Objektorientierte Zeitreihenverwaltung

Carsten Wilhelm

Universität Osnabrück

Juni 1998

Vorwort

Die vorliegende Diplomarbeit entstand im Rahmen einer Projektarbeit zum Thema ,,Zeitreihenprognose mit Neuronalen Netzen" im Fachbereich Mathematik/Informatik der Universität Osnabrück. Diese Arbeit ist in der Arbeitsgruppe Praktische Informatik von Prof. Dr. Oliver Vornberger angesiedelt, das Projekt zur Prognose von Zeitreihen steht unter der Leitung von Dipl.-Math. Frank M. Thiesing.

Mein Dank gilt an dieser Stelle allen, die zum Gelingen dieser Arbeit beigetragen haben:

- Herrn Prof. Dr. Oliver Vornberger für die Betreuung der Arbeit und für die Gespräche, durch die weitere Ideen und Arbeitsansätze erlangt wurden,
- Herrn Dipl.-Math. Frank M. Thiesing für die Betreuung der Arbeit und für die mehrjährige Zusammenarbeit auf dem Gebiet der Zeitreihenprognose. Ebenfalls bin ich für die häufigen Anregungen dankbar, die ich aus der Arbeit mit Herrn Thiesing bekommen habe,
- Herrn Dipl.-Math. Stephan Krone und Herrn Cand.-Math. Karsten Moranz, die im Rahmen der Projektarbeit viele Ideen beigetragen haben und durch die wichtige Teile des Projektes entstanden sind,
- meinem Vater, durch dessen Hilfe mir das Studium ermöglicht wurde
- und nicht zuletzt meiner Freundin Nicole, die mich während der gesamten Diplomarbeit unterstützt hat.

Erklärung

Ich versichere, die vorliegende Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Osnabrück, den 20. Juni 1998

Carsten Wilhelm

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Diese Diplomarbeit ist Teil eines mehrjährigen Projektes, das die Prognose von Zeitreihen untersucht. Dazu wurde ein Prototyp zur Verwaltung und zur Prognose von Zeitreihen erstellt. Dieser Prototyp soll im Rahmen dieser Arbeit neu überarbeitet werden.

1.1 Voraussetzungen

Das bestehende System zur Verwaltung basiert komplett auf der Programmiersprache Perl [WS91], da mit dieser Sprache schnell implementiert werden kann und Perl große Vorteile beim Erstellen eines Prototyps besitzt. Es können schnell Änderungen am System vorgenommen werden, Compilierzeiten entfallen, da der Code interpretiert wird. Ebenfalls ist der Code durch die Benutzung eines Interpreters extrem portabel, da die Quellen ohne Compilierung auf jedem System laufen, welches einen Perl-Interpreter zur Verfügung stellt.

Andererseits wurde das System im Laufe der Projektarbeit immer komplexer, Fehler in der Programmierung führten einerseits nicht sofort zu einem Programmabbruch und andererseits war ein einmal auftretender Fehler in der großen Menge von Skripten schwer aufzufinden. Der Wunsch nach einer neu strukturierten Verwaltung lag also nahe.

Das bestehende Datenformat durfte allerdings nicht geändert werden, da hierauf bereits eine graphische Oberfläche aufsetzt, die für die Visualisierung der Zeitreihen und für die interaktive Prognosesteuerung eingesetzt wird [Kro97].

1.2 Zielsetzung

Das vorrangige Ziel ist es, die Laufzeit des Systems zu reduzieren, indem man von einer interpretierten Sprache wie Perl zu einer compilierbaren wechselt.

Durch schnelleren Programmcode wäre man auch in der Lage neue Funktionen in das System aufzunehmen, als Hauptwunsch ist hier die Vorverarbeitung der Zeitreihen *on the fly* zu nennen.

Ein Vorteil der Vorverarbeitung zur Laufzeit ist die höhere Flexibilität, beispielsweise beim Testen verschiedener Parameter einer Vorverarbeitung.

Hinzu kommt, daß das System immer wieder erweitert wurde, so daß große Teile des Perl-Codes mehrmals umprogrammiert wurden. Bei einer solchen Menge von Skripten traten dann teilweise nicht sofort entdeckte Inkonsistenzen auf.

Ein weiterer wichtiger Aspekt ist die leichte Erweiterbarkeit des Systems. Es soll ohne Eingriff in bestehende Quellen um neue Projekte oder Vorverarbeitungen ergänzt werden können. Dieser Aspekt ist der Hauptgrund für einen objektorientierten Ansatz.

1.3 Überblick

Die Arbeit gliedert sich in vier Teile:

Im 2. Kapitel wird kurz auf das vorhandene System eingegangen. Dieses wurde im Rahmen einer mehrjährigen Projektarbeit erstellt und ist Ausgangsbasis für die vorliegende Arbeit, da die Datenstrukturen erhalten bleiben müssen.

Die folgenden zwei Kapitel beschreiben die Umsetzung in ein objektorientiertes Modell.

Im 3. Kapitel werden die Ideen für einen objektorientierten Ansatz beschrieben.

Im 4. Kapitel wird die Umsetzung nach C++ dargestellt.

Der nächste Teil der Arbeit befaßt sich mit den Möglichkeiten der Anwendung und Erweiterung des objektorientierten Systems.

Im 5. Kapitel werden implementierte Anwendungen der Klassenbibliothek aufgeführt.

Im 6. Kapitel wird dann erläutert, wie neue Verfahren oder Funktionen in das System integriert werden können.

Den letzten Teil der Arbeit bildet das 7. Kapitel: Es wird eine komplette Liste aller Klassen angegeben. Zu jeder Klasse werden die Methoden inklusive der Parameter aufgelistet und die Funktionsweise dargestellt. Dieses Kapitel ist als Kurzreferenz der Klassenbibliothek anzusehen.

Daran anschließend folgt eine kurze Zusammenfassung der Diplomarbeit.

Im Anhang werden weitere Informationen und Zusammenfassungen zur Arbeit aufgelistet. Dazu gehört insbesondere eine Methodik zum Auffinden von Fehlern in der Speicherverwaltung.

Die Arbeit schließt mit einer Literaturliste.

