickler sieht. In Abschnitt 6.5 wird anhand eines Beispiels die Funktionsweise und die Benutzung des SOS beschrieben. Dem softwaretechnisch interessierten und in der C++ - Programmierung bewanderten Leser bietet der Abschnitt 6.6 einen Einblick in die Interna des SOS. Dieser Abschnitt ist sehr technisch und für das Verständnis der Funktionsweise des SOS nicht unbedingt notwendig. Abschnitt 6.7 faßt die Inhalte und Ergebnisse dieses Kapitels abschließend zusammen.

Die Klassenschnittstelle des SOS und ein Verzeichnis der SOS-Funktionen findet man in Anhang B.



Das SQLWindows Object System (SOS) soll den fehlenden Referenzmechanismus in SQLWindows weitestmöglich kompensieren. Das SOS steht exemplarisch für die Erweiterung einer Programmiersprache um neue Sprachkonzepte.

6.1 Kurzbeschreibung des SOS

Dieser Abschnitt beschreibt die Aufgaben des SOS und seine Rolle bei der Softwareentwicklung mit SQLWindows.

Aufgaben

Hauptaufgabe des SQLWindows Object System ist das Speichern von SQLWindows-Objekten zum Zwecke der Übergabe dieser Objekte (genauer: einer Referenz auf diese Objekte) an andere Objekte oder Funktionen. Um eine Objektreferenz übergeben zu können, muß diese eindeutig sein. Deshalb enthält das SOS einen Objekt-Manager, der eindeutige Objekt-Identifikatoren generiert und verwaltet.

Motivation

Die Möglichkeit, Referenzen auf Objekte definieren zu können, ist unter anderem für die Implementierung von Benutztbeziehungen zwischen Objekten notwendig: Der Zustand eines Objektes, d.h. seine (privaten) Attribute, muß neben Variablen einfachen Typs auch Beziehungen zu anderen Objekten beinhalten. Da ein solches Objektattribut eventuell auch für andere Objekte als Attribut von Bedeutung ist, muß anstatt des Objektes selber eine Referenz auf das Objekt als Attribut gespeichert werden, oder das zu referenzierende Objekt wird global deklariert. Letztere Lösung verstieße aber gegen das Prinzip der Datenabstraktion. Objektreferenzen als Attribute sind somit ein notwendiges Sprachmittel objektorientierter Programmiersprachen.

Struktur

Die Klasse fcAtomizable bildet die Schnittstelle des SOS.

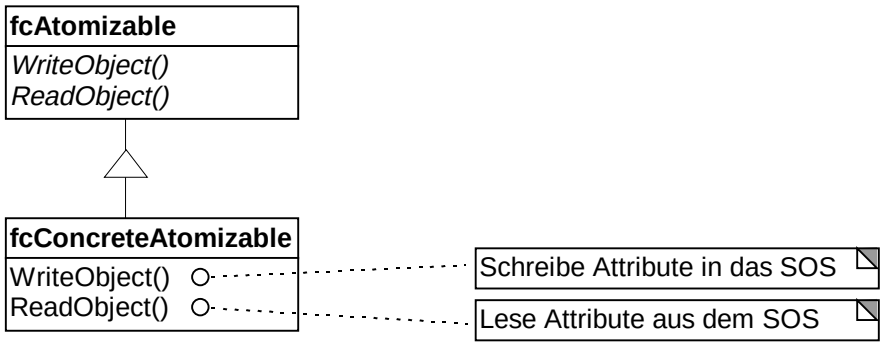


Abbildung 6.1: Struktur der Klasse fcAtomizable als SOS-Schnittstelle

Eine konkrete atomisierbare Klasse implementiert die beiden virtuellen Funktionen von fcAtomizable, WriteObject() und ReadObject(), in denen die Attribute des Objektes in das SOS geschrieben bzw. aus diesem ausgelesen werden. Abschnitt 6.5 beschreibt die Benutzung des SOS anhand eines Beispiels, und Anhang B führt alle SOS-Funktionen auf.

6.2 Entstehungsgeschichte des SOS

Gegenstand dieses Abschnitts sind die Probleme und Beweggründe, die dazu geführt haben, daß SQLWindows um ein externes Objektsystem erweitert werden mußte, um objektorientiert in SQLWindows entwickeln zu können. Aus diesen Problemen erwuchsen konkrete Aufgaben, die das SOS abdecken sollte.

6.2.1 Referenzen auf Objekte

Einer der zentralen Mechanismen in objektorientierten Programmiersprachen ist die Möglichkeit, Referenzen (auch Verweise genannt) auf Objekte als Attribute eines Objektes zu definieren. Benötigt werden solche Referenzen z.B. für den gemeinsamen Zugriff mehrerer Objekte auf ein anderes Objekt unter Wahrung des Geheimnisprinzips oder zur Implementierung des Prinzips der losen Kopplung zwischen Funktions- und Interaktionskomponente (vgl. Abschnitt 2.3.1).

In Sprachen wie Eiffel oder Java sind alle Attribute eines Objektes entweder einfache Typen (wie Integer, Real oder Character) oder Referenzen. In C++ können Objekte selbst Attribut eines anderen Objektes sein, aber die Sprache erlaubt auch die Definition von Objektreferenzen (in Form von Zeigern auf Objekte) als Attribute.

Initial sind solche Objektreferenzen leer, d.h. sie verweisen auf nichts. Dies läßt sich auf zwei Arten ändern:

- Man erzeugt ein neues Objekt und verbindet es mit dem Verweis
- Man verbindet den Verweis mit einem bereits existierenden Objekt

Für den ersten Fall stellen objektorientierte Programmiersprachen entsprechende Operatoren (!! in Eiffel, new in C++ und Java) oder Nachrichten (new in Smalltalk) zur Verfügung. Im zweiten Fall bekommt man i.d.R. einen Verweis auf das existierende Objekt als Parameter einer Methode übergeben und verbindet diesen mit dem eigenen Verweis.

SQLWindows kennt ein solches Referenzkonzept nur in einem sehr eingeschränkten Sinne. Als Attribute eines Objektes können einfache SQLWindows-Typen (Number, String etc.) oder Objekte definiert werden, aber keine Objektreferenzen. Einfache Typen als Funktionsparameter werden als Wert (by value) übergeben. Das Schlüsselwort Receive veranlaßt die Übergabe als Referenz (by reference), wie folgende Beispielfunktion verdeutlicht:

```
◊ Function: Inc
    ◊ Parameters
        ♦ Receive Number: nNumber
        ♦ Number: nIncrement
    ◊ Actions
        ♦ Set nNumber = nNumber + nIncrement
```

Der Aufruf Inc(nCounter, 10) erhöht den Wert der Variablen nCounter um 10, da nCounter als Referenz an Inc() übergeben wurde.

Anders verhält es sich mit Objekten, die Instanzen von funktionalen Klassen sind. Diese werden immer als Referenz übergeben. Mit dieser Referenz kann man dann innerhalb der Funktion ar-

beiten, aber man kann sie nicht als Attribut der Klasse speichern. Objekte, die Instanzen von Fensterklassen sind, können nicht als Parameter übergeben werden.

Die Übergabe von Objektreferenzen als Parameter und das Speichern von Objektreferenzen als Attribute sind eine Grundvoraussetzung für die objektorientierte Programmierung. Es mußte also ein Mechanismus gefunden werden, um in SQLWindows Objektreferenzen als Klassenattribute realisieren zu können, und um über solche Referenzen die entsprechenden Objekte zu erreichen. Die Firma E.I.S. hat mit ihrem Packet-Mechanismus einen Lösungsansatz für dieses Problem gefunden.

6.2.2 Der E.I.S. Packet-Mechanismus

Mit dem Packet-Mechanismus der Entwicklungsumgebung *Application Builder* von der Firma E.I.S. (vgl. [EIS96]) können Daten oder Gruppen von Daten externalisiert („atomisiert“) und wieder zurückgelesen werden. Dazu müssen alle Klassen, die diesen Mechanismus nutzen möchten, von einer Klasse clsContainer erben.

Zum Speichern eines Paketes übergibt das SQLWindows-Objekt alle seine Daten unter Angabe der Paket-ID, eines Namens und des eigentlichen Wertes nacheinander an den Paketdienst. Dieser speichert die Pakete in einer Textdatei, deren Aufbau einer hierarchisch gegliederten Windows-Initialisierungsdatei ähnelt.

Problematisch am Packet-Mechanismus ist die Loslösung vom Objektbegriff: Es werden Daten verpackt, die nicht unbedingt Objekten entsprechen. Das mag zwar sehr praktisch sein, führt aber ein neues Konzept (das des Paketes) ein, auf das ein reines Objektmodell getrost verzichten könnte. Die grundsätzliche Idee der Externalisierung und deren Umsetzung als DLL ist jedoch sehr gut; sie diente als direkte Grundlage für das SQLWindows Object System (SOS). Da das SOS aber auf das Paketkonzept und somit auch auf eine Paket-ID als eindeutigen Identifikator verzichtet, mußte zunächst ein eigenständiges Objektidentitätskonzept entwickelt werden.

Die Aufgaben, die der zu entwickelnde Referenzmechanismus zu erfüllen hatte, ließen sich also wie folgt formulieren:

- Implementierung eines Objektidentitätskonzeptes
- Speichern und Lesen von SQLWindows-Objekten unter Angabe eines eindeutigen Objektidentifikators

Diese beiden Aufgaben werden im folgenden Abschnitt näher beschrieben.

6.3 Aufgaben und Grenzen des SOS

6.3.1 Objektidentität

Den Ausgangspunkt bei der Konstruktion eines Referenzmechanismus bildete die Frage, wie Objektreferenzen charakterisiert werden können. In gängigen objektorientierten Programmiersprachen werden physikalische Speicheradressen benutzt. SQLWindows bietet jedoch keinen Zugriff auf die Speicheradressen der Objekte (sonst hätte es ja einen Referenzmechanismus). Eine

weitere Möglichkeit sind eindeutige Objekt-Identifikatoren (kurz: Objekt-IDs). Für die SQLWindows-Fensterklassen ist dies z.B. der Fensterbezug, allerdings mit der Einschränkung, daß von einem Fensterklassen-Objekt mehrere Instanzen (mit unterschiedlichen Fensterbezügen) erzeugt werden können. Funktionale Klassen haben keinen solchen Identifikator, so daß man einen allgemeineren Objekt-ID-Begriff fassen muß, der beide Klassenkategorien umfaßt. Zu diesem Zweck wurde die Basisklasse fcObject definiert, von der alle anderen Klassen des Frameworks erben (siehe Abschnitt 5.2.1). Diese kann über die SOS-Funktion SOSGetObjectId() eine eindeutige Objekt-ID anfordern, so daß jedes SQLWindows-Objekt, das vom Typ einer von fcObject abgeleiteten Klasse ist, eindeutig identifiziert werden kann.

Die Objekt-ID wird nur dann vergeben, wenn sie explizit verlangt wird. Dieses Prinzip der „object identity on demand“ verhindert die überflüssige Vergabe von Identitäten an Objekte, die während der gesamten Laufzeit des Programmes nie referenziert werden. Sollte sich das in einer späteren Version des Programmes ändern, so ist dieser Mechanismus dennoch mächtig genug, um das Objekt ohne großen Änderungsaufwand mit einer Objekt-ID versehen zu können.

6.3.2 Speichern von SQLWindows-Objekten

Das Speichern der SQLWindows-Objekte im SOS sollte zumindest hinsichtlich der Datenübergabe ähnlich geschehen wie bei dem in Abschnitt 6.2.2 beschriebenen Packet-Mechanismus: Das SQLWindows-Objekt übergibt jedes einzelne Attribut unter Angabe von Objekt-ID, Attributnamen und Attributwert. Zum Speichern stand jedoch mit dem Atomizer-Pattern ein mächtiger Mechanismus zur Verfügung, der von der konkreten Art des Speicherns abstrahiert, so daß Speichermedium und -methode ohne Änderungsaufwand ausgetauscht werden können (vgl. [RS+96]). Ein weiterer Vorteil ist, daß grundsätzlich auch andere Programme über die standardisierte Atomizer-Schnittstelle auf SQLWindows-Objektdaten zugreifen können, so daß ein Datenaustausch zwischen SQLWindows-Programmen und einem in C++ geschriebenen Programm über diese Schnittstelle stattfinden könnte. Allerdings kann ein Atomizer in der vorliegenden Implementation nur C++ - Objekte speichern. Deshalb enthält das SOS einen Mechanismus, der das übergebene SQLWindows-Objekt annimmt und in ein C++ - Objekt umwandelt, welches dann atomisiert wird.

Man mag sich nun fragen, warum ein SQLWindows-Objekt zweimal atomisiert wird: Zunächst bei der (attributweisen) Übergabe an das SOS, und dann nach der Umwandlung in ein C++ - Objekt durch den Atomizer. Der Grund liegt in den unterschiedlichen Datenformaten bei der Umwandlung. Die C++ - Repräsentation der SQLWindows-Objekte verwendet intern C-Definitionen und Strukturen, die äquivalent zu den SQLWindows-Datentypen sind. Im Atomizer werden jedoch nur Standard-C-Datentypen gespeichert, so daß die atomisierten Objekte ohne genaues Wissen über die SQLWindows-Datentypen aus dem Atomizer gelesen werden können. Dieser Abstraktionsschritt rechtfertigt eine doppelte Atomisierung, auch wenn sie für das Referenzkonzept nicht unbedingt notwendig ist – man hätte die über die SOSWrite*()-Funktionen an das SOS übergebenen Attribute der SQLWindows-Objekte auch direkt in den Atomizer schreiben können.

Mit der Verwendung der Standard-C-Datentypen öffnet sich ein weiteres Anwendungsfeld: Man kann die atomisierten SQLWindows-Objekte in einer verteilten Umgebung zwischen verschiedenen Klienten verschicken. Da SQLWindows ein Client/Server-System ist, finden sich leicht Anwendungen, für die solche „Netz-Objekte“ von Nutzen sein können.

In Abschnitt 6.6.4 wird das Problem der Transformation von Datentypen ausführlich behandelt.

6.3.3 Unterschied zu einem „echten“ Referenzmechanismus

Wie gesehen, hält das SOS in Wirklichkeit keine Referenzen auf (SQLWindows-)Objekte, sondern speichert diese als Attributfolgen ab. Dieses Vorgehen hat eine wichtige Konsequenz: Im SOS gespeicherte Objekte werden mit dem Zustand eingefroren, den sie zum Zeitpunkt der Speicherung hatten. Dabei ist der *Zustand* eines Objektes definiert als Tupel <*value*(a_1), *value*(a_2), ..., *value*(a_n)>, wobei *value*(a_i) den aktuellen Wert des Attributes a_i bezeichnet. Wird ein Objekt im SOS gespeichert und anschließend geändert, bevor ein anderes Objekt dieses aus dem SOS anfordert, so ist der Zustand des rekonstruierten Objektes veraltet und somit ungültig. Abbildung 6.2 verdeutlicht diesen Umstand beispielhaft: Objekt 1 wird in das SOS geschrieben. Anschließend wird der Wert des Attributes b geändert. Man beachte, daß die Kopie des Objektes im SOS unverändert bleibt. Wird nun ein neues Objekt 2 instantiiert und mit den Werten des Objektes 1 aus dem SOS initialisiert, so repräsentiert das Objekt 2 den mittlerweile veralteten Zustand des Objektes 1. Dieser Tatsache muß sich der Benutzer des SOS bewußt sein, wenn er das SOS als Referenzmechanismus benutzen möchte.

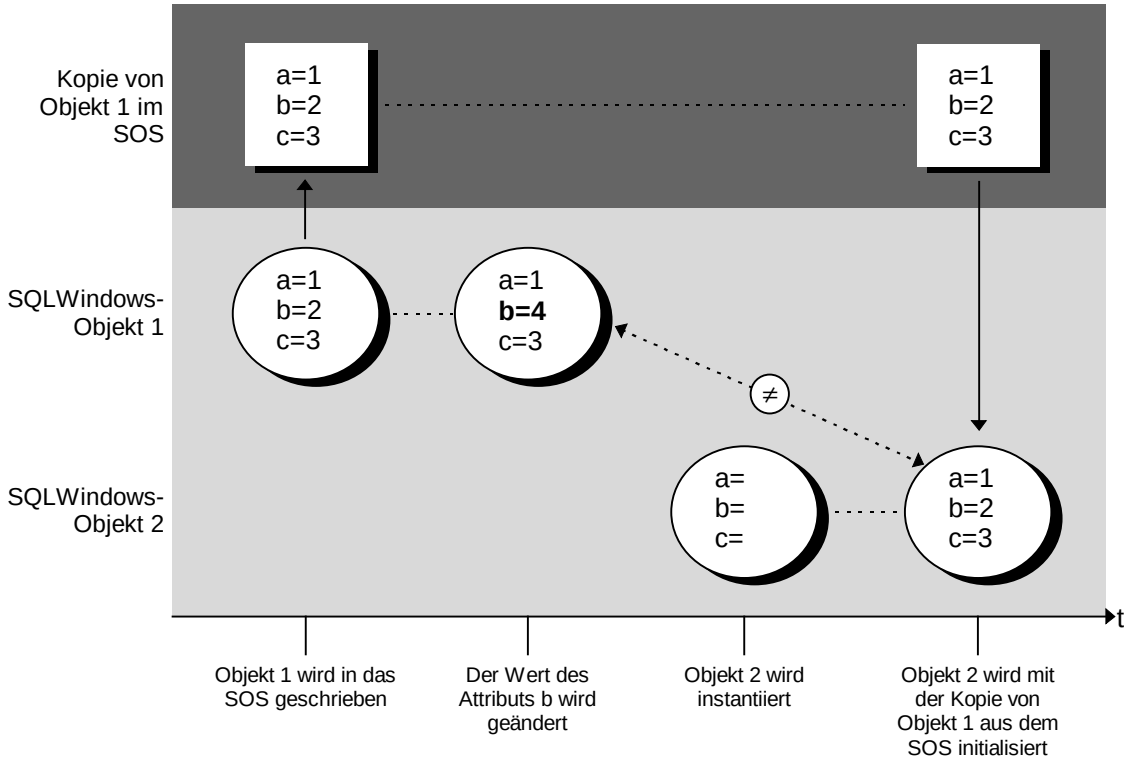


Abbildung 6.2: Zustandsbehaftetheit der im SOS gespeicherten Objekte

Nun könnte man auf die Idee kommen, daß immer dann, wenn ein Objekt aus dem SOS gelesen werden soll, das SOS überprüft, ob das Objekt geändert wurde. Dies setzt aber die Möglichkeit voraus, das zugehörige SQLWindows-Objekt zu identifizieren. Man bräuchte also eine Art Objektverwalter innerhalb von SQLWindows, der alle Objekte kennt und auf Anfrage an das SOS übergeben kann. Dieser Objektverwalter müßte Referenzen auf alle SQLWindows-Objekte hal-

ten, womit wir wieder beim Ausgangsproblem wären. Es gibt keine Möglichkeit, die Zustandsbehaftetheit des SOS zu umgehen. Ist man sich dieses Umstands bewußt, so kann das SOS trotzdem von großem Vorteil sein. So kann z.B. der Beobachtermechanismus auf dem SOS aufbauen, da beim Registrieren eines Beobachters das ins SOS geschriebene Beobachter-Objekt sofort vom beobachteten Objekt gelesen wird und somit aktuell ist. Außerdem interessiert sich das beobachtete Objekt nur für den Fensterbezug des Beobachters – ein Attribut, dessen Wert sich zur Laufzeit nicht ändert.