Kapitel 2

Das bisher eingesetzte System

In diesem Kapitel wird kurz das bisher eingesetzte System beschrieben, inklusive der benutzten Datenstrukturen. Auf dieser Grundlage wird im nächsten Kapitel ein objektorientierter Ansatz aufgebaut. Das Datenformat wird komplett dargestellt, da sich dieses Format auch in der Implementation dieser Diplomarbeit wiederfindet. Auf das bestehende Skript-System in Perl wird nur in Einzelfällen eingegangen, sofern die Skripte einen wichtigen Einfluß auf das zu entwickelnde System haben.

Eine detailiertere Darstellung des eingesetzten Systems ist in [Thi98] nachzulesen.

2.1 Zeitreihen

2.1.1 Was ist eine Zeitreihe ?

Eine Zeitreihe ist im wesentlichen eine Menge von Werten x, die über die Zeit t aufgetragen werden. Es entsteht eine Folge von Werten $x(t)$, $x(t+1)$, $x(t+2)$, ... Man kann sich eine Zeitreihe also ähnlich einer Wertetabelle vorstellen, bei der die Zeiteinheiten alle äquidistant aufgetragen sind (bezogen auf die zugrundeliegende Zeiteinheit). Tabelle 2.1 zeigt die Werte einer Beispielzeitreihe, die dazu passende graphische Repräsentation der Zeitreihe ist in Abbildung 2.1 zu sehen.

Tabelle 2.1: Zeitreihe in Form einer Wertetabelle

t	1	2	3	4	5	6	$\cdots$
$x(t)$	1	2	4	3	1	-1	$\cdots$

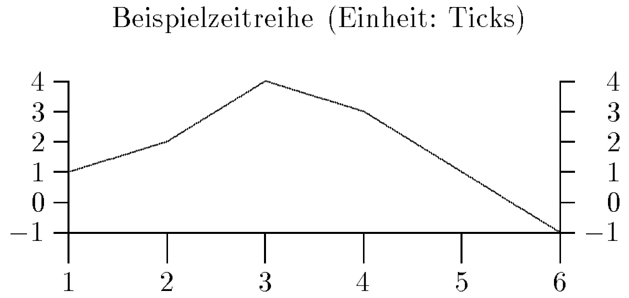

Abbildung 2.1: Graphische Darstellung einer Zeitreihe

In dem während der Projektarbeit entwickelten System besitzen Zeitreihen noch weitere Informationen, z.B. über den Start- und Endzeitpunkt, die Werte- und Zeiteinheit oder den Namen der Zeitreihe. Mit Hilfe all dieser Informationen kann man genau die Position der Zeitreihe im Dateibaum bestimmen, oder man kann die Entstehungsgeschichte der Zeitreihe ablesen (z.B. die Art der Prognose).

2.1.2 Minimale Informationen der Zeitreihe

Damit eine Zeitreihe im System verarbeitet werden kann, muß sie ein Minimum an Informationen enthalten:

- Start- und Endzeitpunkt (gemessen in Zeiteinheiten seit einem Startdatum, hier z.B. seit dem 1.1.1945),
- Zeiteinheit (mit Faktor) und Werteeinheit,
- zugehöriges Projekt und u.U. weitere Komponenten (zur feineren Unterteilung eines Projektes),
- Name der Zeitreihe (vorgeschrieben: <Zeiteinheit>.<Typ>[.Suffix]),
- Beschreibung der Zeitreihe (z.B. für die Anzeige in der Oberfläche),
- der Typ der Aggregation (Summe, Mittelwert, Minimum, Maximum),

2.1.3 Unterteilung in Kopf und Rumpf

Da unsere Zeitreihen eine klare Trennung von Werten und Informationen haben, wurde diese Trennung auch bei der Speicherung der Zeitreihe beibehalten, es existieren also auch zwei getrennte Dateien. Die eine enthält nur die Werte, die andere alle restlichen Informationen der Zeitreihe. Im folgenden nennen wir die Informations-Datei **Kopf** und die Werte-Datei **Rumpf**. Die Trennung hat Vorteile, da man einerseits leicht Zugriff auf die Kopf-Informationen hat, andererseits eine bestehende Zeitreihe verlängern kann, ohne die gesamte Datei neu zu schreiben. Es wird nur der Kopf neu geschrieben, neue Werte werden an den Rumpf angehängt.

Kopf der Zeitreihe

Im Kopf der Zeitreihe stehen pro Zeile immer ein Schlüssel, der zugehörige Wert und optional ein Kommentar in der folgenden Form:

```
# Schlüssel: Wert
```

oder

```
# Schlüssel: Wert (Kommentar)
```

Kommentarzeilen beginnen mit zwei Doppelkreuzen.

```
## Kommentar
```

Wird ein Wert abhängig von der Sprache benötigt (z.B. der Name der Werteeinheit), so tritt der Schlüssel mehrfach mit einem Ländercode zu Beginn auf, zur Zeit sind deutsch (`D-`) und englisch (`E-`) vorgesehen. Paßt keiner der zur Verfügung stehenden Ländercodes, ist der Schlüssel sprachunabhängig (z.B. numerische Werte aller Art).

```
## Wochenzeitreihe, Start KW 4/1998
# Start:         2769 (4/1998)
# D-Zeiteinheit: Wochen
# E-Zeiteinheit: weeks
```

Rumpf der Zeitreihe

Im Rumpf stehen nur noch die numerischen Werte der Zeitreihe. Eine Zeile entspricht dabei einem Tick der verwendeten Zeiteinheit. Fehlende Werte werden durch eine Leerzeile gekennzeichnet und als sogenannter *Missing-Wert* interpretiert. Missing-Werte gehen nicht in die Prognose ein, ebenfalls werden diese Werte aus der Berechnung z.B. des Mittelwertes gestrichen.

Wird eine Zeitreihe mit mehreren Werten pro Tick benötigt (z.B. zur Angabe eines Signifikanz-Intervalls), so werden diese durch ein Leerzeichen getrennt in einer Zeile angegeben.

2.1.4 Verwaltung der Zeitreihen

Die Zeitreihen werden direkt im Dateisystem gespeichert, dabei sind Projekt, Komponenten und der Name der Zeitreihe wichtig. Die Datei

```
$PROJEKT_HOME/<Komponente1>/.../<KomponenteN>/ZR/Head/<Name>
```

beinhaltet den Kopf der Zeitreihe, die Datei

```
$PROJEKT_HOME/<Komponente1>/.../<KomponenteN>/ZR/Body/<Name>
```

den Rumpf. Dabei entspricht die erste Komponente dem Projekt.

Ein Beispiel für den Kopf ist in Anhang A.1 zu sehen, der zugehörige Rumpf der Zeitreihe steht in Anhang A.2.

2.1.5 Indizierung der Zeitreihen

Nicht alle Projekte oder Zeitreihen, die verfügbar sind, sollen oder können in der Oberfläche angezeigt werden. Zu diesem Zweck existiert in jedem Unterverzeichnis

```
$PROJEKT_HOME/<Komponente1>/../<KomponenteM>/INDEX
```

eine Datei des Namens `INDEX`, die alle Zeitreihen oder Projekte enthält, die von der Oberfläche angezeigt werden sollen. Diese Indizierung macht ein Navigieren innerhalb aller Projekte und Zeitreihen möglich.

Beispiele für `INDEX`-Dateien sind in Anhang A.3 zu sehen.

2.1.6 Verwaltung von Prognosedaten

Ein ähnliches Verfahren, wie bei der Indizierung, kommt zum Einsatz, um Informationen über eine prognostizierte Zeitreihe zu sichern. Jede prognostizierte Zeitreihe bekommt eine eindeutige Nummer, anhand der man weitere Informationen zum Prognoseablauf abfragen kann (wie z.B. eingehende Zeitreihen, Prognoseverfahren und Parameter). Die Informationen werden zeilenweise in einer Datei `PROGNOSE.LST` gesichert, dabei steht die Datei im Unterverzeichnis der Zeitreihe (d.h., im selben Verzeichnis, in dem die `INDEX`-Datei mit der Zeitreihe liegt). Am Anfang der Datei stehen Zeilen der Form

```
#FORMAT VERFAHREN:NameParam1:NameParam2:NameParam3
```

Diese geben für jedes Verfahren `VERFAHREN` die Namen der verwendeten Parameter an, beispielsweise könnte eine naive Prognose wie folgt eingetragen werden:

```
#FORMAT NaivePrognose:Anzahl vorausgesagte Werte
```

Später in der Datei steht nun zu jeder prognostizierten Zeitreihe eine weitere Zeile, in der die Parameter eingetragen sind, die bei dieser speziellen Prognosezeitreihe benutzt wurden. Diese Zeile hat die Form

```
NUMMER:VERFAHREN;Param1;Param2;Param3;OUT-ZR;IN-ZR1;IN-ZR2;...
```

Der erste Parameter bezeichnet dabei die Nummer der Prognose, die nächsten Parameter sind das benutzte Verfahren und die zugehörigen Parameter. Nach den Parametern wird die Output-Zeitreihe angegeben, d.h., die Zeitreihe, die prognostiziert wurde. Dabei müssen für `OUT-ZR` zwei Parameter durch `;` getrennt angegeben werden: Zuerst kommt der Name der Zeitreihe, danach folgt die auf diese Zeitreihe angewandte Vorverarbeitungspipeline. Daran anschließend folgen alle eingehenden Zeitreihen `IN-ZR1`, `IN-ZR2`, etc. Dabei müssen jetzt drei verschiedene Parameter pro Zeitreihe angegeben werden. Der erste entspricht zusätzlichen Parametern der Zeitreihe, z.B., ob die Zeitreihe zum Prognosezeitpunkt verfügbar ist. Der zweite Parameter besteht aus dem Namen der Zeitreihe, der dritte gibt analog zur Output-Zeitreihe die benutzte Vorverarbeitungspipeline für diese spezielle Zeitreihe an.

Wurde beispielsweise die Zeitreihe `CO2|Tage.CO2` mit der Vorverarbeitung `LINEAR 0.1 0.9` naiv 4 Tage weit prognostiziert und wäre dabei die Prognosezeitreihe `CO2|Tage.NAIV_002` entstanden, so würde etwa folgender Eintrag in der Datei auftauchen:

```
2:NaivePrognose:4:CO2|Tage.CO2;LINEAR 0.1 0.9 MIN MAX
```

Wobei `MIN` und `MAX` das Minimum bzw. Maximum der Zeitreihe `Tage.CO2` entspricht. Die eingetragenen Vorverarbeitungen speichern mehr Information, da mehr Parameter zur Rückskalierung benötigt werden.

Ein weiteres Beispiel für eine `PROGNOSE.LST`-Datei ist in Anhang A.4 zu sehen.

2.2 Vorverarbeitungen (oder Skalierungen)

Es existieren einige Vorverarbeitungen, die allerdings nicht zur Laufzeit aktiviert werden. Die Vorverarbeitungen werden direkt nach dem Erzeugen der Rohzeitreihen erstellt und ebenfalls im Dateisystem gesichert. Vorverarbeitete Zeitreihen erkennt man an einem angehängten Kürzel, z.B. wird aus `Wochen.ABV` nach einer linearen Skalierung die Zeitreihe `Wochen.ABV.LIN`.

2.2.1 Vorverarbeitungspipeline

Auf eine Zeitreihe im Rohzustand kann man hintereinander mehrere Vorverarbeitungen anwenden. Dabei ist das Ergebnis einer Vorverarbeitung die Eingabe der direkt anschließenden. Dadurch kann man komplexe Modifikationen mit Hilfe vieler einfacher Vorverarbeitungen erreichen. Das Verfahren ähnelt einem Baukasten. Die Schreibweise einer Pipeline ist angelehnt an die Schreibweise von Pipelines in Unix-Systemen. Die verschiedenen Vorverarbeitungen werden durch das Pipe-Symbol (`|`) getrennt. Ein einfaches Beispiel einer Pipeline ist in Abbildung 2.2 zu sehen. Die Zeitreihe wird durch die Pipeline `DIFF 1 1 | LINEAR 0.1 0.9` vorverarbeitet, d.h., es wird zuerst die erste Differenz der Zeitreihe berechnet (wobei die Zeitreihe einen Wert kürzer wird) und danach werden die Werte linear auf das Intervall $[0.1, 0.9]$ skaliert.