6.4 Die Komponenten des SOS

Aus Sicht des Anwendungsprogrammierers besteht das SOS aus zwei Dateien: `SOS.APL` ist eine SQLWindows-Library, welche die SOS-Funktionen für die Benutzung innerhalb von SQLWindows definiert. Die Datei `SOS.DLL` ist eine Dynamic Link Library (DLL), geschrieben in C++, die SQLWindows-Objekte in ein C++ - Objektmodell transferiert und atomisiert. Außerdem stellt sie einen Objekt-Manager zur Verfügung, der eindeutige Objekt-IDs vergeben kann.

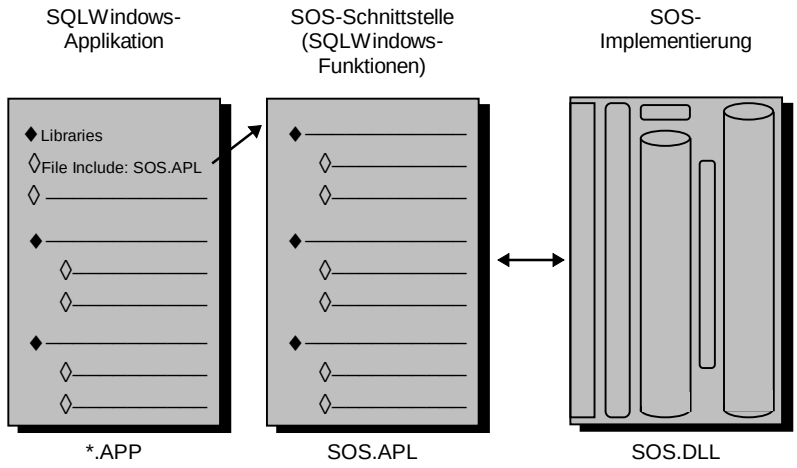


Abbildung 6.3: Die Komponenten des SOS

6.4.1 SOS.APL

Die SQLWindows Application Library `SOS.APL` definiert die Schnittstelle des SOS für SQLWindows. SQLWindows Application Libraries (APL) sind Sammlungen von SQLWindows-Klassen, -Objekten und -Funktionen, die mehreren Applikationen zur Verfügung gestellt werden sollen. Der Aufbau einer Application Library im Outline entspricht dem einer SQLWindows-Applikation, so daß auch externe Funktionen aus Dynamic Link Libraries (DLL) in einer APL deklariert werden können (vgl. auch Abschnitt 6.6.4). Genau dies tut `SOS.APL`, indem es die in Form von C-Funktionen beschriebene Schnittstelle des SOS als externe SQLWindows-Funktionen beschreibt und somit für SQLWindows-Applikationen verfügbar macht. Die einzelnen Funktionen werden in Anhang B beschrieben.

6.4.2 SOS.DLL

Die in C++ geschriebene Dynamic Link Library (DLL) `SOS.DLL` kapselt das eigentliche SQLWindows Object System. DLLs sind Bibliotheken für Microsoft Windows, die nicht zur Übersetzungszeit, sondern erst zur Laufzeit dem benutzenden Programm hinzugebunden werden. Dabei kann in der DLL genau festgelegt werden, welche Funktionen exportiert werden sollen. Neben Funktionen können auch C++ - Klassen exportiert werden, die aber nur in C++ - Programmen genutzt werden können.

Der interne Aufbau des SOS wird später in Abschnitt 6.6 beschrieben. Im folgenden soll zunächst die Nutzung des SOS aus Sicht des Applikationsentwicklers dargestellt werden.

6.5 Die Benutzung des SOS – ein Anwendungsbeispiel

Die Benutzung des SOS soll im folgenden anhand eines einfachen Beispiels erläutert werden. Eine Beschreibung der Klassen- und Funktionsschnittstelle des SOS findet man in Anhang B.

Die folgende Beispielklasse mit zwei einfachen Attributen und einem Array-Attribut soll die Benutzung des SOS exemplarisch zeigen (vgl. `SOS-DEMO.APP`). Der Klassenbeschreibung folgt eine Erläuterung der einzelnen Funktionen.

```
♦Functional Class: fcMyAtomizableClass
   ◊ Description: This class can be written to and read from the SOS
   ♦Derived From
         ◊ Class: fcAtomizable
   ◊ Class Variables
   ♦Instance Variables
         ◊ Boolean: bCheck
         ◊ String: strMessage
         ◊ Number: anPrime[*]
   ♦Functions
         ◊ ! Internal
         ♦Function: WriteAttributes
            ◊ Description:
            ♦Returns
               ◊ ! TRUE if successful, else FALSE:
               ◊ Boolean:
            ◊ Parameters
            ◊ Static Variables
            ♦Local variables
               ◊ Boolean: bOk
            ♦Actions
               ◊ Set bOk = fcAtomizable.WriteAttributes()
               ◊ Set bOk = SOSWriteBoolean(GetObjectId(), 'bCheck', bCheck)
```

- Set bOk = SOSWriteString(GetObjectId(), 'strMessage', strMessage)
- Set bOk = SOSWriteNumberArray(GetObjectId(), 'anPrime', anPrime)
- Return bOk

♦ Function: ReadAttributes
- Description:
- ♦ Returns
  - ! TRUE if successful, else FALSE:
  - Boolean:
- Parameters
- Static Variables
- ♦ Local variables
  - Boolean: bOk
- ♦ Actions
  - Set bOk = fcAtomizable.ReadAttributes()
  - Set bOk = SOSReadBoolean(GetObjectId(), 'bCheck', bCheck)
  - Set bOk = SOSReadString(GetObjectId(), 'strMessage', strMessage)
  - Set bOk = SOSReadNumberArray(GetObjectId(), 'anPrime', anPrime)
  - Return bOk

! Public

♦ Function: Init
- Description:
- Returns
- Parameters
- Static Variables
- Local variables
- ♦ Actions
  - ! Call parent implementation
  - Call fcAtomizable.Init()
  - ! Set object type
  - Call SetObjectType('fcMyAtomizableClass')
  - ! Initialize attributes
  - Set bCheck = TRUE
  - Set strMessage = 'SOS Demo'
  - Set anPrime[1] = 2
  - Set anPrime[2] = 3
  - Set anPrime[3] = 5
  - Set anPrime[4] = 7
  - Set anPrime[5] = 11

♦ Function: Reset
- Description:
- Returns
- Parameters
- Static Variables
- Local variables
- ♦ Actions
  - ! Call parent implementation
  - Call fcAtomizable.Reset()
  - ! Reset attributes

◊ Set bCheck = FALSE
◊ Set strMessage = ''
◊ Set anPrime[1] = NUMBER_Null
◊ Set anPrime[2] = NUMBER_Null
◊ Set anPrime[3] = NUMBER_Null
◊ Set anPrime[4] = NUMBER_Null
◊ Set anPrime[5] = NUMBER_Null

♦ Function: Done
◊ Description:
◊ Returns
◊ Parameters
◊ Static Variables
◊ Local variables
♦ Actions
♦ If Alive()
◊ ! Call parent implementation
◊ Call fcAtomizable.Done()

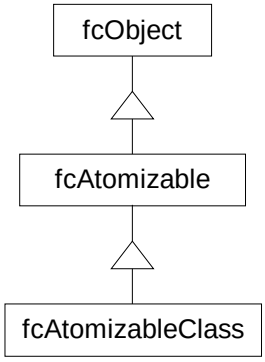


Abbildung 6.4: Vererbungsbaum von fcAtomizableClass

Wie obiger Vererbungsbaum zeigt, erbt fcAtomizableClass von der Klasse fcAtomizable, die wiederum von fcObject erbt.

In fcObject sind die abstrakten Funktionen Init(), Reset() und Done() deklariert, die in fcAtomizable implementiert werden und den Objekt-Lebenszyklus modellieren (vgl. Abschnitt 5.2.1.2). Init() wird bei der Objektinitialisierung aufgerufen, vergleichbar einem Konstruktor in C++ oder Java. Nach dem Aufruf der Init()-Implementierung der Elternklasse wird der Klassentyp gesetzt. Dieser muß im SOS bekannt sein, um die Objekte korrekt speichern und auslesen zu können. Anschließend werden die Attribute von fcAtomizable initialisiert. Die Reset()-Funktion „löscht" die Attributwerte, d.h. diese werden auf einen Initialwert gesetzt. Diese Funktion wird z.B. aufgerufen, bevor ein Objekt mit Werten aus dem SOS gefüllt wird. Die Funktion Done() entspricht schießlich einem Destruktor in C++ (bzw. einem Finalizer in Java).

Durch Erben von der Klasse fcAtomizable kann ein fcAtomizableClass-Objekt auf einfache Art und Weise in das SOS geschrieben und aus diesem ausgelesen werden. Dazu müssen lediglich die Funktionen WriteAttributes() und ReadAttributes() überschrieben werden; in diesen Funktionen ist

das Schreiben bzw. Lesen der einzelnen Objektattribute zu implementieren. Zu diesem Zweck stellt die SOS-Schnittstelle für jeden SQLWindows-Datentyp eine Schreib- und eine Lese-Funktion zur Verfügung (SOSWrite*() bzw. SOSRead*()), z.B. SOSReadString() zum Lesen von SQLWindows-Zeichenketten. Als erster Parameter dieser Funktionen ist die Objekt-ID zu übergeben, die über die (in fcObject definierte) Funktion GetObjectId() ermittelt werden kann. Die Objekt-ID wird automatisch beim Initialisieren des Objektes mittels Init() vergeben.

Durch Aufruf der (in fcAtomizable definierten) Funktionen WriteObject() und ReadObject() wird ein Objekt komplett in das SOS geschrieben bzw. aus diesem ausgelesen. Das Interaktionsdiagramm in der folgenden Abbildung 6.5 soll die Funktionsaufrufe beim Schreiben eines Objektes beispielhaft veranschaulichen. Nach dem Aufruf von WriteObject() wird zunächst der Schreibvorgang initialisiert; anschließend führt ein polymorpher Aufruf von WriteAttributes() dazu, daß die Attribute des Objektes in das SOS geschrieben werden. Die Kontrolle wird daraufhin an die Funktion WriteObject() zurückgegeben, welche den Schreibvorgang beendet. Analog dazu werden beim Lesen eines Objektes aus dem SOS die Funktionen ReadObject() und ReadAttributes() aufgerufen.

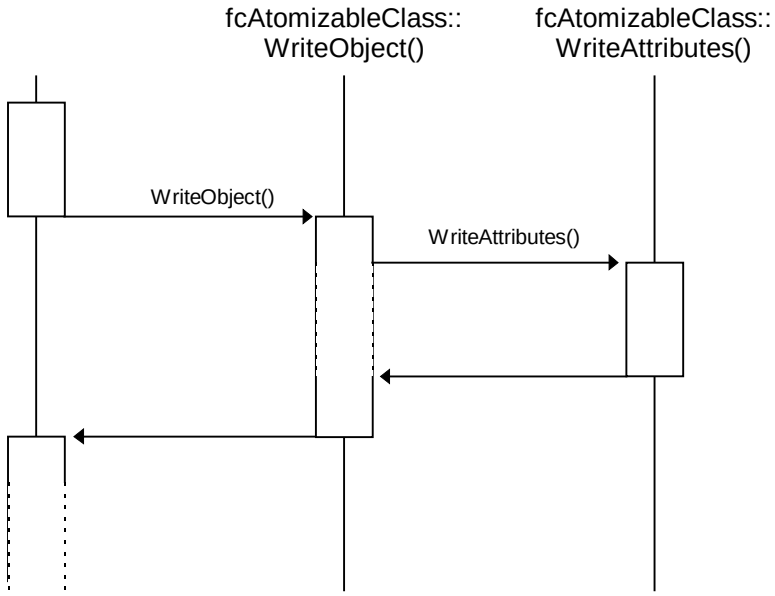


Abbildung 6.5: Interaktionsdiagramm für das Schreiben eines Objektes in das SOS

6.6 SOS intern

Dieser Abschnitt soll einen Einblick in die Interna des SQLWindows Object System geben. Dabei kommen sowohl technische Konzepte als auch Probleme bei der Implementierung zur Sprache. Dieser Abschnitt ist nicht notwendig, wenn auch hilfreich für das Verständnis des SOS. Leser, die lediglich an der Benutzung des SOS interessiert sind, können diesen Abschnitt getrost überspringen.

Neben der Beschreibung der implementierten „Interna“ des SOS werden auch mögliche Alternativen diskutiert und die Argumente genannt, die letztendlich zur gewählten Implementierung geführt haben. So bekommt der Leser einen direkten Einblick in die Entwicklung des SOS. Die Beschreibung des Entwurfsprozesses soll zu einem besseren Verständnis des Systems beitragen und eventuell in ähnlichen Projekten als Anhaltspunkt für die Entscheidungsfindung dienen.

Während des Entwurfs wurden viele Fehler und Unzulänglichkeiten des C++ - Compilers und des SQLWindows Entwicklungssystems aufgedeckt, die einen Teilentwurf nicht selten in einer Sackgasse enden ließen. Die Beschreibung dieser Probleme möge anderen Entwicklern solche „Irrwege“ ersparen.

Derartige Abschnitte programmiersprachlich-technischen Inhalts sind durch ein Schraubenschlüssel-Symbol am Seitenrand gekennzeichnet. Leser, die nur am Entwurf des SOS interessiert sind, können diese Abschnitte überspringen.

Als Entwicklungsumgebung für das SOS stand Microsoft Visual C++ Version 1.5 zur Verfügung. Dieser Compiler wird auch von Gupta für die DLL-Entwicklung empfohlen (vgl. [Gup94a]).

6.6.1 Interner Aufbau von SOS.DLL

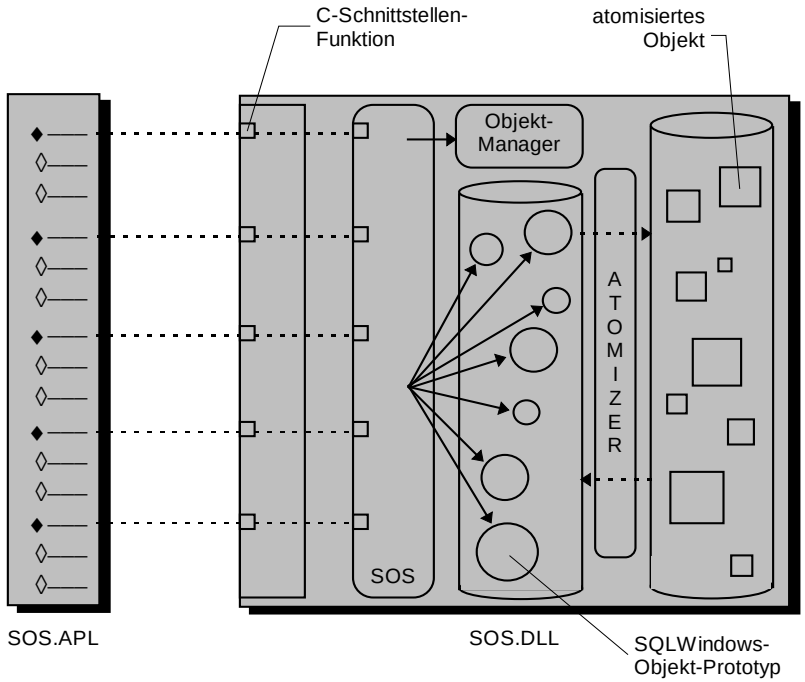


Abbildung 6.6: Interner Aufbau des SOS

`SOS.DLL` besteht aus mehreren Komponenten, wie Abbildung 6.6 zeigt.

Die Funktionen der C-Schnittstelle entsprechen in Namen, Parametern und Rückgabewerten den in `SOS.APL` deklarierten SQLWindows-Funktionen. In der Tat wird beim Aufruf einer Funktion aus `SOS.APL` die entsprechende C-Funktion in `SOS.DLL` aufgerufen.

Die Funktionen der C-Schnittstelle rufen ihrerseits die (bis auf das fehlende Suffix SOS gleichnamigen) Methoden eines globalen Objekts der Klasse SOS auf. Diese Klasse bildet den Kern des Systems. Der Grund für die doppelte Schnittstelle (Klasse SOS und C-Schnittstelle entsprechen

sich im Sinne der von ihnen angebotenen Dienste) ist technischer Natur: SQLWindows-APLs können nur mit C-Funktionen verbunden werden, und nicht mit Klassen.

Die Klasse SOS benutzt den Objekt-Manager und verwaltet eine Menge von SQLWindows-Objekt-Prototypen. Die Klasse SOS ist in Abschnitt 6.6.7 beschrieben.

Der Objekt-Manager dient der Verwaltung der Objekt-IDs. Er liefert eindeutige Objekt-IDs für neue Objekte und kann für Objekte, die bei der Erzeugung bereits eine ID besitzen (z.B. Objekte, die ihre Daten aus einer Datenbank mit einem numerischen Schlüsselattribut beziehen), eine Registrierung vornehmen, damit diese ID nicht nochmals vergeben wird. Bereits registrierte Objekte können auch wieder deregistriert werden. Der Objekt-Manager ist in Abschnitt 6.6.10 beschrieben.

Ein SQLWindows-Objekt-Prototyp ist die C++ - Repräsentation eines SQLWindows-Objekts. Diese Repräsentation ist für die Implementierung eines Referenzkonzeptes nicht unbedingt notwendig, erlaubt aber die Abstraktion der SQLWindows-Attribute aus der Sicht des Atomizers, wie in Abschnitt 6.3.2 beschrieben. Wendet sich ein SQLWindows-Objekt an das SOS, um sich speichern zu lassen, so wird geprüft, ob für den Typ dieses Objekts bereits ein Objekt-Prototyp existiert, und gegebenenfalls wird ein neuer Objekt-Prototyp erzeugt. Dieser Objekt-Prototyp nimmt dann die aktuellen Werte der Attribute des SQLWindows-Objektes auf und wird anschließend atomisiert, d.h. vom Atomizer in eine sprachunabhängige Beschreibungsform überführt. Fordert nun eine SQLWindows-Applikation ein atomisiertes Objekt an, so wird zunächst der SQLWindows-Objekt-Prototyp mit den Attributwerten des Objekts aus dem Atomizer gefüllt – der Prototyp wird zum „echten“ (C++ -) Objekt, das über entsprechende SOS-Funktionen in das anfragende SQLWindows-Objekt kopiert werden kann. Dieser Mechanismus wird in Abschnitt 6.6.3 anhand eines Beispiels erläutert. Eine ausführliche Beschreibung der SQLWindows-Objekt-Prototypen findet man in Abschnitt 6.6.8.