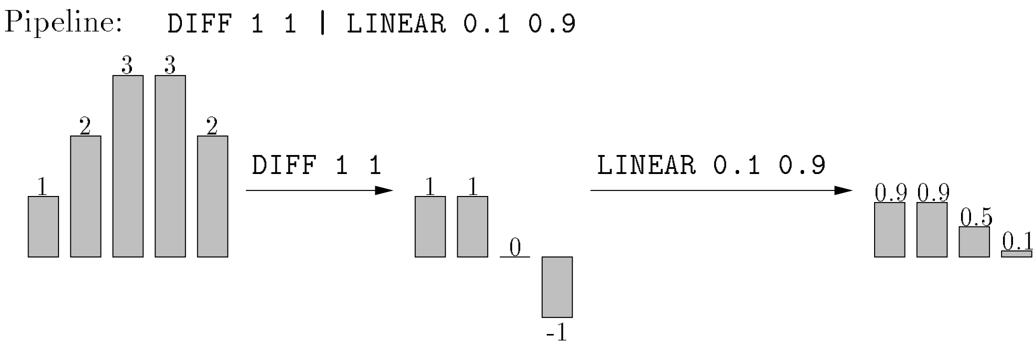

Abbildung 2.2: Beispiel einer Vorverarbeitungspipeline

2.2.2 Die MODELL-Datei

Meistens möchte man eine Zeitreihe auf verschiedene Art und Weise vorverarbeiten. Zu diesem Zweck kann man einem Typ von Zeitreihen mehrere Vorverarbeitungspipelines zuordnen. In einer Datei `MODELL`, die vom jeweiligen Projekt abhängt und daher im Verzeichnis `$PROJEKT_HOME/<Projekt>` zu finden ist, werden nun alle Typen mit den gewünschten Pipelines aufgeführt. Zusätzlich ist jeder Pipeline noch ein Kürzel zugeordnet, welches an den Namen der Zeitreihe angehängt wird. Dadurch sind die Ergebnisse aller Pipelines einfach zu unterscheiden.

Eine Zeile der `MODELL`-Datei hat also folgenden Aufbau:

```
Typ Kürzel Pipeline
```

Ein Beispiel für eine `MODELL`-Datei ist in Anhang A.5 zu sehen.

Nun kann beim Vorverarbeiten einer Zeitreihe einfach ein Typ aus der `MODELL`-Datei angegeben werden, und es entsteht für jede Pipeline des Typs eine neue vorverarbeitete Zeitreihe.

2.3 Prognose

Als Prognose-Verfahren sind bereits Prognosen mit Hilfe Neuronaler Netze und statistischer Methoden eingebaut. Die Neuronalen Netze werden durch SNNS [Zel95, Mor98] und ein im Fachbereich entwickeltes BackPropagation-Tool [Mid95] für Parallelrechner unterstützt, auf Seiten der Statistik sind Regression und ARIMA [Apt96a, Apt96b, Apt96c, Apt96d] möglich.

2.3.1 Neuronale Netze

Die verschiedenen Verfahren der Neuronalen Netze werden alle auf die gleiche Art behandelt: Es werden eine zu prognostizierende Zeitreihe und nach Bedarf weitere eingehende Zeitreihen bestimmt. Die Zeitreihen werden passend für das Netz vorverarbeitet (bzw. hier die bereits fertig vorverarbeiteten Zeitreihen geladen) und danach in Pattern umgruppiert. Ein Input-Pattern enthält von jeder Zeitreihe eine gewisse Menge Informationen aus der Vergangenheit. Das Training legt nun für eine Teilmenge aller Input-Pattern ein dazu passendes Output-Pattern an. Dies Verfahren schiebt also ein Zeitfenster über alle Zeitreihen (der sogenannte Fenster-Algorithmus, siehe Abbildung 2.3). Nach Beendigung des Trainings kann man mit einem Input-Pattern (zu dem noch kein Output bekannt ist) eine Abfrage starten und erhält einen Prognosewert. Die Abfrage kann für mehrere Pattern wiederholt werden, es entsteht eine Zeitreihe von abgefragten Prognosewerten. Diese abgefragten Prognosewerte werden rückskaliert und ergeben die gewünschte Prognosezeitreihe.

Zur Generierung der Pattern wird das Perl-Skript `fe2bp.pl` benutzt, das als Parameter das Verfahren (BP oder SNNS), den Namen der Steuerdatei, den Modus (`CREATE` für die erste Prognose oder `APPEND` für zusätzliche Prognosen), den Kommentar und optional die Komponenten der

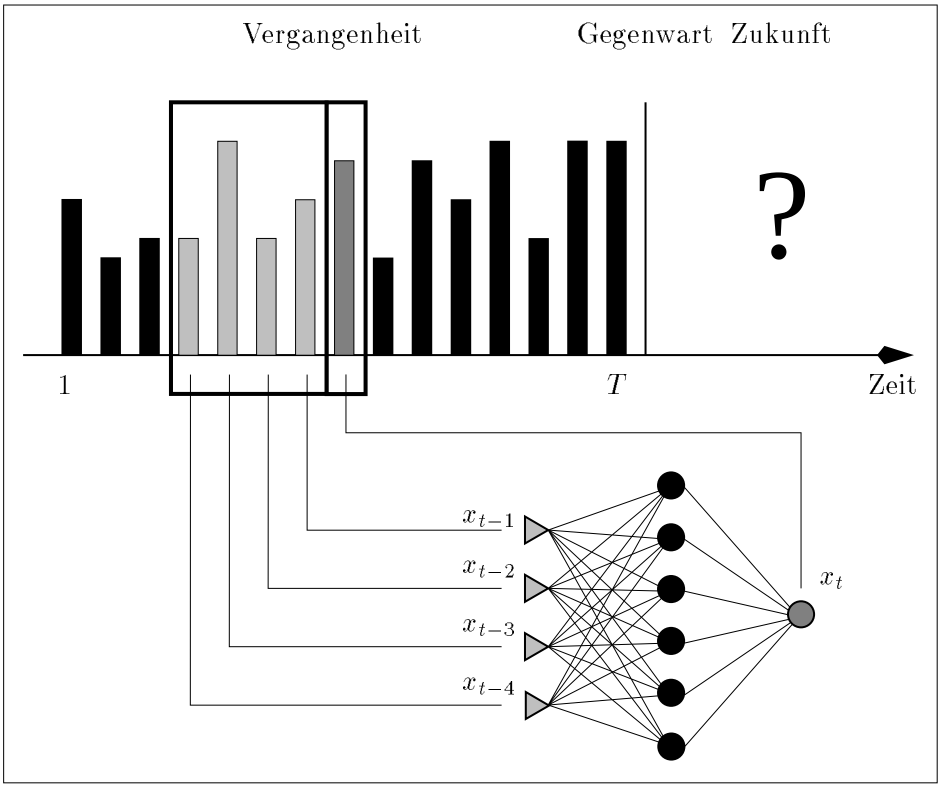

Abbildung 2.3: Pattern-Generierung bei Neuronalen Netzen

zu prognostizierenden Zeitreihe erhält. Es werden alle nötigen Pattern-Dateien zum Training, Test und zur Prognose erstellt. Zusätzlich werden alle Skripte erstellt, die benötigt werden, um das Neuronale Netz zu starten. Nach dem Aufruf des Neuronalen Netzes wird mit Hilfe des Perl-Skriptes `create_prognose.pl` die Ausgabe des Netzes in eine Prognosezeitreihe umgewandelt. Als Parameter wird wieder das Verfahren (BP oder SNNS) und die vom Netz erzeugte(n) Log-Datei(en) angegeben. Da Skript trägt die Zeitreihen im Index und in der Prognoseliste ein.

2.3.2 Statistische Methoden

Statistische Methoden wie Regression oder ARIMA werden auf dieselbe Art angewandt, wie die Neuronalen Netze. Zuerst werden die Einflußgrößen bestimmt und mit demselben Fenster-Algorithmus wie bei den Neuronalen Netzen einem Ausgabewert zugeordnet. Mit Hilfe dieser Zuordnungen berechnet man die Modell-Parameter der Methode. Die zu prognostizierenden Werte werden dann aus den gerade ermittelten Parametern und den eingehenden, nicht zur Berechnung benutzten Werten gewonnen.

Die statistischen Verfahren werden zur Zeit mit Hilfe des Gauss-Systems implementiert, zur Prognose stehen zwei Perl-Skripte zur Verfügung: `arima.pl` prognostiziert die Zeitreihe mit dem ARIMA-Verfahren, das Skript `regression.pl` führt eine lineare Regression durch. Als

Parameter bekommen beide Skripte den Namen der Steuerdatei und den Kommentar. Zusätzlich können die Komponenten der zu prognostizierenden Zeitreihe angegeben werden.

Durch die ähnliche Vorgehensweise bei Neuronalen Netzen und statistischen Methoden können Verfahren aus beiden Teilgebieten der Prognoseverfahren leicht miteinander verglichen werden, da es möglich ist, beiden Verfahren die gleichen Eingabezeitreihen zu liefern.

2.4 Batchbetrieb

Im Prototyp ist es möglich, die Prognosen aus einer graphischen Oberfläche zu starten. Ein großer Vorteil des Systems ist aber, daß ebenfalls Prognoseläufe im Batchbetrieb, d.h., ohne Interaktion mit dem Benutzer, gestartet werden können. Zu jedem Lauf wird in einer Steuerdatei festgelegt, welche Zeitreihen benutzt werden, welches Verfahren zur Prognose verwandt wird und welche Parameter dabei eingestellt werden. Danach wird der Lauf gestartet und das Ergebnis wird als Prognosezeitreihe gesichert. Auf diese Weise kann man eine große Anzahl von Testläufen unbeaufsichtigt laufen lassen, ganze Testreihen können beispielsweise über Nacht abgearbeitet werden.

2.5 Nachteile des bisher eingesetzten Systems

Der entwickelte Perl-Prototyp ist in der Lage, Zeitreihen zu verwalten und zu prognostizieren. Außerdem hat er seine Zuverlässigkeit in langen Tests während der Projektphase gezeigt. Warum sollte man diesen Prototyp, der seine Lauffähigkeit bereits bewiesen hat, nun durch ein neues System ersetzen?

2.5.1 Interpreter vs. Compiler

Perl ist eine Interpreter-Sprache. Das heißt, daß die Skripte nicht compiliert werden, sondern zur Laufzeit interpretiert werden. Außerdem muß man in Perl nicht jede Variable oder Methode im Voraus definieren, bei Verwendung eines neuen Variablennamens wird einfach eine neue Variable erzeugt. Zusätzlich sind alle Variablen typenlos. Dadurch kann man leicht und schnell lauffähige Skripte programmieren, ebenso fallen Änderungen am System leicht.

Allerdings leidet dabei die Möglichkeit, das System zu warten. Auch eine fehlende Typ-Prüfung führt manchmal zu schwer auffindbaren Fehlern. Und durch die Interpretation des Codes ist auch die Laufzeit nicht optimal. Dies zeigte sich insbesondere bei langen Zeitreihen und viel Gleitkommarechnung (z.B. bei Vorverarbeitungen wie dem exponentiellem Glätten).

In Punkto Typprüfung und Geschwindigkeit ist also durch die Wahl einer compilierbaren Programmiersprache ein deutlicher Vorteil gegeben. Ebenso werden durch moderne Compiler einige Flüchtigkeitsfehler bei der Programmierung eher gefunden. Insbesondere betrifft dies falsche oder ungültige Variablennamen.

Diese Vorteile wiegen in einer so rechenintensiven Umgebung wie der Zeitreihenverwaltung die Nachteile gegenüber einem Interpreter auf: Der Code muß für jede Architektur neu übersetzt werden und Änderungen sind aufwendiger und zeitintensiver. Der Nachteil der Systemabhängigkeit kann durch portable Programmierung eliminiert werden (insbesondere bei verbreiteten Sprachen wie C oder C++), der zweite Nachteil der Erweiterung wird z.B. durch vorher festgelegte Schnittstellen gemildert.

2.5.2 Vorverarbeitungen zur Laufzeit

Durch die höhere Ausführgeschwindigkeit kann ein Nachteil im bestehenden System behoben werden: Die Zeitreihen wurden einmalig skaliert und danach gesichert (Abbildung 2.4).