Die atomisierten SQLWindows-Objekte werden in der vorliegenden Atomizer-Implementation durch ein Repository repräsentiert, das vom Aufbau her einer Windows-Initialisierungsdatei entspricht. Der Atomizer könnte somit auch als Persistenzmechanismus für SQLWindows-Objekte genutzt werden. In der Regel werden die Objekte aber (in Form von Tabellen) in Datenbanken gespeichert. Konzept und Implementierung des Atomizers werden in Abschnitt 6.6.11 näher beschrieben.

6.6.2 SOS-Klassendiagramm

Das Klassendiagramm in Abbildung 6.7 soll den inneren Aufbau des SOS und die Beziehungen der SOS-Komponenten untereinander auf Klassenebene verdeutlichen:

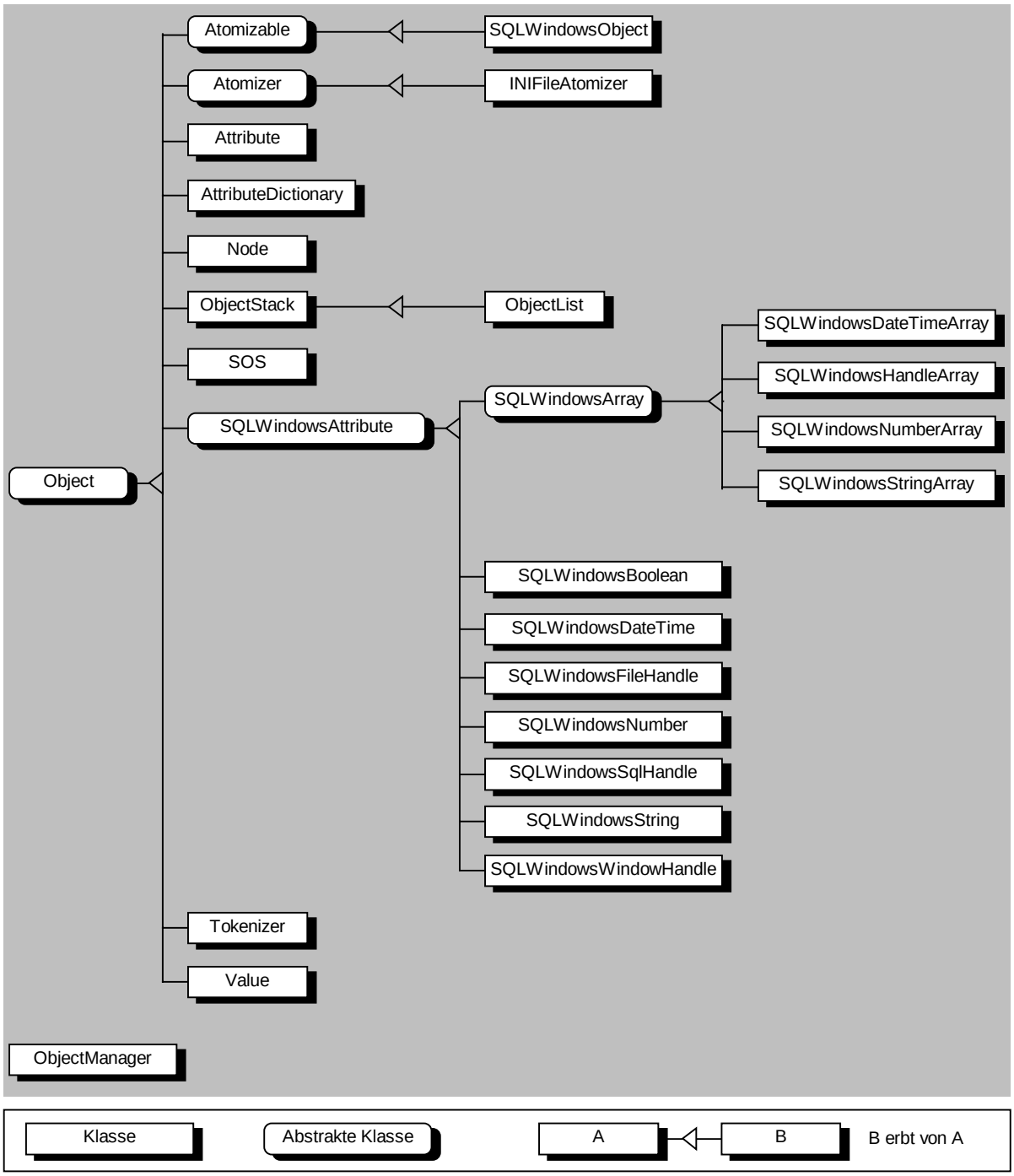


Abbildung 6.7: Klassendiagramm des SOS

6.6.3 Die Atomisierung eines SQLWindows-Objektes

Bevor in den folgenden Abschnitten der Entwurf und die Implementierung der einzelnen Komponenten des SOS detailliert beschrieben werden, soll zunächst anhand eines Beispiels der Vorgang der Atomisierung eines SQLWindows-Objektes schrittweise erläutert werden. Die SOS-Funktionsbeschreibung und die Beschreibung der SQLWindows-Klasse fcAtomizable in Anhang B erläutern die Vorgehensweise bei der Übergabe eines SQLWindows-Objekts an das SOS bzw. bei der Rückgabe der Attribute vom SOS. Dies geschieht über SQLWindows-Funktionen, die im SOS (genauer: in der Datei SOS.DLL) ein in C implementiertes Pendant besitzen. In diesem Beispiel soll ein einfaches SQLWindows-Objekt vom Typ fcMyClass mit nur einem Attribut vom Typ Number betrachtet werden. fcMyClass erbt von fcAtomizable. Wie bereits in Abschnitt 6.5 erläutert, stellt fcAtomizable die Grundfunktionalität für die Benutzung des SOS zur Verfügung.

- ♦ Functional Class: fcMyClass
  - ◊ Description: This class can be written to and read from the SOS
  - ♦ Derived From
    - ◊ Class: fcAtomizable
  - ◊ Class Variables
  - ♦ Instance Variables
    - ◊ Number: nMyNumber
  - ♦ Functions
    - ♦ Function: Init
    - ♦ Function: Reset
    - ♦ Function: Done
    - ♦ Function: WriteAttributes
    - ♦ Function: ReadAttributes

Die einzelnen Funktionen sind ausgeblendet. Die Darstellung ihrer Implementierung wird bis zur Beschreibung ihrer ersten Verwendung (d.h. ihres ersten Aufrufs) aufgeschoben. Die Funktionen Init(), Reset() und Done() sind in der Klasse fcObject definiert und werden in diser Klassen überschrieben und implementiert.

Initialisiert wird ein Objekt der Klasse fcMyClass durch Aufruf der Methode Init():

- ♦ Function: Init
  - ♦ Actions
    - ◊ Call fcAtomizable.Init()
    - ◊ Call SetObjectType(fcMyClass)
    - ◊ Set nMyNumber = 42

Zunächst wird die Basisklassen-Implementierung von Init() aufgerufen. Dort wird durch Aufruf der Funktion SOSGetObjectId() unter Benutzung des Objekt-Managers eine eindeutige ID ermittelt, wie in Abbildung 6.8 dargestellt. Das fcMyClass-Objekt kann nun eindeutig identifiziert werden. Mit SetObjectType() wird der Objekttyp der Klasse fcMyClass gesetzt.

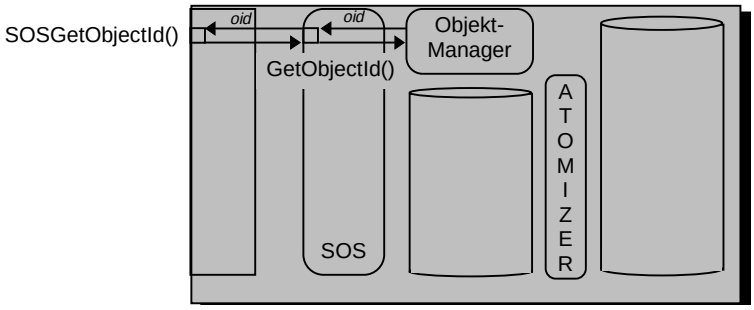


Abbildung 6.8: Aufruf von SOSGetObjectId()

Soll ein Objekt der Klasse fcMyClass in das SOS geschrieben werden, so ruft man dessen (in fcAtomizable implementierte) Funktion WriteObject() auf. Diese Funktion führt nun folgende Aktionen durch:

Zunächst wird mittels SOSBeginWrite() das Schreiben eines SQLWindows-Objektes in das SOS eingeleitet. BeginWrite() ruft SOSBeginWrite() mit dem Objekttyp als zusätzlichen Parameter auf. Diese ruft wiederum die Methode BeginWrite() in der Klasse SOS. Diese prüft, ob für den Typ des übergebenen Objektes bereits ein SQLWindows-Objekt-Prototyp erzeugt wurde, erzeugt diesen gegebenenfalls und initialisiert ihn mit der Objekt-ID des übergebenen SQLWindows-Objektes, wie Abbildung 6.9 schematisch zeigt:

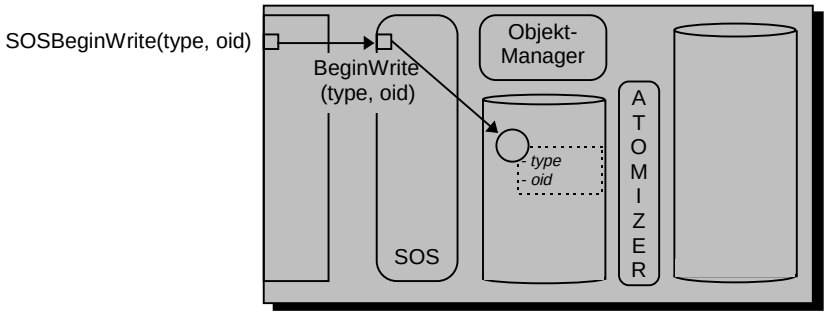


Abbildung 6.9: Aufruf von SOSBeginWrite()

Ist der Aufruf dieser Funktion erfolgreich, so wird polymorph die Funktion WriteAttributes() aufgerufen, die in der Klasse fcMyClass wie folgt implementiert ist:

- ♦ Function: WriteAttributes
  - ◊ Description:
  - ♦ Returns
    - ◊ ! TRUE if successful, else FALSE:
    - ◊ Boolean:
  - ◊ Parameters
  - ◊ Static Variables
  - ♦ Local variables
    - ◊ Boolean: bOk
  - ♦ Actions
    - ◊ Set bOk = fcAtomizable.WriteAttributes()
    - ◊ Set bOk = SOSWriteNumber(GetObjectId(), 'nMyNumber', nMyNumber)
    - ◊ Return bOk

Nach dem Aufruf der Basisklassen-Implementierung muß jedes Attribut einzeln unter Angabe von Objekt-ID, Attributnamen und Attributwert an das SOS übergeben werden. In unserem Beispiel wird das Number-Attribut nMyNumber mittels SOSWriteNumber() geschrieben (vgl. Abbildung 6.10).

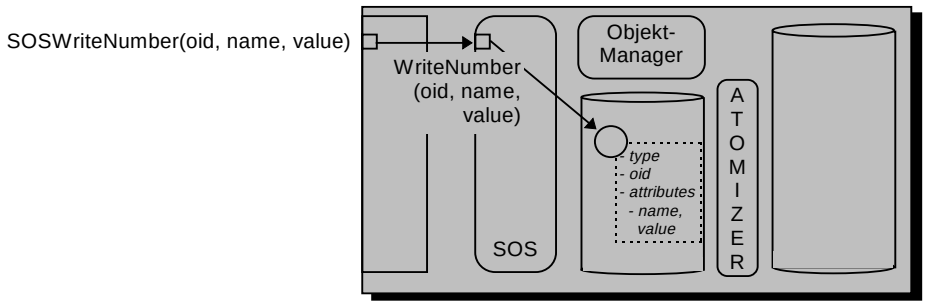


Abbildung 6.10: Aufruf von SOSWriteNumber()

Der SQLWindows-Objekt-Prototyp führt seine Attribute in einer Liste, die um das neue Attribut ergänzt wird.

Schließlich beendet WriteObject() den Schreibvorgang durch Aufrufen von EndWrite() (das wiederum SOSEndWrite() aufruft). Dadurch wird die Methode EndWrite() der Klasse SOS gerufen. Diese übergibt den SQLWindows-Objekt-Prototyp an den Atomizer, der diesen atomisiert, wie in Abbildung 6.11 dargestellt.

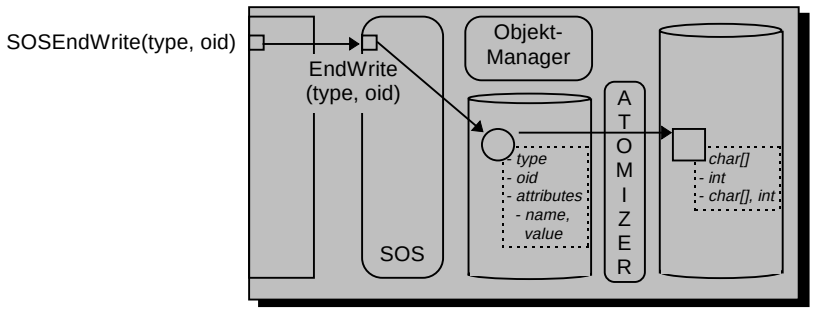


Abbildung 6.11: Aufruf von SOSEndWrite ()

Ähnlich verhält es sich mit dem Zurücklesen von SQLWindows-Objekten aus dem SOS, die mit der in fcAtomizable definierten Funktion ReadObject() durchgeführt wird.

6.6.4 Repräsentation der SQLWindows-Datentypen in C

Bei der Deklaration von externen Funktionen in SQLWindows werden für den Rückgabewert und jeden Funktionsparameter jeweils zwei Datentypen angegeben:

Der *interne Datentyp* (*internal data type*) ist einer der Standard-SQLWindows-Datentypen. Er dient dem SQLWindows-Compiler zur Typprüfung beim Aufruf einer externen Funktion aus einer SQLWindows-Applikation heraus.

Der *externe Datentyp* (*external data type*) definiert den Datenaustausch zwischen SQLWindows und der DLL. Er legt das Format fest, mit dem die Funktionsparameter an die DLL übergeben werden bzw. das der Rückgabewert einer DLL-Funktion hat. Externe Datentypen sind:

- alle Standard-Datentypen der Programmiersprache C, wie z.B. int oder char
- skalare Windows-Datentypen, wie z.B. LONG, INT, DWORD oder HWND (definiert in `WINDOWS.H`)
- externe SAL-Datentypen, wie z.B. HSTRING oder HARRAY (definiert in `SWTYPE.H`)
- aus den obigen Datentypen zusammengesetzte SAL-Strukturen in C, wie z.B. NUMBER oder DATETIME (definiert in `SWTYPE.H`)

Ein Beispiel: Die Funktion SOSBeginWrite ist in `SOS.APL` wie folgt definiert:

```
♦Function: SOSBeginWrite
    ◊ Description: Starts the writing process for a SQLWindows object; in fact, this
    ◊              just starts the construction of a C++ object representation for the
    ◊              SQLWindows object, which itself is written to the atomizer after
    ◊              SOSEndWrite() has been called.
    ◊ Export Ordinal: 0
    ♦Returns
        ◊ ! TRUE if successful, FALSE otherwise.
        ◊ Boolean: BOOL
    ♦Parameters
        ◊ ! The object's type
        ◊ String: LPSTR
        ◊ ! The object's id
        ◊ Number: DWORD
```

Der zweite Parameter der Funktion ist die Objekt-ID. Diese wird innerhalb von SQLWindows als Number repräsentiert, während sie in `SOS.DLL` vom Typ DWORD ist, dem von Windows definierten Datentyp für vorzeichenlose lange Ganzzahlen (unsigned long).

Werte können zwischen SQLWindows und einer DLL als Werte (by value) oder als Verweise (by reference) übergeben werden. Letzteres ist z.B. nötig, um SQLWindows-Strings oder ganze SQLWindows-Arrays an eine DLL übergeben zu können. Dazu stehen entsprechende externe Datentypen zur Verfügung, die (segmentübergreifende) 16-bit-Pointer (FAR-Pointer) oder *Bezüge* (*handles*) definieren. Eine genauere Beschreibung der Verweissemantik erfordert detailliertes Wissen über das Speichermodell von Intel 80x86-Prozessoren, so daß in dieser Arbeit darauf verzichtet werden soll.

Mit der doppelten Datentypisierung gelingt der Übergang von SQLWindows zu C bzw. C++ in der DLL sehr gut, da sich weder der SQLWindows- noch der DLL-Entwickler Gedanken über die Konvertierung zwischen den beiden „Welten“ machen müssen. Es genügt, sinnvolle Paare von internen und externen Datentypen festzulegen – die eigentliche Konvertierung erledigt der SQLWindows-Compiler. Dennoch bleibt ein Problem: Die Verwendung von Windows-Datentypen bindet – neben der Implementierung als Windows Dynamic Link Library (DLL) – das SOS an das Windows-Betriebssystem. Im SOS-Code sind alle Stellen kommentiert, an denen von Windows-Funktionen Gebrauch gemacht wird. Da sich sowohl die Anzahl dieser Stellen als

auch die Anzahl der benutzten Windows-Funktionen in Grenzen hält, könnte das SOS relativ schnell in eine neue Betriebssystemumgebung übertragen werden.

Streng genommen handelt es sich nicht um eine doppelte, sondern um eine dreifache Datentypisierung, da der Atomizer ein Windows-unabhängiges Datenformat verwendet, so daß die Attribute bei der Atomisierung in Standard-Datentypen von C++ übertragen werden müssen. Tabelle 6.1 listet alle verwendeten Typrelationen auf.

SQLWindows-Typ	SOSWrite*()	SOSRead*()	SOS	Atomizer
Boolean	BOOL	LPBOOL	BOOL	unsigned short
Date/Time	DATETIME	LPDATETIME	DATETIME	char[ ]
Number	NUMBER	LPNUMBER	NUMBER	char[ ]
String	LPSTR	LPHSTRING	char[ ]	char[ ]
Long String	–	–	–	–
FileHandle	HFILE	LPHFILE	HFFILE	unsigned short
SqlHandle	HSQLHANDLE	LPHSQLHANDLE	SQLHANDLENUMBER	int
WindowHandle	HWND	LPHWND	HWND	unsigned int
Date/Time[*]	HARRAY	HARRAY	Value[ ]	char[ ]
Number[*]	HARRAY	HARRAY	Value[ ]	char[ ]
String[*]	HARRAY	HARRAY	Value[ ]	char[ ]
SqlHandle[*]	HARRAY	HARRAY	Value[ ]	char[ ]
WindowHandle[*]	HARRAY	HARRAY	Value[ ]	char[ ]

Tabelle 6.1: Datentyp-Relationen

SOSWrite*() und SOSRead*() stehen für die in `SOS.APL` definierten Schreib- und Lesefunktionen, z.B. SOSWriteBoolean() für boolesche Werte. [*] steht für Arrays, char[ ] für C-Zeichenketten. Value[ ] bezeichnet eine Liste von Objekten der Klasse Value, die einfache Werte repräsentiert.