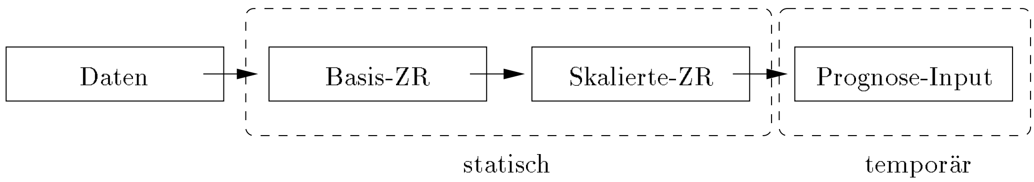

Abbildung 2.4: Bisherige Verarbeitungspipeline

Zur Prognose können nur Zeitreihen benutzt werden, die statisch vorliegen, d.h., es können nur solche vorverarbeitete Zeitreihen benutzt werden, die im Voraus berechnet wurden. Möchte man den Einfluß einer Vorverarbeitung auf die Prognose testen, beispielsweise bei einer Vorverarbeitung den Einfluß eines Parameters bestimmen, so muß man für jeden zu testenden Parameter eine Zeitreihe vorverarbeiten und speichern. Dies ist jedoch sehr aufwendig und verbraucht viel Speicherplatz.

Beispiel 2.1:
Eine Zeitreihe wird durch eine Vorverarbeitung mit 2 Parametern verändert. Möchte man jetzt jeden Parameter mit 10 verschiedenen Werten testen, so muß man 100 verschiedene Zeitreihen im Voraus berechnen und speichern. Möchte man diese Vorverarbeitung nicht nur für eine Zeitreihe testen, sondern für beispielsweise 50 verschiedene, so fallen insgesamt 5000 Zeitreihen an.

Eine Vorverarbeitung zur Laufzeit macht die Änderungen an einem Parameter *on the fly* möglich. Ein neues System sollte also weniger statische Daten aufweisen, wie in Abbildung 2.5 gezeigt.

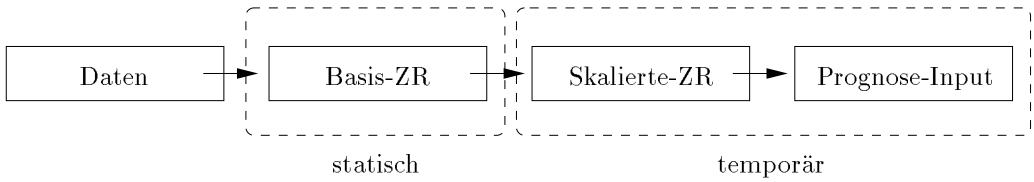

Abbildung 2.5: Gewünschte Verarbeitungspipeline

Weiterhin soll es aber möglich bleiben, die Art der Vorverarbeitungen in einer Pipeline anzugeben, d.h., man muß aus der Menge aller Vorverarbeitungen zur Laufzeit eine anhand ihres Namens auswählen können.

Kapitel 3

Ein objektorientierter Ansatz

In diesem und dem nächsten Kapitel wird ein objektorientierter Ansatz entwickelt, der die vorhandene Funktionalität enthält und einfach erweiterbar ist. In diesem Kapitel wird zuerst die Idee des Ansatzes erläutert, im folgenden Kapitel folgt die Umsetzung in eine C++-Codierung inklusive der Beschreibung aller Schnittstellen.

3.1 Modellierung des Problems mit Objekten

Objektorientierung ist einfach zu beschreiben, indem man die gewünschte Funktionalität mit Hilfe von „Dingen" verdeutlicht (die Bezeichnung „Objekt" wird hier vorerst bewußt vermieden). Verschiedene Dinge können verschiedene Aufgaben erledigen und mit anderen Dingen kommunizieren. Man muß also das gegebene Problem zuerst stark abstrahieren, damit man einen Ansatz für die Modellierung bekommt.

Das Modell für eine objektorientierte Programmierung sollte bereits geplant werden, bevor man mit der Implementation beginnt. Während der Implementation kann das gewählte Modell meist nur noch mit erheblichem Aufwand geändert werden. aus diesem Grund wird hier der Ansatz in einer allgemeinen Form dargestellt.

Zeitreihen lassen sich recht einfach als Objekte modellieren: Sie haben Werte und Informationen, die sich als Objekt-Variablen widerspiegeln, und Methoden, um die Werte und Informationen abzufragen und zu ändern. Es macht allerdings keinen Sinn, die gesamte Funktionalität der Zeitreihe in einem Objekt zu vereinen. Dadurch wäre die Erweiterung nicht mehr ohne weiteres gegeben.

Die Idee, verschiedene Aufgaben der Zeitreihe auf verschiedene Objekte zu verteilen, liegt also nah.

3.1.1 Aufgabengebiete einer Zeitreihe

Was sind nun die Hauptaufgaben einer Zeitreihe? Die wichtigste ist sicherlich die Verwaltung der Werte. Diese Aufgabe sollte im Objekt der Zeitreihe verankert bleiben. Ebenso die Verwaltung der zusätzlichen Informationen aus dem Kopf der Zeitreihe. Zusätzlich treten aber noch weitere klar eingegrenzte Teilaufgaben hervor.

Persistenz

Persistenz bezeichnet die Speicherung der Zeitreihe auf einem nichtflüchtigen Speicher. Somit bleiben die Daten auch nach einem Ausschalten des Rechners erhalten (siehe Abschnitt 4.4).

Die persistente Speicherung einer Zeitreihe kann in ein weiteres Objekt ausgelagert werden, dadurch kann man einfach durch Änderung dieses Objekts das Datenformat der Zeitreihe auf der Festplatte ändern. Wird es zudem möglich gemacht, das für die Speicherung zuständige Objekt zur Laufzeit aus einer größeren Menge von vorhandenen Speicherungsobjekten zu wählen, kann auf diese Art einfach ein Konverter zwischen verschiedenen Datenformaten implementiert werden, indem man zum Laden und Speichern unterschiedliche persistente Objekte benutzt.

Zeiteinheiten

Jede Zeitreihe wird auf eine bestimmte Zeiteinheit bezogen gemessen. Leider sind die verschiedenen Zeiteinheiten aber nicht einfach durch ein Vielfaches untereinander umzurechnen, man denke nur an die Umrechnung von Tagen in Monate. Um nicht alle Informationen über alle Zeiteinheiten in die Zeitreihe mit aufzunehmen, sollte auch diese Gruppe ausgelagert werden. Weiterhin besteht dadurch später die Chance, bisher nicht benötigte Zeiteinheiten einfach hinzuzufügen.