Die Spalten führen die in den unterschiedlichen Kontexten/Umgebungen definierten Datentypen auf:

- **SQLWindows-Typ**: Der in SQLWindows definierte (interne) Datentyp. Für den Datentyp Long String gibt es keine Möglichkeit der Übergabe an eine DLL. Bei Betrachtung der Array-Typen fällt auf, daß Arrays für die einfachen Typen Boolean und File Handle fehlen. Diese lassen sich zwar in SQLWindows definieren, aber für sie ist kein externer Datentyp deklariert, der eine Übergabe an eine DLL erlaubte.
- **SOSWrite*()**: Der (externe) Datentyp, in den ein SQLWindows-Datentyp (vom SQLWindows-Compiler) bei der Übergabe in das SOS (genauer: `SOS.DLL`) konvertiert wird.
- **SOSRead*()**: Der (externe) Datentyp, aus dem ein SQLWindows-Datentyp (vom SQLWindows-Compiler) bei der Rückgabe vom SOS (genauer: `SOS.DLL`) konvertiert wird; die Übergabe geschieht per Referenz (das Präfix LP bezeichnet einen FAR-Zeiger auf den entsprechenden Datentyp), so daß die einfachen Datentypen in SOSRead...() als Receive-Datentypen deklariert werden müssen (Arrays werden immer per Referenz übergeben).

- **SOS**: Der Datentyp, durch den die SQLWindows-Datentypen innerhalb des SOS repräsentiert werden. Dabei ergab sich im Falle der SQLWindows-Datentypen File Handle und Sql Handle das Problem, daß die äquivalenten C-Datentypen HFILE und HSQLHANDLE (bzw. LPHFILE und LPHSQLHANDLE) nicht definiert waren, so daß „ähnliche“, in der C-Schnittstelle von SQLWindows definierte Datentypen (HFFILE und SQLHANDLENUMBER) verwendet wurden.
- **Atomizer**: Der Standard-C-Datentyp, der zur Speicherung im Atomizer verwendet wird. Dieser entspricht dem C-Datentyp, der einem im SOS verwendeten Windows-Datentyp wie HWND oder BOOL zugrunde liegt.

Die SQLWindows-Datentypen Number und Date/Time werden SQLWindows-intern in einem sehr speziellen Format gespeichert. Dementsprechend bilden die Typdefinitionen von NUMBER und DATETIME nicht auf Standard-C-Datentypen ab, sondern definieren C-Strukturen. Diese müssen zur Speicherung in einfache C-Datentypen umgewandelt werden:

Über die SQLWindows-Funktion SalDateToStr() kann ein DATETIME-Wert in einen SQLWindows-String (HSTRING) umgewandelt werden. Dieser kann dann in eine C-Zeichenkette kopiert werden. Für die Rückkonvertierung wird das Verfahren in der umgekehrten Reihenfolge angewendet, wobei anstatt von SalDateToStr() nun die Funktion SalStrToDate() benutzt wird.

Alle numerischen Daten – sowohl Fest-, als auch Fließkommazahlen – werden im SQLWindows-Datentyp Number als Fließkommazahl zur Basis 100 mit maximal 22 Stellen in einem speziellen Byteformat gespeichert (vgl. [Gup94d]). Der verwendete Microsoft C++-Compiler (16 Bit) bietet keinen Datentyp mit vergleichbarem Wertebereich bzw. gleicher Genauigkeit. In einer ersten Implementierung wurde der Number-Wert in eine Fließkommazahl vom Typ double (15 Ziffern; Wertebereich: 1.7E ± 308) umgewandelt und dem Atomizer übergeben. Da die Genauigkeit mit 15 Ziffern jedoch unter der des Datentyps Number liegt, werden in der jetzigen Implementierung die einzelnen Bytes des Number-Byteformats als Zeichenkette gespeichert, so daß der Wertebereich von Number erhalten bleibt. Damit sind die Number-Werte im Atomizer jedoch nicht ohne weiteres von einem C++ - Programm aus lesbar.

6.6.5 Repräsentation der SAL-Funktionen in C

Von der DLL aus kann jede SAL-Funktion aufgerufen werden. Dies ist insbesondere im Falle der Funktionen für Number-Datentypen (SalNumber*), DateTime-Datentypen (SalDate*) und Array-Funktionen nützlich, da erst diese Funktionen die umfassende Behandlung und Konvertierung der SQLWindows-Datentypen in der DLL erlauben. Aber auch jede andere SAL-Funktion kann von der DLL aus aufgerufen werden. Dazu wird die Bibliothek `SWIN.LIB` (statisch) zur DLL hinzugebunden. Zur Laufzeit der DLL laden die Funktionen in `SWIN.LIB` eine weitere DLL nach – `SWIN*.DLL`, wobei der Asterisk '*' für die verwendete SQLWindows-Version steht (in diesem Falle `SWIN50.DLL`).

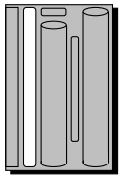
In `SWIN*.DLL` sind auch Funktionen definiert, die nur an der C-Schnittstelle, nicht aber in SQLWindows selbst zur Verfügung stehen. Dabei handelt es sich im wesentlichen um Datentyp-Konvertierungsfunktionen (SWinCvt*), Array-Zugriffsfunktionen (SWinArray*) und Funktionen zur String-Bearbeitung.

6.6.6 Versionskontrolle

Wie bereits erwähnt, deklariert die SQLWindows-Bibliothek SOS.APL alle von SOS.DLL exportierten Funktionen für die Benutzung in SQLWindows-Applikationen. Dabei muß gewährleistet sein, daß die Funktionsdeklarationen in SOS.APL denen in SOS.DLL bezüglich Typ und Anzahl der Parameter und Typ des Rückgabewertes übereinstimmen. Dies soll durch eine Versionskontrolle erreicht werden: Sowohl SOS.APL als auch SOS.DLL führen eine interne Versionsnummer, die im Falle von SOS.DLL über eine (exportierte) Funktion abgefragt werden kann.

SOS.APL definiert eine interne Funktion[14] SOSInit(), die von der SQLWindows-Applikation vor der Benutzung des SOS aufgerufen werden muß. Beim Initialisieren des SOS mittels SOSInit() wird ein Versionsvergleich durchgeführt. Schlägt dieser fehl, sind die Versionen von SOS.APL und SOS.DLL also verschieden, dann wird eine entsprechende Meldung ausgegeben, und die SQLWindows-Applikation wird beendet. Diese harte Form der Fehlerbehandlung ist durchaus gerechtfertigt, wenn man bedenkt, welchen Schaden falsche Rückgabewerte, die aus Versionsunterschieden resultieren können, an den Daten und der Konsistenz der Datenbanken anrichten kann, auf denen SQLWindows-Applikationen üblicherweise arbeiten.

6.6.7 Die Klasse SOS



Wie aus Abbildung 6.6 ersichtlich und bereits kurz erwähnt, rufen die Funktionen der C-Schnittstelle die (bis auf das fehlende Suffix SOS gleichnamigen) Methoden eines globalen Objekts der Klasse SOS auf. Die C-Schnittstellenschicht ist notwendig, um die Anbindung der DLL an die APL zu ermöglichen, welche die Schnittstelle zu SQLWindows bildet und nur auf C-Funktionen zugreifen kann. Im Sinne eines durchgängigen objektorientierten Entwurfs wurde die Schnittstelle zunächst als Klasse entworfen – eben jene Klasse SOS.

Die C-Schnittstellenfunktionen greifen auf eine globale Instanz der Klasse SOS zu. Da es nur eine solche Instanz zur Laufzeit geben darf, wäre es sinnvoll, die Klasse SOS als Singleton zu implementieren (siehe [GHJV95] für eine Beschreibung dieses Entwurfsmusters). Dies führte jedoch zu Problemen, die während der Entwicklung des SOS bereits in einem anderen Zusammenhang aufgetreten waren:

Ursprünglich sollten die C-Schnittstellenfunktionen direkt den Objekt-Manager benutzen. Da die Klasse ObjectManager als Singleton implementiert ist, muß ein Kunde dieser Klasse deren Methode Instance() aufrufen, um einen Verweis auf die (einzige) Instanz der Klasse zu erhalten:

```
ObjectManager* poObjectManager;

poObjectManager = ObjectManager::Instance();
```

[14] Es soll an dieser Stelle noch einmal erwähnt werden, daß der Begriff „interne Funktion“ im SQLWindows-Sprachgebrauch eine Funktion bezeichnet, deren Implementierung in SQLWindows selbst und nicht in einer externen DLL geschieht. Der Sichtbarkeitsbereich der Funktion ist die gesamte Applikation, und nicht etwa nur die Bibliothek SOS.APL, wie man angesichts des Begriffs „intern“ annehmen könnte.

Dazu wird bei jedem Aufruf von ObjectManager::Instance() geprüft, ob das Attribut m_Instance, ein Verweis auf ObjectManager, bereits initialisiert ist:

```
ObjectManager* ObjectManager::Instance()
{
     if (m_Instance == NULL)
     {
          m_Instance = new ObjectManager;
     }
     return m_Instance;
}
```

Dieses Attribut wird global mit NULL initialisiert:

```
ObjectManager* ObjectManager::m_Instance = NULL;
```

Erstaunlicherweise wird diese Initialisierung in der DLL nicht ausgeführt, so daß nie eine Instanz von ObjectManager erzeugt wird, wenn in einer der C-Schnittstellen-Funktionen die Instance()-Methode aufgerufen wird. Ruft man diese Methode jedoch in der Methode einer Klasse auf, so wird die ObjectManager-Instanz korrekt erzeugt. Aus diesem Grund wurde die ObjectManager-Schnittstelle in die Klasse SOS integriert.

Das Problem mit dem ObjectManager-Singleton rührt wahrscheinlich von der gemischten Benutzung von C- und C++ - Code her. Historisch bedingt werden DLLs vorrangig in C geschrieben und bieten auch nur eine C-Schnittstelle. Für die C++ - Compiler von Microsoft erweiterte man die DLLs dahingehend, daß sie auch C++ - Klassen exportieren können. Da SQLWindows jedoch eine C-Schnittstelle erwartet, das SOS intern aber objektorientiert in C++ entworfen wurde, kam es zu besagter Mischung aus C und C++ und dem genannten Problem.

Der durchgängig objektorientierte Entwurf mit der Klasse SOS hat einen bisher noch nicht genutzen Vorteil: Durch Exportieren dieser Klasse können C++ - Programme Instanzen von SOS erzeugen und sind somit nicht auf die C-Schnittstelle angewiesen. Zu diesem Zweck muß die Klassendeklaration um das Schlüsselwort __export erweitert werden:

```
class __export SOS
{...
};
```

Da dieses Schlüsselwort jedoch Compiler-spezifisch ist, ließe sich diese Klasse nicht mehr mit einem anderen als dem Microsoft C++ - Compiler übersetzen. Deshalb wurde (in einer separaten Header-Datei) folgende Definition eingeführt:

```
#define EXPORT __export
```

Die Klassendefinition sieht dann folgendermaßen aus:

```
class EXPORT SOS
{...
};
```

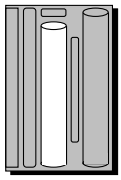
Möchte man die exportierten Klassen mit einem Compiler übersetzen, der das Schlüsselwort __export nicht kennt, so genügt es, die Definition aufzuheben:

```
#define EXPORT
```

Auf diese Weise muß nur eine Datei (die Header-Datei) geändert werden, um die Exportdefinition in allen exportierten Klassen aufzuheben, und der Entwurf wird dadurch nicht beeinflußt.

Neben der zu exportierenden Klasse – in diesem Falle die Klasse SOS – müssen auch all jene Klassen exportiert werden, die als Attribut der zu exportierenden Klasse auftauchen. Des weiteren sind alle Basisklassen der zu exportierenden Klassen ebenfalls zu exportieren.

6.6.8 Die SQLWindows-Objekt-Prototypen



Die SQLWindows-Objekt-Prototypen sind das C++ - Abbild der zu atomisierenden SQLWindows-Objekte. Der Begriff „Prototyp“ soll andeuten, daß nicht für jedes einzelne SQLWindows-Objekt ein C++ - Äquivalent angelegt wird, sondern nur für jede Klasse zu atomisierender SQLWindows-Objekte. Somit könnte man auch von SQLWindows-Klassen-Prototypen sprechen. Es wurde der erste Begriff gewählt, da die Prototypen zur Laufzeit mit den Daten von SQLWindows-Objekten (aus der SQLWindows-Applikation oder dem Atomizer) „gefüllt“ werden und zu diesem Zeitpunkt einem Objekt entsprechen. Den Objektcharakter tragen sie jedoch nur so lange, bis sie auf ein „echtes“ SQLWindows-Objekt übertragen werden oder aber atomisiert werden. Anschließend werden alle Attribute (über die Methode Reset()) auf einen voreingestellten Wert zurückgesetzt – aus dem Objekt wird wieder ein Prototyp.

Anzumerken ist, daß alle Instanzen von SQLWindows-Objekt-Prototypen denselben statischen Typ haben, auch wenn sie SQLWindows-Objekte unterschiedlichen Typs darstellen. Unterschiede lassen sich erst im „Inneren“ der Objekte feststellen, wie wir bald sehen werden. Außerdem soll angemerkt werden, daß die SQLWindows-Objekt-Prototypen vom SOS bereits „atomisierte“ SQLWindows-Objekte übergeben bekommen – „atomisiert“ in dem Sinne, als daß das SQLWindows-Objekt Attribut für Attribut an den SQLWindows-Objekt-Prototypen übergeben wird.

Technisch gesehen sind SQLWindows-Objekt-Prototypen Instanzen der (sehr allgemeinen) Klasse SQLWindowsObject, die lediglich einen Objekttyp und eine Objekt-ID als Attribute besitzt und eine Liste von Attributen verwaltet.

Ein neu erzeugter SQLWindows-Objekt-Prototyp hat zunächst keine Attribute. Diese werden vom SOS nach und nach hinzugefügt, wenn ein SQLWindows-Objekt, zu dem dieser neue Prototyp erzeugt wurde, seine Attribute nacheinander an das SOS übergibt. Die Attribute werden charakterisiert durch

- einen Typ; es stehen alle Standard-SQLWindows-Datentypen zur Auswahl
- einen eindeutigen Namen, der normalerweise dem Attributbezeichner des „echten“ SQLWindows-Objektes entspricht
- einen Wert

Repräsentiert werden Attribute durch Instanzen von abgeleiteten Klassen der Klasse SQLWindowsAttribute (vgl. Klassendiagramm, Abbildung 6.7). Jede SQLWindowsObject-Instanz verwaltet eine Liste solcher Attribut-Objekte. Das bedeutet, daß alle SQLWindowsObject-Instanzen denselben statischen Typ haben, auch wenn sie SQLWindows-Objekte unterschiedlichen Typs dar-

stellen. Sie unterscheiden sich nur intern im Wert ihrer Attribute (Objekttyp und Objekt-ID) sowie in Anzahl und Wert der verwalteten Attribut-Objekte.

Die SQLWindowsAttribute-Subklassen „wissen“ auch, wie sie sich selbst atomisieren können, so daß die Methode SQLWindowsObject::WriteTo(), die zum Atomisieren eines SQLWindows-Objekt-Prototypen dient, die eigentliche Arbeit des Atomisierens einfach an die Attribute delegiert:

```
void SQLWindowsObject::WriteTo(Atomizer* poAtomizer) const
{
     SQLWindowsAttribute* poAttribute;

     poAttribute = (SQLWindowsAttribute*)(m_pAttributes->GetFirst());
     while (poAttribute != NULL)
     {
          poAttribute->WriteTo(poAtomizer);
          poAttribute = (SQLWindowsAttribute*)(m_pAttributes-
>GetNext());
     }
}
```

Das funktioniert natürlich nur, weil poAttribute per dynamischem Binden ein Objekt einer von SQLWindowsAttribute abgeleiteten konkreten Attributklasse zugewiesen wird, in der die WriteTo()-Methode entsprechend den Anforderungen des Attributs überschrieben ist.

Ähnlich verhält es sich mit dem Zurücklesen atomisierter Objekte. Die Klasse SOS fordert einen „passenden“ SQLWindows-Objekt-Prototypen an und weist ihm die Objekt-Id des zurückzulesenden Objektes zu. Der SQLWindows-Objekt-Prototyp weist nun jedes seiner Attribute an, den entsprechenden Attributwert aus dem Atomizer auszulesen:

```
     SQLWindowsAttribute* poAttribute;

     poAttribute = (SQLWindowsAttribute*)(m_pAttributes->GetFirst());
     while (poAttribute != NULL)
     {
          poAttribute->ReadFrom(poAtomizer);
          poAttribute = (SQLWindowsAttribute*)
                        (m_pAttributes->GetNext());
     }
```

6.6.9 Die SQLWindows-Attribute

Jedes Attribut eines SQLWindows-Objekt-Prototypen wird als eigenes Objekt repräsentiert. Für jeden SQLWindows-Datentyp gibt es eine eigene Klasse. All diese Klassen erben von SQLWindowsAttribute, dessen Schnittstelle wie folgt aussieht:

```
class SQLWindowsAttribute : public Object
{
public:
// Methods
     // Object interface.
     //
     virtual char* GetObjectType() = 0;
```

```
    // functions to let the attribute atomize itself
    //
    virtual void ReadFrom(Atomizer* poAtomizer) = 0;
    virtual void WriteTo(Atomizer* poAtomizer) = 0;

    // Reset value.
    //
    virtual void Reset() = 0;


    // Set name.
    //
    void SetName(const char* pcName);

    // Retrieve name.
    //
    char* GetName();
};
```

Die Methoden ReadFrom() und WriteTo() werden vom SQLWindows-Objekt-Prototypen gerufen, damit das Attribut seinen Wert aus dem Atomizer ausliest bzw. in diesen schreibt. Diese Methoden entsprechen zwar den in der Klasse Atomizable definierten Funktionen, aber dennoch erben SQLWindows-Attribute nicht von Atomizable. Der Grund ist, daß sie nie eigenständig atomisiert werden können, sondern nur im Kontext eines SQLWindows-Objekts, dessen Namen sie über die Methode SetName() gesetzt bekommen und über GetName() abfragen können.

Als Beispiel einer konkreten SQLWindows-Attributklasse für einen einfachen Datentyp soll der SQLWindows-Datentyp Date/Time dienen. Die Klasse SQLWindowsDateTime definiert – wie jede konkrete SQLWindows-Attributklasse – zwei zusätzliche Methoden an der Schnittstelle:

```
    // set attribute value.
    //
    void SetValue(DATETIME dtValue);

    // retrieve attribute value.
    //
    DATETIME GetValue();
```

Über diese Methoden kann der SQLWindows-Objekt-Prototyp die Attributwerte setzen und auslesen. Konzeptuell gehören diese beiden Methoden zwar in die Oberklasse SQLWindowsAttribute, aber da sie für jedes Attribut unterschiedliche Datentypen haben, können diese Methoden erst in den konkreten Attributklassen deklariert werden.

Die Methoden ReadFrom() und WriteTo() sind in dieser Klasse wie folgt implementiert:

```
void SQLWindowsDateTime::ReadFrom(Atomizer* poAtomizer)
{
    // read string representation of DateTime value from atomizer
    // and convert string to DateTime
    m_dtValue = SalStrToDate(poAtomizer->ReadCharPtr(GetName()));
}
```

SalStrToDate() ist eine SQLWindows-Funktion, die eine C-Zeichenkette in ein DATETIME-Konstrukt umwandelt (zur Nutzung von SQLWindows-Funktionen in C - oder C++ - Programmen siehe auch Abschnitt 6.6.5).

```
void SQLWindowsDateTime::WriteTo(Atomizer* poAtomizer)
{
     HSTRING  hDateTimeString;
     LPSTR    lpszDateTimeString;
     long     lStringLength;

     hDateTimeString = 0;
     // associate the variable with an actual HSTRING value
     SWinInitLPHSTRINGParam(&hDateTimeString, 128L);

     // convert DateTime to string
     SalDateToStr(m_dtValue, &hDateTimeString);
     // lock string in memory
     lpszDateTimeString = SWinHStringLock(hDateTimeString,
                                          &lStringLength);
     // write string representation of DateTime value to atomizer
     poAtomizer->WriteCharPtr(GetName(), lpszDateTimeString);
     // unlock
     SWinHStringUnlock(hDateTimeString);
}
```

Die SQLWindows-Funktion SalDateToStr() wandelt einen DATETIME-Wert nicht in eine C-Zeichenkette, sondern in einen SQLWindows-String vom Typ HSTRING um. Dieser muß mittels SWinHStringLock() im Hauptspeicher fixiert werden, um auf seinen Inhalt über einen C-Zeichenketten-Zeiger (char*) zugreifen zu können. Durch SWinHStringUnlock() wird diese Fixierung wieder aufgehoben.

Wie aus Tabelle 6.1 ersichtlich, werden Arrays aller (als Array speicherbaren) SQLWindows-Datentypen in Form von Zeichenketten atomisiert. Dies liegt an der besonderen Behandlung von Arrays in den entsprechenden Attributklassen: Alle Array-Attributklassen erben von SQLWindowsArray. Diese Abstraktion ist möglich, da alle Array-Datentypen vom Typ HARRAY sind. Die Array-Elemente können über die SWinArrayGet*()-Funktionen ausgelesen werden (z.B. SWinArrayGetDateTime() für Date/Time-Arrays).

SQLWindowsArray bietet dieselbe Schnittstelle wie die „einfachen“ Attributklassen, definiert aber zwei zusätzliche abstrakte Methoden:

```
// retrieve HSTRING representation of array element
// in derived classes.
//
virtual void GetArrayElementString(HARRAY hArray,
                                   long lIndex,
                                   LPHSTRING lphString) = 0;

// sets the array element, converting the string to the correct data
// type
//
virtual void SetArrayElement(HARRAY hArray,
                             long lIndex,
                             Value* pValue) = 0;
```

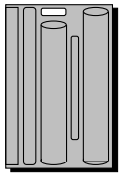
Diese Methoden werden in den konkreten Array-Attributklassen implementiert. GetArrayElementString() liefert die HSTRING-Repräsentation des Array-Elements an der Position lIndex. SetArrayElement() wandelt einen als Value-Objekt übergebenen Attributwert in den korrekten Datentyp um und speichert ihn als Array-Element an der Position lIndex. Mit diesen Methoden ist es nun möglich, die SQLWindowsAttribute-Schnittstelle bereits in der Klasse SQLWindowsArray zu implementieren. In SQLWindowsArray wird ein Array durch eine Liste von Value-Objekten

repräsentiert. Da Value-Objekte ihren Wert als Zeichenkette speichern, müssen die Array-Elemente mit Hilfe der oben genannten Methoden in Zeichenketten umgewandelt werden. Lediglich dieses Typkonvertierungs-Wissen ist es, was die konkreten Array-Klassen ausmacht – das Speichern der Array-Elemente als Strings kann von der gemeinsamen Oberklasse SQLWindowsArray erledigt werden, wie am Beispiel der Methode SetValue() der Klasse SQLWindowsArray gezeigt werden soll:

```
void SQLWindowsArray::SetValue(HARRAY hArray)
{
     HSTRING   hString;
     LPSTR     lpszString;
     long      lStringLength;
     Value*    poValue;

     hString = 0;
     m_Array.Flush();

     // associate the variable with an actual HSTRING value
     SWinInitLPHSTRINGParam(&hString, 128L);
     // get the current array bounds
     SalQueryArrayBounds(hArray, &m_lLowerLimit, &m_lUpperLimit);

     for (long i = m_lLowerLimit; i <= m_lUpperLimit; i++)
     {
          // retrieve string representation of array element
          GetArrayElementString(hArray, i, &hString);
          // lock string in memory
          lpszString = SWinHStringLock(hString, &lStringLength);
          // create new value object and add to array
          poValue = new Value(lpszString);
          m_Array.Push(poValue);
          // unlock
          SWinHStringUnlock(hString);
     }
}
```

Nach dem Ermitteln der Array-Grenzen wird die HSTRING-Repräsentation jedes Array-Elements von der konkreten Array-Klasse erfragt und in einem neu erzeugten Value-Objekt gespeichert, das dann in die Liste m_Array eingefügt wird.

6.6.10 Der Objekt-Manager



Der Objekt-Manager dient weniger der Verwaltung von Objekten, wie der Name vermuten läßt, sondern vielmehr der Verwaltung von Objekt-IDs. Der Name „Objekt-Manager“ wurde vom Atomizer-Pattern übernommen, um die begriffliche Nähe zum Entwurfsmuster zu wahren, auch wenn „ID-Manager“ ein passenderer Name wäre. Die Notwendigkeit eines allgemeinen Identitätskonzeptes für SQLWindows-Objekte wurde bereits in Abschnitt 6.3.1 diskutiert.

Die Schnittstelle des Objekt-Managers sieht wie folgt aus:

```
const int OBJMGR_MAXID  = 2000;
const int OBJMGR_OTHERBASE   = 1000;
```

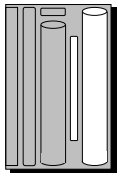
```
const int OBJMGR_NOTAVAIL    = 0;

class EXPORT ObjectManager
{
public:
// Singleton instance method
     static ObjectManager* Instance();

protected:
// Constructor
     ObjectManager();


public:
// Methods
     // returns a unique object id or OBJMGR_NOTAVAIL if there is no
     // such id.
     //
     unsigned long GetObjectId();

     // registers an object which already has an id (e.g. a material
     // which has a unique data base key as object id). returns TRUE if
     // successful, FALSE otherwise.
     //
     BOOL RegisterObject(unsigned long nObjectId);

     // unregisters an object; this object's id is now available again.
     // returns TRUE if successful, FALSE otherwise.
     //
     BOOL UnregisterObject(unsigned long nObjectId);
};
```

Der Objekt-Manager verwaltet Objekt-IDs mit einem Wertebereich von [1..OBJMGR_MAXID]. Dabei ist das Intervall [1..OBJMGR_OTHERBASE – 1] für Objekt-IDs vorgesehen, die extern vergeben wurden. So haben z.B. Material-Objekte[15], die in einer Datenbank gespeichert sind, u.U. eine Objekt-ID als Schlüsselattribut. Diese kann mittels RegisterObject() beim Objekt-Manager registriert werden. Neu vergebene Objekt-IDs, die über die Methode GetObjectId() angefordert werden können, entstammen demnach dem Intervall [OBJMGR_OTHERBASE..OBJMGR_MAXID]. Ein Objekt kann mittels UnregisterObject() die von ihm belegte Objekt-ID wieder freigeben.

Der Objekt-Manager ist ein Singleton, da gewährleistet sein muß, daß zur Laufzeit nur eine einzige Instanz der Klasse erzeugt werden kann (siehe [GHJV95] für eine Beschreibung des Singleton-Entwurfsmusters). Dies führte beim Einsatz innerhalb einer DLL mit C - und C++ - Code zu Problemen, die in Abschnitt 6.6.7 beschrieben sind.

[15] Der Materialbegriff versteht sich hier im Sinne des WAM-Sprachgebrauchs (vgl. Abschnitt 5.1.2).

6.6.11 Der Atomizer



Der E.I.S. Package-Mechanismus benutzt zum Speichern der Pakete eine Textdatei, die vom Aufbau her den Windows-INI-Dateien ähnelt. Im SOS sollte aus zwei Gründen kein festgeschriebenes Format verwendet werden: Erstens schränkt man dadurch die Erweiterbarkeit und Portabilität des Systems ein. Vielleicht möchte man später einen effektiveren Persistenzmechanismus benutzen oder – bei Plattformwechsel – einen Mechanismus, der vom darunterliegenden Betriebssystem besser unterstützt wird. Dann müßte man alle Zugriffe auf den Persistenzmechanismus den neuen Gegebenheiten anpassen. Außerdem erschwert eine zu spezielle Schnittstelle die Anbindung anderer Programme. So könnte man sich vorstellen, daß ein C++ - Programm auf die atomisierten Objekte zugreifen möchte, ohne daß es genaues Wissen darüber hat, wie diese konkret gespeichert sind. Diesen Anforderungen entspricht das Atomizer-Pattern von Dirk Riehle et. al. (vgl. [RS+96]), das einen standardisierten Mechanismus zum Schreiben beliebig komplexer Objektstrukturen auf verschiedene Datenstrukturen und zum Wiedereinlesen der Objekte von diesen Datenstrukturen beschreibt. Hier soll weniger das Entwurfsmuster selbst beschrieben werden, als vielmehr die Probleme, die eine Implementierung in C++ mit sich brachte.

Zum Verständnis des Atomizer-Patterns soll die folgende Strukturskizze mit einer kurzen Beschreibung genügen:

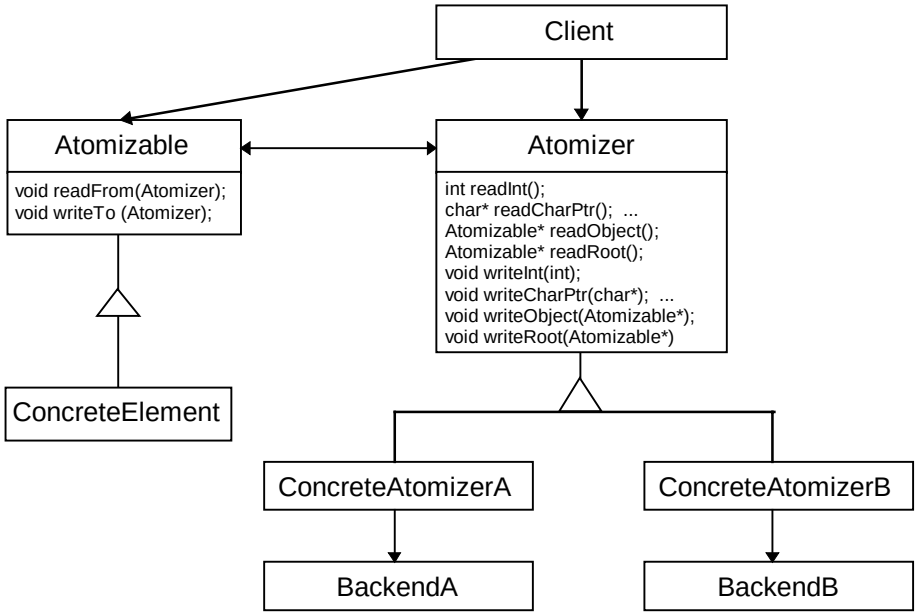


Abbildung 6.12: Struktur des Atomizer-Entwurfsmusters

Die zu atomisierenden Objekte müssen die Atomizable-Schnittstelle implementieren. Dazu erben sie von der Klasse Atomizable. Die Schnittstellenfunktionen readFrom() und writeTo() bekommen jeweils einen konkreten Atomizer als Parameter übergeben. Über die in der Klasse Atomizer definierten Funktionen read*() und write*() (der Asterisk '*' steht für einen Datentyp bzw. Object oder

Root) können die zu atomisierenden Objekte ihre Attribute in den Atomizer schreiben bzw. aus ihm auslesen.

Der Atomizer dient der Speicherung ganzer Objektbäume, die zur Laufzeit eines Programms aufgebaut werden. Gestartet wird der Atomisierungsvorgang für einen Objektbaum mit dem Wurzelobjekt MyObject durch Aufruf der Atomizer-Methode writeRoot(MyObject). Analog verhält es sich mit dem Zurücklesen über readRoot(MyObject).

Da der Atomizer allgemeine Informationen über das Objekt wie Objekt-ID und Objekttyp benötigt, müssen die zu atomisierenden Objekte zusätzlich von der Klasse Object erben – ein Umstand, der in [RS+96] nicht erwähnt wird, aber weitreichende Konsequenzen hat. Die einfachste Lösung ist, alle atomisierbaren Klassen ausschließlich von der Klasse Atomizable erben zu lassen, die ihrerseits von Object erbt:

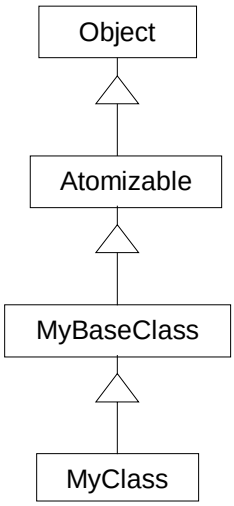


Abbildung 6.13: Atomizable, in die Object-Klassenhierarchie eingebaut

Diese Lösung ist zwar im allgemeinen unglücklich, da Atomizable als eine Aspektklasse im Sinne der WAM-Metapher zu verstehen ist (vgl. [KGZ94]), die eine zusätzliche Funktionalität zu einer Klasse hinzufügt und deshalb nicht von Object erben sollte. Im SOS ist dieser Entwurf aber unproblematisch, da die Implementierung aller SOS-Klassen zugänglich ist und somit leicht an die besonderen Gegebenheiten dieser Klassenhierarchie angepaßt werden kann. Anders verhielte es sich mit „zugekauften“ Klassen, z.B. aus kommerziellen Frameworks, die atomisierbar gemacht werden sollen. Diese wären wahrscheinlich nicht in obige Klassenhierarchie einzubinden.

Leider stellt obiger Entwurf die einzig technisch umsetzbare Lösung dar. Im folgenden werden die Ergebnisse eines Versuchs beschrieben, eine unabhängige Atomizable-Klasse zu entwerfen.

Wie bereits gesagt, sollte Atomizable selbst kein Erbe von Object, sondern eine Aspektklasse im Sinne der WAM-Metapher sein (vgl. [KGZ94]). Technisch wird dies über Mehrfachvererbung realisiert. Angenommen, Atomizable erbte von Object. Nun gebe es eine Klasse MyClass, die Erbe einer Klasse MyBaseClass ist, welche wiederum von Object erbt. MyClass soll nun atomisierbar gemacht werden. Dazu wird MyClass zusätzlich von Atomizable erben. Nun haben beide Elternklassen von MyClass die Klasse Object als Elternklasse, wie in folgender Abbildung dargestellt:

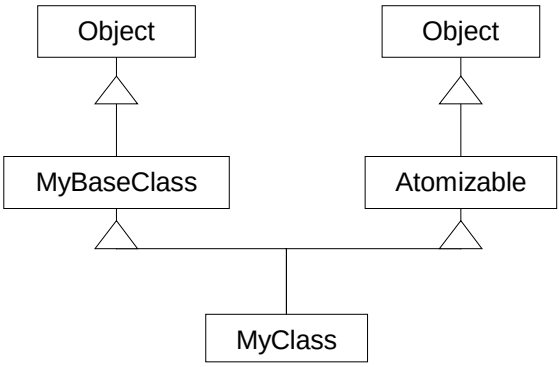


Abbildung 6.14: Atomizable als Erbe von Object

Wird nun bei einem Objekt MyObject vom Typ MyClass eine explizite Typumwandlung in den Typ Object durchgeführt ((Object*)MyObject), so weiß der Compiler nicht, welcher Object-Vorfahre zur Typumwandlung dienen soll. Was man also eigentlich haben möchte, ist eine Struktur der folgenden Form:

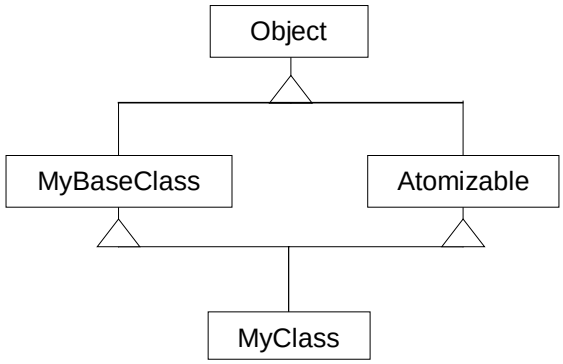


Abbildung 6.15: Virtuelles Erben von der Klasse Object

Dies läßt sich in C++ dadurch erreichen, daß alle Klassen virtuell von der Klasse Object erben:

```
class Atomizable : public virtual Object
{...};
class MyBaseClass : public virtual Object
{...};
class MyClass : public MyBaseClass, public Atomizable
{...};
```

Wenn die Klasse MyBaseClass jedoch nicht-virtuell von Object erbt, weil der Entwickler der Klasse solche Konstrukte wie oben aufgeführt nicht vorhersehen konnte, dann besteht keine Möglichkeit mehr, zu obiger Rautenstruktur zu gelangen. Hinzu kommt, daß die Klasse Atomizable lediglich ein Protokoll (oder einen „Aspekt“ im Sinne von [KGZ94]) darstellen soll, was ein Erben von Object kaum rechtfertigt. Also wurde die Klasse Atomizable zunächst ohne Vorfahren implementiert. Eine atomisierbare Klasse MyClass hätte dann folgenden Vererbungsbaum:

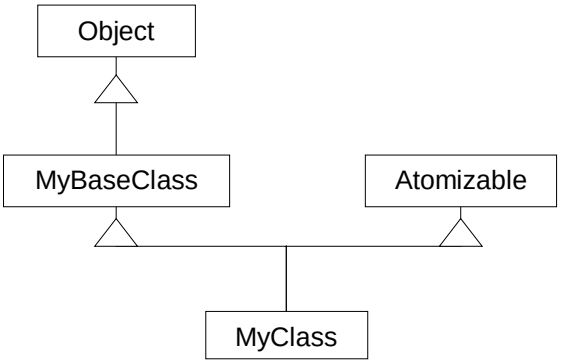


Abbildung 6.16: Atomizable als von Object unabhängige Aspektklasse

Nun muß aber ein Atomizer, der das zu atomisierende Objekt nur unter der Atomizable-Schnittstelle kennt, auf die in Object implementierten Objekteigenschaften zugreifen können. Eine Typumwandlung von Atomizable auf Object ist nicht möglich, da ja Atomizable nicht von Object erbt. Eine mögliche Lösung ist die, daß Atomizable eine Zeigervariable m_poThisAsObject vom Typ Object* auf das zu atomisierende Objekt hält, der im Konstruktor von Atomizable übergeben wird:

```
Atomizable::Atomizable(Object* poThisAsObject)
{
     m_poThisAsObject = poThisAsObject;
}
```

Nun kann ein Typkonvertierungs-Operator für einen Zeiger auf Object definiert werden:

```
inline Atomizable::operator Object*()
{
     return (Object*)m_poThisAsObject;
}
```

Man kann erwarten, daß dieser Operator in einer Situation wie der folgenden aufgerufen wird:

```
void ASCIIStreamAtomizer::WriteRoot(const Atomizable* poRoot)
{
     ...
     // push first object to be written
     m_pToHandle->Push((Object*)poRoot);
     ...
}
```

In diesem Ausschnitt aus der WriteRoot()-Methode eines konkreten Atomizers soll das zu atomisierende Objekt in die Liste der zu behandelnden Objekte eingetragen werden. Diese Liste arbeitet auf Object-Zeigern, so daß eine explizite Typumwandlung vonnöten ist. Eigenartigerweise wandelt der Visual C++ - Compiler den Typ eigenhändig um, ohne von dem in Atomizable implementierten Typumwandlungsoperator Gebrauch zu machen, so daß der Typ nicht korrekt konvertiert wird. Nun sollte die elegante Operator-Lösung durch eine Methode Object* ASCIIStreamAtomizer::AsObjectPtr() ersetzt werden, deren Aufruf anstatt der expliziten Typumwandlung stehen muß. Auch diese führte nicht zum gewünschten Ergebnis. Der Versuch, die oben erwähnte

Lösung mit virtuellen Basisklassen zu implementieren, scheiterte daran, daß der Microsoft-Compiler keine Typumwandlung von der (virtuellen Basis-)Klasse Object auf Atomizable zuließ. Was blieb, war die eingangs erwähnte, etwas unglückliche Lösung.