Vorverarbeitungen

Objekte, die Zeitreihen verändern, sind alle Arten von Vorverarbeitungen. Die häufigsten davon sind die Skalierungen, aber auch Umrechnungen zwischen verschiedenen Zeiteinheiten fallen in diese Sparte. Hier ist eine Auswahl der Vorverarbeitung durch eine textuelle Steuerung zur Laufzeit im Hinblick auf den Einsatz im Batchbetrieb sehr wichtig.

Prognosen

Die Prognose von Zeitreihen ist ein weiteres Gebiet, das in einem Objekt modelliert werden kann. Da bereits mehrere Verfahren implementiert wurden und im Einsatz sind, sollten auch diese in einer separaten Klasse abgebildet werden. Prognosen bekommen eine Auswahl von Zeitreihen als Eingabe und erzeugen daraus eine neue Zeitreihe, also ist hier eine Art „Zeitreihenfabrik“ gegeben.

Projekte

Solange man nur eine Art von Zeitreihe zu bearbeiten hat, spielt das Projekt nur eine untergeordnete Rolle. Hat man allerdings mehrere verschiedene Zeitreihen, z.B. einerseits Absatzzahlen eines Supermarktes, andererseits ökologische Meßwerte von CO_2-Emissionen, so wird ein weiteres Problem sichtbar: Die Zeitreihe muß etwas über ihre Entstehung wissen, beispielsweise ihren Namen oder die Art ihrer Einheiten. Eine Auslagerung macht auch hier eine einfache Erweiterung möglich.

3.1.2 Zusammenfassung

Insgesamt haben wir also eine Reihe von Aufgabenbereichen herausgearbeitet, die alle von der Zeitreihe benutzt werden. Die Trennung in mehrere Bereiche ist stark durch die einfache Erweiterbarkeit motiviert worden. Wird dieses nicht bereits bei der Planung des Klassenmodells einbezogen, kann dies später nur noch sehr schwer nachgebessert werden. Abbildung 3.1 zeigt die Aufteilung im Überblick.

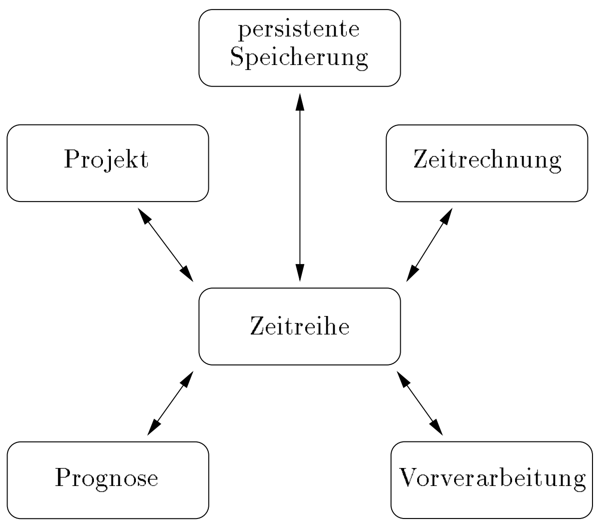

Abbildung 3.1: Aufgabenfelder einer Zeitreihe

3.2 Klassenhierarchie

Anhand der Aufteilung in Zeitreihe und fünf weitere Bereiche ergibt sich die grundlegende Klassenhierarchie, die in Abbildung 3.2 dargestellt wird.

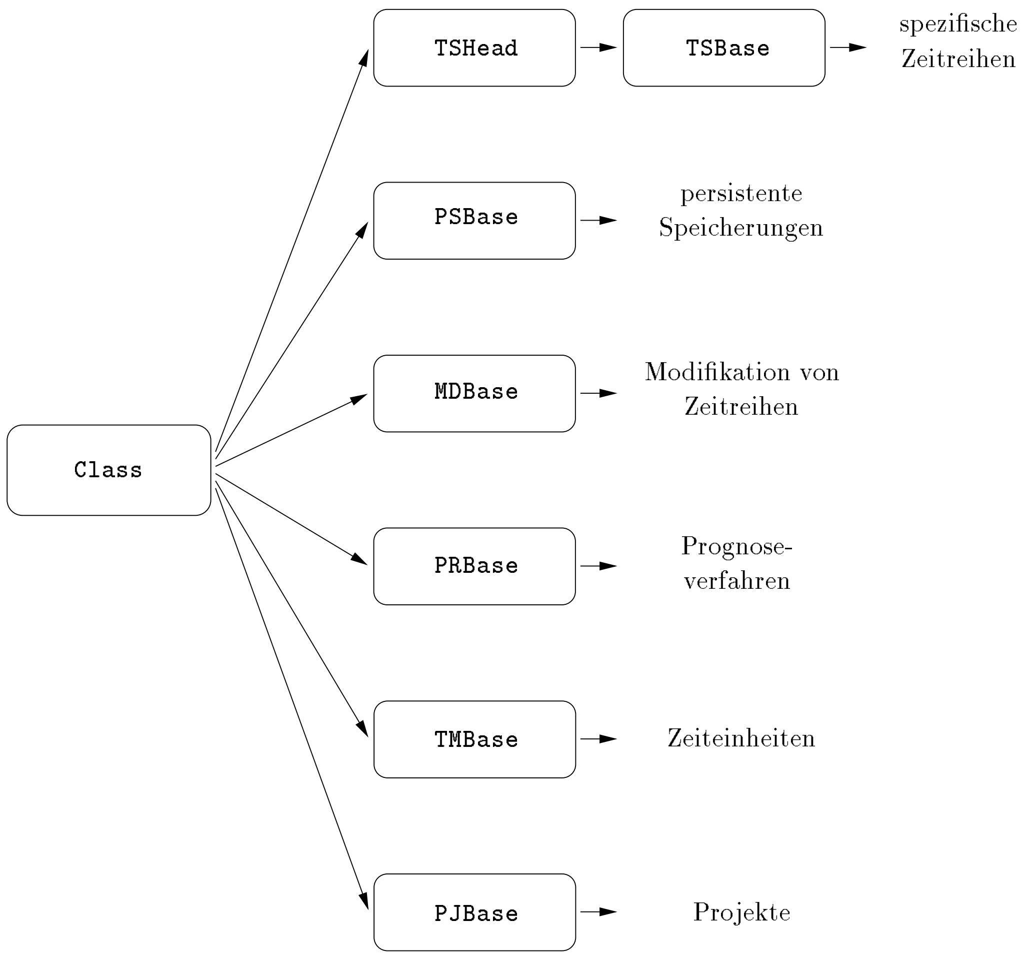

Abbildung 3.2: Ansatz für den Klassenbaum

Die Klasse `Class` erfüllt hierbei administrative Aufgaben, die im nächsten Kapitel erklärt werden. Die beschriebenen Aufgabenbereiche der Zeitreihe werden durch eine Basisklasse repräsentiert. Dadurch definiert man eine Schnittstelle, an die sich alle weiteren abgeleiteten Klassen halten müssen und die zur Kommunikation zwischen den verschiedenen Arbeitsschritten benutzt wird. Die Erweiterung der Klassenhierarchie darf ausschließlich durch Vererbung bereits bestehender Klassen erfolgen.