Im Unterschied zu dem in [RS+96] beschriebenen Atomizer, der ganze Objektbäume speichert, muß ein Atomizer für das SOS in der Lage sein, einzelne Objekte zu speichern und selektiv wieder auszulesen. Eine Methode WriteRoot() war also nicht vonnöten – alle Objekte sollten mittels WriteObject() geschrieben und durch ReadObject() wieder ausgelesen werden.

Der in [RS+96] beschriebene konkrete Atomizer arbeitet auf einem ASCII-stream-Objekt. Ein stream zeichnet sich durch die serielle Ein- und Ausgabe aus und ist daher für wahlfreien Zugriff auf die gespeicherten Objekte ungeeignet. Für das SOS mußte also ein anderes „Backend“ gefunden werden. Gewählt wurde das Windows-INI-Datei-Format – aus dem einfachen Grund, daß das Windows-API die Funktionen WritePrivateProfileString() und GetPrivateProfileString() für den Zugriff auf solche Dateien zur Verfügung stellt (vgl. [Mic93]). Um eine Unabhängigkeit von Windows zu erlangen, genügt es, die beiden Funktionen selbst zu implementieren.

Windows-INI-Dateien sind nach folgendem Schema aufgebaut:

```
[Rubrik]
Eintrag1=String1
Eintrag2=String2
...
[Rubrik2]
...
```

Dieses Schema legte nahe, für jedes atomisierte Objekt eine eigene Rubrik anzulegen, deren Einträge die Attribute des Objektes beschreiben. So entstand folgendes Format für die INI-Datei, welche die atomisierten Objekte aufnehmen sollte:

```
[Objekt-ID]
Type=Objekt-Typ
Attributname1=Datentyp, Attributwert1
Attributname2=Datentyp, Attributwert2
...
```

Objekt-Typ bezeichnet den Namen der SQLWindows-Klasse, von deren Typ das Objekt ist. *Datentyp* bezeichnet den im Atomizer verwendeten Datentyp. Diese Information ist notwendig, um den als Zeichenkette gespeicherten Attributwert beim Lesen des Objekts in den korrekten Datentyp umzuwandeln.

Diese spezielle INI-Datei wurde *Atomizer-Repository* (kurz: *Repository*) getauft, da sie für den Atomizer in der Tat eine Art Objekt-„Lager“ darstellt. Das Repository wird im aktuellen Verzeichnis unter dem Namen ATOMIZER.REP angelegt.

Für Array-Attribute war das Dateiformat noch zu erweitern. Da SQLWindows-Arrays sehr flexibel bezüglich der oberen und unteren Array-Grenzen sind, ist es notwendig, diese Grenzen zu speichern. Die Attributnamen wurden mit einem Index versehen, so daß ein Array folgendermaßen im Repository erscheint:

```
ArrayAttributnameLowerLimit=long, LowerLimit
ArrayAttributnameUpperLimit=long, UpperLimit
ArrayAttributname[LowerLimit]=Datentyp, Elementwert
ArrayAttributname[LowerLimit+1]=Datentyp, Elementwert
```

```
...
ArrayAttributname[UpperLimit]=Datentyp, Elementwert
```

Ein Auszug aus einem Beispiel-Repository soll das Format verdeutlichen:

```
[1000]
Type=fcAtomizableClass
m_bBooleanAttribute=ushort, 1
m_dtDateTimeAttribute=charptr, 1997-01-22-09.49.41.620000
m_nNumberAttribute=charptr, 4 193 3 14 10 0 0 0 0 0 0 0 0
m_sStringAttribute=charptr, SOS Test Drive
m_hFileHandleAttribute=ushort, 0
m_hSqlHandleAttribute=int, 0
m_hWndHandleAttribute=uint, 15456
m_sStringArrayAttributeLowerLimit=long, 0
m_sStringArrayAttributeUpperLimit=long, 3
m_sStringArrayAttribute[0]=charptr,
m_sStringArrayAttribute[1]=charptr, This is the
m_sStringArrayAttribute[2]=charptr, SQLWindows Object System (SOS)
m_sStringArrayAttribute[3]=charptr, in action
```

Das Attribut m_nNumberAttribute zeigt die Speicherung eines Number-Wertes als Bytefolge, wie in Abschnitt 6.6.4 beschrieben.

Wird ein Objekt, das bereits atomisiert wurde (und somit im Repository repräsentiert ist), erneut atomisiert, dann wird der bestehende Eintrag mit den neuen Werten überschrieben – das garantiert die Windows-Funktion WritePrivateProfileString().

Die Speicherung der Objekte im Repository erleichtert nicht zuletzt das Debugging von SQLWindows-Programmen, die mit dem SOS arbeiten. In der Debug-Version von SOS.DLL wird das Repository (d.h. die Datei ATOMIZER.REP) nach Beenden der SQLWindows-Applikation nicht gelöscht. Der SQLWindows-Anwendungsentwickler kann somit z.B. überprüfen, ob er alle Attribute seiner zu atomisierenden Objekte unter Angabe des korrekten Typs an den Atomizer übergeben hat.

6.6.12 Basisklassen

Als Basisklassen sollen all jene Klassen bezeichnet werden, die eine grundlegende Funktionalität zur Verfügung stellen (wie z.B. die Klasse Object) oder rein technisch motiviert sind (wie z.B. die Listenklassen). In den folgenden Klassendeklarationen sind nur die öffentlichen Elemente der Klassen wiedergegeben.

6.6.12.1 Object

```
class EXPORT Object
{
public:
// Constructor
     Object();

// Methods
     // retrieves the object type.
     //
     virtual char* GetObjectType() = 0;
```

```
     // retrieves the object id.
     //
     unsigned long GetObjectId();

     // sets the object id
     //
     void SetObjectId(unsigned long ulObjectId);
};
```

Die Klasse Object dient als Basisklasse aller Klassen des SOS. Auch wenn einige der SOS-Klassen die Object-Schnittstelle nicht benötigen, so hat man doch auf diese Weise einen erweiterbaren Klassenbaum mit einer gemeinsamen Wurzel geschaffen, dessen Elemente man immer unter einem gemeinsamen Typ ansprechen kann. Object ist eine abstrakte Klasse, deren Methode GetObjectType() in allen abgeleiteten Klassen überschrieben werden muß und den Objekttyp (d.h. den Klassennamen) als Zeichenkette zurückgeben soll.

Eine eindeutige Objekt-ID wird nur dann vergeben, wenn sie explizit verlangt wird. Dieses Prinzip der „object identity on demand“ verhindert die überflüssige Vergabe von Identitäten an Objekte, die während der gesamten Laufzeit des Programmes nie referenziert werden. Die Objekt-ID wird genau dann vergeben, wenn die Methode GetObjectId() zum ersten Mal aufgerufen wird. Dazu bedient sich die Klasse Object der (einzigen) Instanz der Klasse ObjectManager (siehe Abschnitt 6.6.10)

Ein manuelles Setzen der Objekt-ID mittels SetObjectId() kann dann vonnöten sein, wenn ein Objekt eine extern definierte Objekt-ID besitzt, beispielsweise den numerischen Schlüssel eines Datenbank-Eintrags, den das Objekt repräsentiert.

Die Klasse Object implementiert im wesentlichen ein Meta-Objekt-Protokoll mit Runtime-Type-Information, das im neuen C++ - Standard zum Sprachumfang gehören wird. Bis dieser in den Compilern implementiert ist, muß man sich mit Konstrukten wie dieser Klasse behelfen.

6.6.12.2 ObjectStack / ObjectList

Ein Nachteil des Visual C++ - Compilers der Versionen 1.x ist das Fehlen von Templates. So war es nicht möglich, generische Containerklassen zu benutzen, um Objekt-Container zu implementieren. Mit Hilfe der Basisklasse Object aber kann dieses Problem gelöst werden: Die Klassen ObjectStack und ObjectList implementieren nämlich einen Keller bzw. eine Liste von Object-Objekten, in die natürlich auch Objekte eingefügt werden können, deren Typ ein Erbe von Object ist. Die Schnittstellen der beiden Klassen sehen wie folgt aus:

```
class EXPORT ObjectStack : public Object
{
public:
// Constructor
     ObjectStack();

// Destructor
     ~ObjectStack();

// Methods

     // Object interface
     //
     char* GetObjectType();
```

```
     // Push a new element on the stack.
     //
     BOOL Push(const Object* poElement);

     // Pop the top element from the stack.
     //
     Object* Pop();

     // Test for the empty stack condition (no elements).
     //
     BOOL IsEmpty();

     // Empty the stack.
     //
     void Flush();
};


class EXPORT ObjectList : public ObjectStack
{
public:
// Methods

     // Object interface.
     //
     char* GetObjectType();

     // Test if the stack contains a given element.
     //
     BOOL Contains(const Object* poElement);

     // Test if the stack contains an element with a given Id.
     //
     BOOL ContainsKey(const unsigned long uiKey);

     // Returns the element that matches a given Id.
     //
     Object* At(const unsigned long uiKey);

     // Retrieve the first element in the list.
     //
     Object* GetFirst();

     // Retrieve the next element in the list.
     //
     Object* GetNext();

     // Retrieve the number of elements in the list.
     //
     int GetCount();
};
```

Die Methoden bedürfen wohl keiner weiteren Erklärung, abgesehen von den Methoden ContainsKey() und At() der Klasse ObjectList: Diese bekommen als Parameter einen Schlüsselwert (*key*) übergeben, welcher der Objekt-ID entspricht. Die Methoden werden mit diesen Namen im Atomizer-Entwurfsmuster verwendet. Auch wenn ein Name wie ContainsObjectId() vielleicht sinnvoller gewesen wäre, so wurden doch die Namen der Atomizer-Beschreibung übernommen, um die Identifikation des Entwurfsmusters zu erleichtern.

6.6.12.3 Node

Intern arbeiten die beiden Containerklassen mit Node-Objekten, die eine doppelt verkettete Liste implementieren:

```
class Node : public Object
{
public:
// Constructors
     Node();
     Node(Node* poPrev, Node* poNext, Object* poEntry);

// Methods

     // Object interface
     //
     char* GetObjectType();

     // Set previous node.
     //
     void SetPrevious(Node* poPrev);

     // Set next node.
     //
     void SetNext(Node* poNext);

     // Set the node's entry.
     //
     void SetEntry(Object* poEntry);

     // Set 'all in one'.
     //
     void SetNode(Node* poPrev, Node* poNext, Object* poEntry);

     // Retrieve the previous node.
     //
     Node* GetPrevious();

     // Retrieve the next node.
     //
     Node* GetNext();

     // Retrieve the node's entry (a pointer to an Object).
     //
     Object* GetEntry();
};
```

6.6.12.4 Tokenizer

Da das Atomizer-Repository auf formatierten Zeichenketten arbeitet, sollte eine elegante Lösung gefunden werden, um die einzelnen Elemente der Zeichenkette extrahieren zu können. Dies ist die Aufgabe der Klasse Tokenizer:

```
class Tokenizer : public Object
{
public:
// Constructor
     Tokenizer(const char* pcTokenString, const char* pcDelimiters);
```

```
// Methods

     // Object interface
     //
     char* GetObjectType();

     void SetDelimiters(const char* pcDelimiters);
     int GetToken(char* pcBuffer, int iBufferSize);
};
```

Beim Erzeugen eines Tokenizer-Objektes bekommt dieses die zu zerlegende Zeichenkette und eine Zeichenkette mit den Trennzeichen übergeben. Die Methode GetToken() liefert dann die jeweils nächste Teilzeichenkette, begrenzt durch eines der Trennzeichen bzw. das Zeilenende. Mit der Methode SetDelimiters() kann man eine neue Menge von Trennzeichen festlegen. Diese Menge kann auch leer sein, was z.B. in einer Situation wie der folgenden notwendig ist. Es handelt sich um einen Ausschnitt aus der Methode InitAttributes() der Klasse INIFileAtomizer:

```
Tokenizer AttributeTokenizer(pcAttributes, "=,");
AttributeTokenizer.GetToken(pcName, sizeof(pcName));
// don't read the 'Type' entry
if (strcmp(pcName, "Type") != 0)
{
     AttributeTokenizer.GetToken(pcType, sizeof(pcType));
     // delete delimiters to read the rest of the line as one string
     AttributeTokenizer.SetDelimiters("");
     AttributeTokenizer.GetToken(pcValue, sizeof(pcValue));
     // put type/value pair into attributes dictionary
     m_pAttributes->Append(pcName, pcType, pcValue);
}
```

Der Tokenizer soll eine aus dem Repository stammende Attribut-Zeichenkette in ihre Elemente (Attributname, Datentyp, Attributwert) zerlegen. Wir erinnern uns an den Aufbau einer Attributbeschreibung im Repository:

Attributname=Datentyp, Attributwert

Als Trennzeichen werden die Zeichen '=' und ',' benutzt. Da der Attributwert – z.B. im Falle eines Zeichenkettenwertes – diese Trennzeichen enthalten kann, muß nach dem Lesen des Datentyps die Menge der Trennzeichen zur leeren Menge reduziert werden – mit dem Ergebnis, daß nun der Rest der Attribut-Zeichenkette als ein Token betrachtet wird.

6.7 Zusammenfassung

Hauptaufgabe des SQLWindows Object System (SOS) ist das Speichern von SQLWindows-Objekten zum Zwecke der Übergabe dieser Objekte (genauer: von Referenzen auf diese Objekte) an andere Objekte oder Funktionen. Um eine Objektreferenz übergeben zu können, muß diese eindeutig sein. Deshalb enthält das SOS einen Objekt-Manager, der eindeutige Objekt-Identifikatoren generiert und verwaltet. Da dem SOS der Typ der gespeicherten Objekte bekannt ist, kann eine (eingeschränkte) Typprüfung beim Lesen eines Objektes aus dem SOS stattfinden.

Die Möglichkeit, Referenzen auf Objekte definieren zu können, ist unter anderem für die Implementierung von Benutztbeziehungen zwischen Objekten notwendig.

Wenngleich das SOS sicherlich ein Schritt in Richtung Referenzmechanismus mit (eingeschränkter) Typprüfung bei SQLWindows-Programmen ist, so muß sich der Benutzer des SOS doch der Zustandsbehaftetheit dieses Systems bewußt sein, die den Unterschied zu einem echten Referenzmechanismus ausmacht (vgl. Abschnitt 6.3.3).

Mit seiner Implementierung als DLL ist das SOS sehr stark an das Betriebssystem Microsoft Windows gebunden. Das ließ sich jedoch nicht vermeiden, da SQLWindows – selbst stark mit diesem Betriebssystem verknüpft – die Implementierung externer Erweiterungen nur über solche DLLs gestattet. Die innere Struktur des SOS ist jedoch plattformunabhängig: Die Windows-Datentypen reichen nur bis zur Klasse SOS in das System hinein und werden dort in Standard-Datentypen von C++ konvertiert, so daß eine Portierung des SOS höchstens eine Änderung der C-Schnittstellenfunktionen und der Klasse SOS zur Folge hätte. Es existiert eine SQLWindows-Portierung für das Betriebssystem Solaris, die aber auf einer Windows-Emulation aufsetzt, welche dann auch Grundlage für das SOS sein könnte.

Das SOS kann nicht alle SQLWindows-Datentypen speichern. So gibt es für den Typ Long String und für Arrays mit Elementen vom Typ Boolean und File Handle keine C-Definitionen oder Strukturen, die eine Übergabe von Werten diesen Typs an eine DLL erlauben.

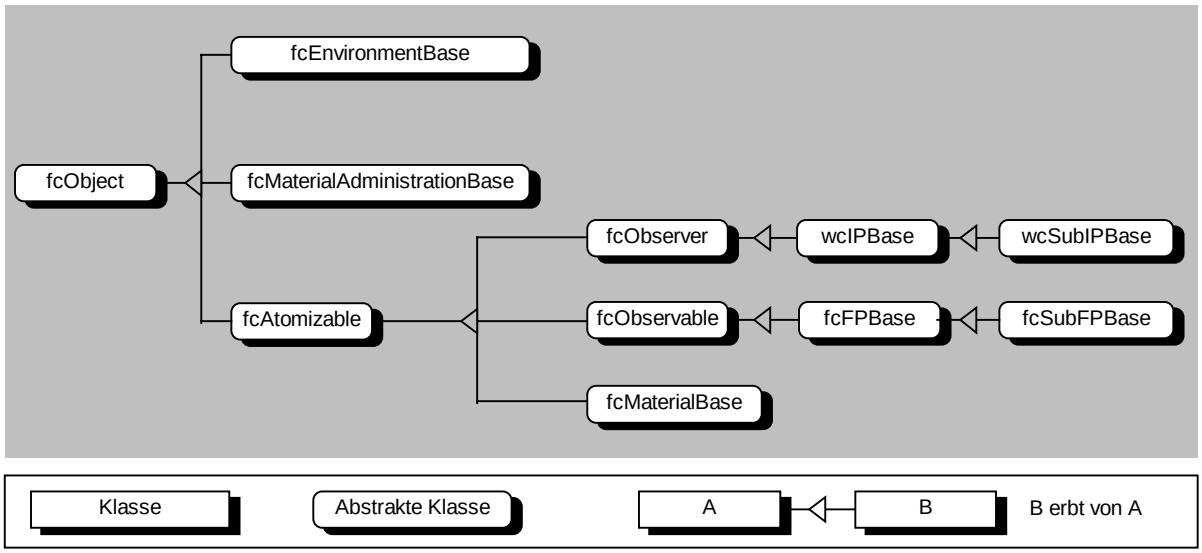
A Die Klassen des TANDEM-Frameworks

Die SQLWindows-Bibliothek `TANDEM.APL` enthält das WAM-Framework TANDEM. Der Name steht für das englische „T and M“ (für: Tools and Materials).

Ein Klassendiagramm zeigt zunächst alle Klassen des Frameworks und ihre Vererbungsbeziehungen im Überblick, gefolgt von der Auflistung der Klassen, alphabetisch sortiert nach dem Klassennamen ohne Typsuffix (`fc` bzw. `wc`). Die Klassenfunktionen werden ebenfalls in alphabetischer Reihenfolge aufgeführt.

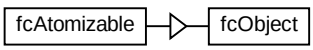
A.1 Klassendiagramm

Alle Klassen des folgenden Klassendiagramms sind in der SQLWindows-Bibliothek `TANDEM.APL` zu finden, mit Ausnahme der Klasse fcAtomizable, die zum SQLWindows Object System (SOS) gehört und in der Bibliothek `SOS.APL` deklariert ist.



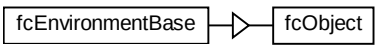
A.2 Klassenbeschreibung

A.2.1 fcAtomizable



Die Klasse fcAtomizable wird in Anhang B beschrieben.

A.2.2 fcEnvironmentBase



Die Klasse fcEnvironmentBase realisiert eine Systemumgebung, die Informationen über das Programm und dessen Kommandozeilenparameter hält und für die Verbindung zu den angeschlossenen Datenbanken verantwortlich ist.

ConnectToDB() — To be overridden [Internal]

Parameter *keine*

Return-Wert *keiner*

Stellt die Verbindung zu allen vom Programm benötigten Datenbanken her. Insbesondere sollten hier alle benötigten Sql Handles initialisiert werden, die auf Anfrage an die Materialverwalter übergeben werden können.

DisconnectFromDB() — To be overridden [Internal]

Parameter *keine*

Return-Wert *keiner*

Löst die Verbindung zu allen vom Programm benötigten Datenbanken.

Done() — Public

Parameter *keine*

Return-Wert *keiner*

Beschließt den Lebenszyklus des Objektes und führt „Aufräumarbeiten“ durch. Nach dem Aufräumen muß immer die Done()-Funktion der Elternklasse aufgerufen werden.

GetArgument(strParamName, rstrArgString) : Boolean — Public

Parameter	String	strParamName	Name des Kommandozeilenparameters
	Receive String	rstrArgString	Argument des Kommandozeilenparameters
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE.

Liefert das Argument eines Kommandozeilenparameters.

GetModuleName() : String — Public

Parameter	*keine*	
Return-Wert	String	Programmname

Liefert den Programmnamen ohne Suffix. GetProgramName() liefert den Namen inklusive Suffix.

GetPath() : String — Public

Parameter	*keine*	
Return-Wert	String	Programmpfad

Liefert den Programmpfad.

GetProgramName() : String	Public

Parameter *keine*

Return-Wert String Programmname

Liefert den Programmnamen inklusive Suffix. GetModuleName() liefert den Namen ohne Suffix.

Init()	Public

Parameter *keine*

Return-Wert *keiner*

Initialisiert das Objekt. Vor den eigentlichen Initialisierungstätigkeiten muß immer die Init()-Funktion der Elternklasse aufgerufen werden.

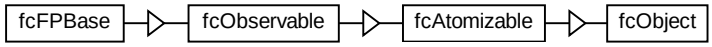
Reset()	Public

Parameter *keine*

Return-Wert *keiner*

Setzt das Objekt in einen Initialzustand zurück, d.h. alle Attribute erhalten ihre Initialwerte. Vor dem eigentlichen Zurücksetzen muß immer die Reset()-Funktion der Elternklasse aufgerufen werden.

A.2.3 fcFPBase

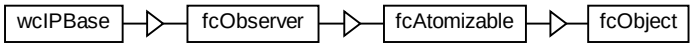


fcFPBase ist die Basisklasse für Funktionskomponenten. Sie besitzt gegenüber ihrer Basisklasse fcObservable keine zusätzlichen Funktionen.

Das Konzept der Trennung von Funktions- und Interaktionskomponente wird in Abschnitt 5.2.6 ausführlich beschrieben.

Für Sub-Funktionskomponenten steht die Klasse fcSubFPBase als Basisklasse zur Verfügung.

A.2.4 wcIPBase



wcIPBase ist die Basisklasse für Interaktionskomponenten. wcIPBase ist eine General Window Class (abstrakte Klasse), die nicht direkt instantiiert werden kann.

Das Konzept der Trennung von Funktions- und Interaktionskomponente wird in Abschnitt 5.2.6 ausführlich beschrieben.

Für Sub-Interaktionskomponenten steht die Klasse wcSubIPBase als Basisklasse zur Verfügung.

Exit() — Internal

Parameter *keine*

Return-Wert *keiner*

Beendet das Programm.

OnCreate() — To be overridden [Internal]

Parameter *keine*

Return-Wert *keiner*

Diese Funktion wird nach dem Eintreffen der Nachricht SAM_Create (bei der Erzeugung des Fensterobjektes) polymorph aufgerufen. Vor der Ausführung eigener Aktionen muß die Funktion OnCreate() der Basisklasse aufgerufen werden.

OnDestroy() — To be overridden [Internal]

Parameter *keine*

Return-Wert *keiner*

Diese Funktion wird nach dem Eintreffen der Nachricht SAM_Destroy (beim Zerstören des Fensterobjektes) polymorph aufgerufen. Nach der Ausführung eigener Aktionen muß die Funktion OnDestroy() der Basisklasse aufgerufen werden.

RegisterToFP(udvFP) : Boolean — Internal

Parameter	fcFPBase	udvFP	Funktionskomponente
Return-Wert	Boolean		TRUE, wenn die Interaktionskomponente registriert wurde, sonst FALSE.

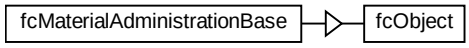
Registriert die Interaktionskomponente bei einer Funktionskomponente, um von dieser benachrichtigt werden zu können.

UnregisterFromFP(udvFP) : Boolean — Internal

Parameter	fcFPBase	udvFP	Funktionskomponente
Return-Wert	Boolean		TRUE, wenn die Interaktionskomponente deregistriert wurde, sonst FALSE.

Deregistriert die Interaktionskomponente bei einer Funktionskomponente.

A.2.5 fcMaterialAdministrationBase



Die Materialverwaltung kontrolliert und koordiniert die Zugriffe auf Materialien. Sie abstrahiert vom konkreten Speicherkonzept, indem sie die Datenbankzugriffe kapselt und vor dem Benutzer der Materialien verbirgt.

Das Konzept der Materialverwaltung wird in Abschnitt 5.2.11 beschrieben.

GetFirst() : Number — To be overridden [Public]

Parameter	*keine*		
Return-Wert	Number		Objekt-ID des ersten Material-Objektes, oder TANDEM_ERROR, falls kein solches Material existiert.

Liefert das erste Material-Objekt.

GetLast() : Number — To be overridden [Public]

Parameter	*keine*		
Return-Wert	Number		Objekt-ID des letzten Material-Objektes, oder TANDEM_ERROR, falls kein solches Material existiert.

Liefert das letzte Material-Objekt.

GetNext() : Number — To be overridden [Public]

Parameter	*keine*		
Return-Wert	Number		Objekt-ID des nächsten Material-Objektes, oder TANDEM_ERROR, falls kein solches Material existiert.

Liefert das nächste Material-Objekt.

GetPrevious() : Number — To be overridden [Public]

Parameter	*keine*		
Return-Wert	Number		Objekt-ID des vorhergehenden Material-Objektes, oder TANDEM_ERROR, falls kein solches Material existiert.

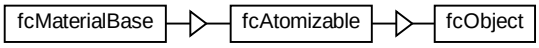
Liefert das vorhergehende Material-Objekt.

MakePersistent(nObjectId) : Boolean — To be overridden [Public]

Parameter	Number	nObjectId	Objekt-ID des zu speichernden Materials
Return-Wert	Boolean		TRUE, falls das Objekt gespeichert werden konnte, sonst FALSE.

Speichert ein Material-Objekt.

A.2.6 fcMaterialBase



fcMaterialBase bildet die Basisklasse für Materialien. Die Klassenfunktionen dienen der Überwachung des Materialstatus (unverändert oder geändert), sind also technisch und nicht so sehr fachlich motiviert.

Weitere Informationen zu Materialien findet man in den Abschnitten 5.1.2 und 5.2.2.

BooleanSafeSet(rbDestination, bSource)			Internal
Parameter	Receive Boolean	rbDestination	Zielvariable
	Boolean	bSource	Quellvariable
Return-Wert	*keiner*		

Weist der Zielvariablen den Wert der Quellvariable zu, falls sich die Werte der beiden Variablen voneinander unterscheiden. In diesem Fall wird das Modifikationsflag auf TRUE gesetzt.

DateTimeSafeSet(rdtDestination, dtSource)			Internal
Parameter	Receive Date / Time	rdtDestination	Zielvariable
	Date / Time	dtSource	Quellvariable
Return-Wert	*keiner*		

Weist der Zielvariablen den Wert der Quellvariable zu, falls sich die Werte der beiden Variablen voneinander unterscheiden. In diesem Fall wird das Modifikationsflag auf TRUE gesetzt.

GetModify() : Boolean			Public
Parameter	*keine*		
Return-Wert	Boolean	TRUE, wenn das Material verändert wurde, sonst FALSE.	

Liefert den Wert des Material-Modifikationsflags.

LongStringSafeSet(rlsDestination, lsSource)			Internal
Parameter	Receive Long String	rlsDestination	Zielvariable
	Long String	lsSource	Quellvariable
Return-Wert	*keiner*		

Weist der Zielvariablen den Wert der Quellvariable zu, falls sich die Werte der beiden Variablen voneinander unterscheiden. In diesem Fall wird das Modifikationsflag auf TRUE gesetzt.

NumberSafeSet(rnDestination, nSource)			Internal
Parameter	Number	rnDestination	Zielvariable
	Number	nSource	Quellvariable
Return-Wert	*keiner*		

Weist der Zielvariablen den Wert der Quellvariable zu, falls sich die Werte der beiden Variablen voneinander unterscheiden. In diesem Fall wird das Modifikationsflag auf TRUE gesetzt.

SetModify(bModified)			Internal
Parameter	Boolean	bModified	TRUE, wenn das Material verändert wurde, sonst FALSE.
Return-Wert	*keiner*		

Setzt das Modifikationsflag des Materials.

StringSafeSet(rstrDestination, strSource)			Internal
Parameter	Receive String	rstrDestination	Zielvariable
	String	strSource	Quellvariable
Return-Wert	*keiner*		

Weist der Zielvariablen den Wert der Quellvariable zu, falls sich die Werte der beiden Variablen voneinander unterscheiden. In diesem Fall wird das Modifikationsflag auf TRUE gesetzt.

A.2.7 fcObject

fcObject

fcObject ist die Basisklasse aller Frameworkklassen. Sie verwaltet Informationen über den Objekttyp und die Objekt-ID. Da SQLWindows keine Klassenkonstruktoren und -destruktoren kennt, enthält diese Klasse außerdem Funktionen, die beim Erzeugen bzw. Zerstören eines Objektes aufgerufen werden sollen.

Abschnitt 5.2.1 enthält eine ausführliche Klassenbeschreibung.

Alive() : Boolean		Public
Parameter	*keine*	
Return-Wert	Boolean	TRUE, wenn das Objekt gültig ist, sonst FALSE

Prüft, ob das Objekt noch „lebendig“ ist (d.h. Init() wurde aufgerufen, aber noch nicht Done()).

Done() — To be overridden [Public]

Parameter *keine*

Return-Wert *keiner*

Beschließt den Lebenszyklus des Objektes und führt „Aufräumarbeiten“ durch. Nach dem Aufräumen muß immer die Done()-Funktion der Elternklasse aufgerufen werden.

GetObjectId() : Number — Public

Parameter *keine*

Return-Wert Number Objekt-ID

Diese Funktion liefert die Objekt-ID.

GetObjectState() : Number — Public

Parameter *keine*

Return-Wert Number Objektstatus

Liefert den Objektstatus in Form einer der folgenden vordefinierten Konstanten:

OBJECT_NOT_INITIALIZED	Objekt noch nicht initialisiert (Init() wurde noch nicht aufgerufen)
OBJECT_INITIALIZED	Objekt initialisiert (Init() wurde aufgerufen)
OBJECT_INVALID	Objekt nicht mehr gültig (Done() wurde aufgerufen)

GetObjectType() : String — Public

Parameter *keine*

Return-Wert String Objekttyp

Diese Funktion liefert den (mittels SetObjectType()) gesetzten Klassennamen (Objekttyp).

Init() — To be overridden [Public]

Parameter *keine*

Return-Wert *keiner*

Initialisiert das Objekt. Vor den eigentlichen Initialisierungstätigkeiten muß immer die Init()-Funktion der Elternklasse aufgerufen werden.

Reset()	To be overridden [Public]

Parameter *keine*

Return-Wert *keiner*

Setzt das Objekt in einen Initialzustand zurück, d.h. alle Attribute erhalten ihre Initialwerte. Vor dem eigentlichen Zurücksetzen muß immer die Reset()-Funktion der Elternklasse aufgerufen werden.

SetObjectId(nObjectId)	Public

Parameter Number nObjectId Objekt-ID

Return-Wert *keiner*

Diese Funktion setzt die Objekt-ID. Dies wird normalerweise von der Implementierung der Funktion Init() in fcObject übernommen. Klassen mit externen Objekt-IDs (z.B. auf relationalen Datenbanken basierende Materialklassen) können ihre Objekt-ID mit Hilfe dieser Funktion manuell setzen.

SetObjectType(strObjectType)	Internal

Parameter String strObjectType Objekttyp

Return-Wert *keiner*

Diese Funktion setzt den Klassennamen (Objekttyp). Dieser kann mit GetObjectType() abgefragt werden.

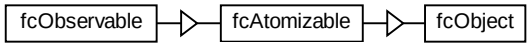
_Init()	Late-bound

_Reset()	Late-bound

_Done()	Late-bound

Diese Funktionen sind die spät gebundenen Äquivalente zu den oben beschriebenen Funktionen.

A.2.8 fcObservable



Die Klasse fcObservable implementiert die Schnittstelle des beobachteten Objektes im Rahmen des Beobachtermechanismus, der in Abschnitt 5.2.7 beschrieben ist.

Announce(nMessage)			Internal
Parameter	Number	nMessage	Nachricht
Return-Wert	*keiner*		

Benachrichtigt die registrierten Beobachter (d.h. fcObserver-Objekte), indem es deren Listen()-Funktion aufruft und die Nachricht nMessage übergibt.

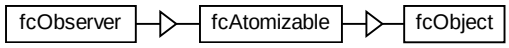
Register(strObjectType, nObjectId) : Boolean			Public
Parameter	String	strObjectType	Objekttyp (= Klassenname) des zu registrierenden Beobachters
	Number	nObjectId	Objekt-ID des zu registrierenden Beobachters
Return-Wert	Boolean		TRUE, wenn der Beobachter registriert werden konnte, sonst FALSE.

Registriert einen Beobachter. Ein Beobachter muß vom Typ fcObserver (oder einer von fcObserver abgeleiteten Klasse) sein.

Unregister(strObjectType, nObjectId) : Boolean			Public
Parameter	String	strObjectType	Objekttyp (= Klassenname) des zu deregistrieren - den Beobachters
	Number	nObjectId	Objekt-ID des zu deregistrierenden Beobachters
Return-Wert	Boolean		TRUE, wenn der Beobachter deregistriert werden konnte, sonst FALSE.

Deregistriert einen Beobachter.

A.2.9 fcObserver



Die Klasse fcObserver implementiert die Schnittstelle des beobachtenden Objektes im Rahmen des Beobachtermechanismus, der in Abschnitt 5.2.7 beschrieben ist.

GetWindowHandle() : Window Handle	Public

Parameter *keine*

Return-Wert Window Handle Fensterbezug des Objektes

Liefert den Fensterbezug des Objektes. Dieser wird nur bei Fensterklassen gesetzt, so z.B. von der Klasse wcIPBase, der Basisklasse für Interaktionskomponenten.

Listen(nMessage)	To be overridden [Public]

Parameter Number nMessage Nachricht

Return-Wert *keiner*

Dient dem Nachrichtenempfang von beobachteten Klassen (d.h. Erben der Klasse fcObservable).

ReadAttributes() : Boolean	Internal

Parameter *keine*

Return-Wert Boolean TRUE, wenn der Beobachter aus dem SOS ausgelesen werden konnte, sonst FALSE.

Liest die fcObserver-Attribute aus dem SOS aus. Dabei handelt es sich im wesentlichen um den Fensterbezug, der bei der beobachtenden Klasse zur Identifizierung des Beobachters verwendet wird.

Reset()	Public

Parameter *keine*

Return-Wert *keiner*

Setzt das Objekt in einen Initialzustand zurück, d.h. alle Attribute erhalten ihre Initialwerte. Vor dem eigentlichen Zurücksetzen muß immer die Reset()-Funktion der Elternklasse aufgerufen werden.

SetWindowHandle(hWndThis)	Public

Parameter Window Handle hWndThis Fensterbezug des Objektes

Return-Wert *keiner*

Setzt den Fensterbezug des Objektes. Dieses geschieht im Falle der Klasse wcIPBase, die von fcObserver erbt, automatisch.

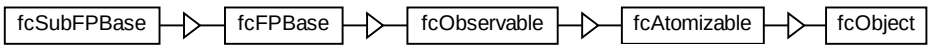
WriteAttributes() : Boolean	Internal

Parameter *keine*

Return-Wert Boolean TRUE, wenn der Beobachter in das SOS geschrieben werden konnte, sonst FALSE.

Schreibt die fcObserver-Attribute in das SOS. Dabei handelt es sich im wesentlichen um den Fensterbezug, der bei der beobachtenden Klasse zur Identifizierung des Beobachters verwendet wird.

A.2.10fcSubFPBase



fcSubFPBase ist die Basisklasse für Sub-Funktionskomponenten.

Das Konzept der Sub-Funktions- und -Interaktionskomponenten wird in Abschnitt 5.2.8 ausführlich beschrieben.

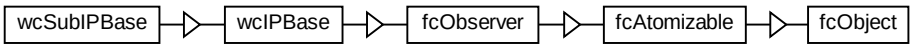
NotifyContextFP(nMessage)	To be overridden [Internal]

Parameter Number nMessage Nachricht

Return-Wert *keiner*

Benachrichtigt die Kontext-Funktionskomponente. Dazu sollte innerhalb von NotifyContextFP() die Listen()-Funktion der Kontext-Funktionskomponente aufgerufen werden.

A.2.11wcSubIPBase



wcSubIPBase ist die Basisklasse für Sub-Interaktionskomponenten.

Das Konzept der Sub-Funktions- und -Interaktionskomponenten wird in Abschnitt 5.2.8 ausführlich beschrieben.

NotifyContextIP(nMessage)	To be overridden [Internal]

Parameter Number nMessage Nachricht

Return-Wert *keiner*

Benachrichtigt die Kontext-Interaktionskomponente. Dazu sollte innerhalb von NotifyContextIP() die Listen()-Funktion der Kontext-Interaktionskomponente aufgerufen werden.

B Die Schnittstelle des SQLWindows Object System (SOS)

Das in Kapitel 6 beschriebene SQLWindows Object System (SOS) hat neben einer Klassenschnittstelle auch eine funktionale Schnittstelle.

Die Klassenschnittstelle besteht in der Klasse fcAtomizable, die in Abschnitt B.1 beschrieben ist.

Abschnitt B.2 beschreibt die funktionale Schnittstelle des SOS.

B.1 Die Klasse fcAtomizable

Objekte, die im SOS gespeichert werden, sollten von der abstrakten Klasse fcAtomizable erben, die in `SOS.APL` deklariert ist. Dabei sollte die Schnittstelle von fcAtomizable im Sinne einer Aspektklasse (vgl. [KGZ94]) über Mehrfachvererbung in die zu speichernde Klasse hineingeerbt werden. Da fcAtomizable von fcObject erben muß[16], SQLWindows aber – wie bereits in Abschnitt 4.3.8 ausgeführt – keine diamantenförmige Mehrfachvererbung unterstützt, können Klassen, die von fcAtomizable erben, nicht gleichzeitig von fcObject erben. fcAtomizable ist in dieser Implementierung demnach keine Aspektklasse, sondern fest in den fcObject-Klassenbaum eingebunden.

- ♦ Functional Class: fcAtomizable
  - ◊ Description: 'Aspect' class for atomizable classes.
    Note: This is not a real aspect class which is inherited by multiple inheritance, because SQLWindows doesn't support 'diamond' inheritance. fcAtomizable inherits from fcObject, so classes using the fcAtomizable interface cannot inherit directly from fcObject – they just inherit from fcAtomizable.
  - ♦ Derived From
    - ◊ Class: fcObject
  - ◊ Class Variables
  - ◊ Instance Variables
  - ♦ Functions
    - ◊ ! Public
    - ♦ Function: Init
    - ♦ Function: Reset
    - ♦ Function: Done
    - ♦ Function: WriteObject
    - ♦ Function: ReadObject
    - ◊ ! To be overridden [Internal]
    - ♦ Function: WriteAttributes
    - ♦ Function: ReadAttributes
    - ◊ ! Late-bound
    - ♦ Function: _WriteObject
    - ♦ Function: _ReadObject

Der Grund für das Erben von fcObject ist der Zugriff auf den Objekttyp innerhalb der Funktionen WriteObject() und ReadObject(). Diese Funktionen automatisieren – aus Sicht des Benutzers der Klasse fcAtomizable – den Aufruf der entsprechenden SOS-Funktionen (SOSBeginWrite() etc.) und stellen sicher, daß die SOS-Funktionen mit dem korrekten Objekttyp aufgerufen werden, den sie mit Hilfe der fcObject-Funktion GetObjectType() ermitteln. Auf diese Weise kann das SOS prüfen, ob ein entsprechendes Objekt (mit passendem Objekttyp und Objekt-Id) existiert. Kurz gesagt, ermöglichen diese Funktionen eine eingeschränkte Typprüfung – eingeschränkt vor allem

[16] Die Klasse fcObject gehört zum WAM-Framework und ist in `TANDEM.APL` deklariert.

deshalb, weil keine Super-/Subtypbeziehungen berücksichtigt werden: Nur „exakt passende Objekte“ werden akzeptiert.

Init() — Public

Reset() — Public

Done() — Public

Die Funktionen Init(), Reset() und Done() sind in der Klasse fcObject definiert.

WriteObject() : Boolean — Public

Parameter	*keine*	
Return-Wert	Boolean	TRUE, wenn das Objekt in das SOS geschrieben werden konnte, sonst FALSE.

Diese Funktion schreibt das Objekt in das SOS. Jedes Attribut des Objektes wird dabei einzeln an das SOS übergeben. Dazu ruft WriteObject() polymorph die Funktion WriteAttributes() auf.

ReadObject() : Boolean — Public

Parameter	*keine*	
Return-Wert	Boolean	TRUE, wenn das Objekt aus dem SOS ausgelesen werden konnte, sonst FALSE.

Diese Funktion liest das Objekt aus dem SOS aus. Jedes Attribut des Objektes wird dabei einzeln vom SOS angefordert. Dazu ruft ReadObject() polymorph die Funktion ReadAttributes() auf. Zuvor wird das Objekt durch einen polymorphen Aufruf der Funktion Reset() in seinen Initialzustand zurückgesetzt.

WriteAttributes() : Boolean — To be overridden [Internal]

Parameter	*keine*	
Return-Wert	Boolean	TRUE, wenn die Attribute in das SOS geschrieben werden konnten, sonst FALSE.

Diese Funktion schreibt die einzelnen Attribute des Objektes in das SOS. Dazu werden die SOSWrite*()-Funktionen des SOS verwendet. Die Reihenfolge, in der die Attribute übergeben werden, ist beliebig, da sie durch den Attributnamen eindeutig identifiziert werden können.

ReadAttributes() : Boolean	To be overridden [Internal]

Parameter *keine*

Return-Wert Boolean TRUE, wenn die Attribute aus dem SOS ausgelesen werden konnten, sonst FALSE.

Diese Funktion liest die einzelnen Attribute des Objektes aus dem SOS aus. Dazu werden die SOSRead*()-Funktionen des SOS verwendet. Die Reihenfolge, in der die Attribute gelesen werden, ist beliebig, da sie durch den Attributnamen eindeutig identifiziert werden können.

_WriteObject()	Late-bound

_ReadObject()	Late-bound

Diese Funktionen sind die spät gebundenen Äquivalente zu den oben beschriebenen Funktionen.

B.2 Verzeichnis der SOS-Funktionen

B.2.1 Initialisierung des SOS

SOSInit()

Parameter *keine*

Return-Wert *keiner*

Initialisiert das SQLWindows Object System (SOS). SOSInit() führt einen Versionsvergleich von `SOS.APL` und `SOS.DLL` durch. Schlägt dieser fehl, dann wird eine entsprechende Meldung ausgegeben, und die SQLWindows-Applikation wird beendet.

B.2.2 Versionskontrolle

SOSGetDLLVersion() : Number

Parameter *keine*

Return-Wert Number Versionsnummer

Liefert die Versionsnummer der Datei `SOS.DLL`. Die Versionsnummer von `SOS.APL` ist in der Konstanten SOS_APL_VERSION gespeichert. Beide Versionsnummern müssen übereinstimmen, um die Funktionsfähigkeit des SOS gewährleisten zu können. SOSInit() führt einen solchen Versionsvergleich durch.

B.2.3 Objektverwaltung

SOSGetObjectId() : Number

Parameter	*keine*		
Return-Wert	Number		eindeutige Objekt-ID

Liefert eine eindeutige Objekt-ID oder SOS_ID_NOTAVAIL, falls es keine solche ID mehr gibt.

SOSRegisterObject(nObjectId) : Boolean

Parameter	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Registriert ein Objekt, das bereits eine (extern vergebene) Objekt-ID hat, so z.B. ein Objekt, das seine Daten aus einer Datenbank mit einem numerischen Schlüsselattribut bezieht.

SOSUnregisterObject(nObjectId) : Boolean

Parameter	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Deregistriert ein Objekt und gibt dessen Objekt-ID wieder frei.

B.2.4 Schreiben und Lesen von SQLWindows-Objekten

SOSBeginWrite(strObjectType, nObjectId) : Boolean

Parameter	String	strObjectType	Objekttyp
	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Bereitet das Schreiben eines SQLWindows-Objektes vor. Diese Funktion muß vor dem Aufruf einer der Funktionen zum Schreiben von Attributen (SOSWrite*()) aufgerufen werden. Abgeschlossen wird der Schreibvorgang durch Aufruf der Funktion SOSEndWrite().

SOSEndWrite(nObjectId) : Boolean

Parameter	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schließt das Schreiben eines SQLWindows-Objektes ab. Die Objektdaten werden anschließend in das Atomizer-Repository eingetragen. Nach dem Aufruf von SOSEndWrite() darf keine der Funktionen zum Schreiben von Attributen (SOSWrite*()) mehr aufgerufen werden.

SOSBeginRead(strObjectType, nObjectId) : Boolean			
Parameter	String	strObjectType	Objekttyp
	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Bereitet das Lesen eines SQLWindows-Objektes vor. Dabei werden die Objektdaten aus dem Atomizer-Repository in das SOS eingelesen. Diese Funktion muß vor dem Aufruf einer der Funktionen zum Lesen von Attributen (SOSRead*()) aufgerufen werden. Abgeschlossen wird der Lesevorgang durch Aufruf der Funktion SOSEndRead().

SOSEndRead(nObjectId) : Boolean			
Parameter	Number	nObjectId	Objekt-ID
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schließt das Lesen eines SQLWindows-Objektes ab. Nach dem Aufruf von SOSEndRead() darf keine der Funktionen zum Lesen von Attributen (SOSRead*()) mehr aufgerufen werden.

B.2.5 Schreiben von Attributen

Achtung:
Das Schreiben der Attribute muß mit SOSBeginWrite() eingeleitet und mit SOSEndWrite() abgeschlossen werden!

SOSWriteBoolean(nObjectId, strName, bValue) : Boolean			
Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Boolean	bValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Boolean-Attribut in das SOS.

SOSWriteDateTime(nObjectId, strName, dtValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Date/Time	dtValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Date/Time-Attribut in das SOS.

SOSWriteNumber(nObjectId, strName, nValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Number	nValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Number-Attribut in das SOS.

SOSWriteString(nObjectId, strName, strValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	String	strValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein String-Attribut in das SOS.

SOSWriteFileHandle(nObjectId, strName, hFileValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	File Handle	hFileValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein File Handle-Attribut in das SOS.

SOSWriteSqlHandle(nObjectId, strName, hSqlValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Sql Handle	hSqlValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Sql Handle-Attribut in das SOS.

SOSWriteWindowHandle(nObjectId, strName, hWndValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Window Handle	hWndValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Window Handle-Attribut in das SOS.

SOSWriteDateTimeArray(nObjectId, strName, adtArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Date/Time	adtArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Date/Time-Array in das SOS.

SOSWriteNumberArray(nObjectId, strName, anArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Number	anArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Number-Array in das SOS.

SOSWriteStringArray(nObjectId, strName, astrArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	String	astrArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein String-Array in das SOS.

SOSWriteSqlHandleArray(nObjectId, strName, ahSqlArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Sql Handle	ahSqlArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Sql Handle-Array in das SOS.

SOSWriteWindowHandleArray(nObjectId, strName, ahWndArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Window Handle	ahWndArray	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Schreibt ein Window Handle-Array in das SOS.

B.2.6 Lesen von Attributen

Hinweis:
Das Lesen der Attribute muß mit SOSBeginRead() eingeleitet und mit SOSEndRead() abgeschlossen werden!

SOSReadBoolean(nObjectId, strName, rbValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive Boolean	rbValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Boolean-Attribut aus dem SOS.

SOSReadDateTime(nObjectId, strName, rdtValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive Date/Time	rdtValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Date/Time-Attribut aus dem SOS.

SOSReadNumber(nObjectId, strName, rnValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive Number	rnValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Number-Attribut aus dem SOS.

SOSReadString(nObjectId, strName, rstrValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive String	rstrValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein String-Attribut aus dem SOS.

SOSReadFileHandle(nObjectId, strName, rhFileValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive File Handle	rhFileValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein File Handle-Attribut aus dem SOS.

SOSReadSqlHandle(nObjectId, strName, rhSqlValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive Sql Handle	rhSqlValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Sql Handle-Attribut aus dem SOS.

SOSReadWindowHandle(nObjectId, strName, rhWndValue) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Receive Window Handle	rhWndValue	Attributwert
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Window Handle-Attribut aus dem SOS.

SOSReadDateTimeArray(nObjectId, strName, adtArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Date/Time	adtArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Date/Time-Array aus dem SOS.

SOSReadNumberArray(nObjectId, strName, anArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Number	anArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Number-Array aus dem SOS.

SOSReadStringArray(nObjectId, strName, astrArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	String	astrArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein String-Array aus dem SOS.

SOSReadSqlHandleArray(nObjectId, sName, hSqlArray)

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Sql Handle	hSqlArray[*]	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Sql Handle-Array aus dem SOS.

SOSReadWindowHandleArray(nObjectId, strName, ahWndArray) : Boolean

Parameter	Number	nObjectId	Objekt-ID
	String	strName	Attributname
	Window Handle	ahWndArray	Attributwert (Array)
Return-Wert	Boolean		TRUE, wenn die Funktion erfolgreich ausgeführt wurde, sonst FALSE

Liest ein Window Handle-Array aus dem SOS.

Literaturverzeichnis

[Att96] Karim H. Attia: Persistenz in objektorientierten Anwendungssystemen: Über die Abbildung von Objektstrukturen auf relationale Strukturen; Studienarbeit, Universität Hamburg (1996)

[Boo91] Grady Booch: Object Oriented Design with Applications; Benjamin/Cummings, Redwood City, CA (1991)

[Buc92] Marcellus Buchheit: Windows Programmierbuch; Sybex, Düsseldorf (1992)

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven: Prototyping – an approach to Evolutionary System Development; Springer, Berlin/Heidelberg etc. (1992)

[CP95] Sean Cotter with Mike Potel: Inside Taligent Technology; Addison-Wesley, Reading, MA (1995)

[Dud88] Duden »Informatik«; Bibliographisches Institut & F.A. Brockhaus AG, Mannheim (1988)

[EIS96] E.I.S. Konzept GmbH: Application Builder User Manual (1996)

[FKP96] Christian Fürll, Holger Koschek, Thomas Pfohe: Eine Fallstudie zur Objektorientierten Domänenanalyse; Studienarbeit, Universität Hamburg (1996)

[Fla96] David Flanagan: Java in a Nutshell; O'Reilly & Associates, Inc., Sebastopol, CA (1996)

[Flo93] Christiane Floyd: Arbeitsunterlagen zur Lehrveranstaltung „Einführung in die Softwaretechnik", Universität Hamburg (1993)

[Gaa93] Jostein Gaarder: Sofies Welt; Hanser, München/Wien (1993)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley, Reading, MA (1995)

[Gie93] William Gietz: SQLWindows Programming; Gupta Corp. (1993)

[Gup94a] Gupta Corporation: SQLWindows Developer's Reference (1994)

[Gup94b] Gupta Corporation: SQLWindows Help (1994)

[Gup94c] Gupta Corporation: SQLWindows Solo Quick Start (1994)

[Gup94d] Gupta Corporation: SQLBase C Application Programming Interface (1994)

[Hol95] David Holmes-Kinsella: Special Edition Using Gupta SQLWindows 5; Que, Indianapolis (1995)

[JV87] Eike Jessen, Rüdiger Valk: Modelle für Rechensysteme; Springer, Berlin/Heidelberg etc. (1987)

[KGZ94] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven: Objektorientierte Anwendungsentwicklung; Vieweg, Braunschweig/Wiesbaden (1994)

[KR90] Brian W. Kernighan, Dennis M. Ritchie: Programmieren in C, 2. Ausgabe, ANSI C; Hanser, München/Wien; Prentice-Hall Internat., London (1990)

[Lou94] Kenneth C. Louden: Programmiersprachen – Grundlagen, Konzepte, Entwurf; International Thomson Publishing, Bonn/Albany etc. (1994)

[LS87] Peter C. Lockemann, Joachim W. Schmidt (Hrsg.): Datenbank-Handbuch; Springer, Berlin/Heidelberg etc. (1987)

[Mey90] Bertrand Meyer: Objektorientierte Softwareentwicklung; Hanser, München/Wien (1990)

[Mic93] Microsoft Windows 3.1 Software Development Kit, Programmer's Reference, Volume 2: Functions; Redmond, WA (1993)

[MMP88] Ole Lehrmann Madsen, Birger Møller-Pedersen: What object-oriented programming may be – and what it does not have to be; in: Stein Gjessing, Kristen Nygaard: ECOOP '88 (1988), S. 1-20

[Nie] Charles Niemeier: A Discussion of Object-Oriented Programming Languages, Gupta Technical Presentation, reprinted with permission from: Leap Into Object-Oriented Productivity, Gupta Fifth Annual Developers Conference Proceedings

[Obe94] Horst Oberquelle: Arbeitsunterlagen zur Vorlesung „Programmiersprachen"; Universität Hamburg (1994)

[Ovum94] Ovum Ltd.: Ovum Evaluates: 4GLs, SQLWindows (1994)

[Pet90] Charles Petzold: Programming Windows, 2nd Edition; Microsoft Press, Redmond, WA (1990)

[RS95] Dirk Riehle, Martin Schnyder: Design and Implementation of a Smalltalk Framework for the Tools and Materials Framework; UBILAB Technical Report 95.7.1 (1995)

[RS+96] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, Heinz Züllighoven: The Atomizer – Efficiently Streaming Object Structures (1996)

[RZ95] Dirk Riehle, Heinz Züllighoven: A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor; in: J.O. Coplien, D.C. Schmidt: Pattern Languages of Program Design, Addison-Wesley (1995)

[Str87] Bjarne Stroustrup: What is 'Object-Oriented Programming' ?; in: Jean Bézivin et al.: ECOOP '87 (1987), S. 51-70

[Str92] Bjarne Stroustrup: Die C++ Programmiersprache; Addison-Wesley, Bonn/München etc. (1992)

[Syl96] Karl-Heinz Sylla: Der Pausenplaner – Szenario, CRC-Karten, Systemvision, zur Unterstützung der Komponenten und Framework Diskussion; http://set.gmd.de/~sylla/DACH/ (1996)

[Tal95] Taligent, Inc.: The Power of Frameworks, Addison-Wesley, Reading, MA (1995)

[Weg87] Peter Wegner: Dimensions of Object-Based Language Design; in: Norman Meyrowitz: OOPSLA-87, ACM SIGPLAN Notices Vol. 22 (1987), S. 168-182

[Zeh87] C.A. Zehnder: Informationssysteme und Datenbanken; Teubner, Stuttgart (1987)

[Zül94] Heinz Züllighoven: Arbeitsunterlagen zur Lehrveranstaltung „Objektorientierte Systementwicklung“, Teil A: Entwurf, WS 1994/95; Universität Hamburg (1994)

[Zül97] Heinz Züllighoven: Das WAM-Konstruktionshandbuch (Entwurf); Hamburg (1997)