3.3 Klassen über den Namen identifizieren

Im bestehenden System war es möglich, eine Prognose mit allen zur Verfügung stehenden Parametern, inklusive der beteiligten Zeitreihen, in einer Steuerdatei anzugeben und damit eine Prognose im Batchbetrieb zu starten (siehe Abschnitt 2.4). Dies soll natürlich auch mit dem objektorientierten Modell möglich sein. Zu diesem Zweck muß man aus der textuellen Beschreibung einer Klasse eine Instanz der Klasse erstellen können.

3.3.1 Warum Referenzierung über Namen ?

Bisher stand im Steuerfile immer nur der Name einer bereits vorverarbeiteten Zeitreihe. Da diese Vorverarbeitung jetzt zur Laufzeit erfolgen soll, tritt an diese Stelle der Name der unvorverarbeiteten Zeitreihe und eine Liste von Vorverarbeitungen. Die Verbindung zwischen textueller Referenz (im Steuerfile) und Vorverarbeitung (im Klassenbaum) muß also möglich sein.

Beispiel 3.1:
Im Steuerfile steht eine Zeile der folgenden Form:

```
INPUT PRESENCE CO2|Tage.CO2(5) DIFF 1 1 | LINEAR 0.1 0.9
```

`INPUT` bezeichnet dabei eine in die Prognose eingehende Zeitreihe, `PRESENCE` sagt aus, daß diese Zeitreihe zum Prognosezeitpunkt (in der Zukunft) nicht bekannt ist. Danach folgt der Name der Zeitreihe (`CO2|Tage.CO2`) mit einem Parameter `(5)`, der angibt, wieviele Werte als Input benutzt werden. Danach folgt eine Verarbeitungspipeline (siehe auch Abbildung 2.2). Aus den Texten `DIFF` und `LINEAR` müssen nun die zugehörigen Objekte erzeugt werden. Ähnliche Probleme ergeben sich, da in der Steuerdatei auch das zu verwendende Prognoseverfahren sowie die gewünschte Zeiteinheit angegeben wird.

3.3.2 Konventionen für Klassennamen in der Hierarchie

Damit die Klassen anhand ihres Namens gefunden werden, müssen alle Klassen im Projekt bestimmten Richtlinien entsprechen. Fast alle Klassen haben ein Kürzel vor dem eigentlichen Namen, das die Gruppe des Objekts beschreibt (z.B. `MD` für Modifikationen, d.h. Vorverarbeitungen oder `PS` für persistente Speicherung).

Eine Übersicht über alle verfügbaren Kürzel ist in Tabelle 3.1 angegeben.

Weiterhin müssen das Kürzel und das direkt daran anschließende Zeichen groß, der Rest klein geschrieben werden. Ansonsten wird die Klasse nicht zur Laufzeit erkannt. Diese beiden Einschränkungen in Bezug auf den Klassennamen verhindern, daß einerseits mehrere Klassen einen

Tabelle 3.1: Liste aller Präfixe

Kürzel	Ursprung	Gruppe von Objekten
`TS`	*TimeSeries*	Zeitreihen
`PS`	*PerSistent*	persistente Speicherung
`MD`	*MoDifier*	Vorverarbeitungen
`PR`	*PRediction*	Prognose-Verfahren
`TM`	*TiMe*	Zeiteinheiten
`PJ`	*ProJect*	Projekte

Namen bekommen und andererseits Klassen identifiziert werden, die nicht in die zugehörige Gruppe gehören. Eine Klasse der Vorverarbeitungsgruppe `MD` kann beispielsweise nicht als Zeitreihen-Objekt der Gruppe `TS` identifiziert werden.

Beispiel 3.2:
Im Beispiel 3.1 wurde eine Zeile einer Steuerdatei angeführt. Bei der Suche nach den beiden Vorverarbeitungsklassen erhält man folgendes Ergebnis: Der Text `DIFF` wird dem Klassennamen `MDDiff` zugeordnet, der Text `LINEAR` dem Namen `MDLinear`. Daraufhin wird geprüft, ob die Klassen existieren. Bei einem Erfolg werden Instanzen der Klassen erzeugt und die Methoden aufgerufen, die die Zeitreihe vorverarbeiten.

3.3.3 Verbindung von Klassenname und Klasse

Um die Verbindung zwischen dem Klassennamen und der Klasse herzustellen, muß eine Liste existieren, die ähnlich einer Hashtabelle arbeitet. Dabei ist der textuelle Klassenname der Schlüssel, der Wert kennzeichnet die Klasse (dazu mehr im nächsten Kapitel). In einigen Programmiersprachen, z.B. Java, ist ein solcher Mechanismus bereits eingebaut. In anderen Sprachen dagegen muß eine solche Liste selbst implementiert werden, dazu gehört auch die hier benutzte Sprache C++.

Jede Klasse muß sich selbständig in die Liste aller Klassen einhängen, eine zentrale Methode, die alle Klassen registriert, verhindert die Erweiterbarkeit ohne Eingriff in den bestehenden Code (denn eine zentrale Methode müßte alle Klassen kennen, die sie registrieren soll).

Weitere Details zur Registrierung folgen im nächsten Kapitel (siehe Abschnitt 4.2).

Kapitel 4

Umsetzung in C++

In diesem Kapitel wird der objektorientierte Ansatz des letzten Kapitels mit Hilfe der Programmiersprache C++ umgesetzt. Dabei werden die Klassen nur grundlegend angesprochen. Eine Liste aller Klassen mit einer Beschreibung aller Methoden ist im Kapitel 7 angeführt.

4.1 Warum C++ ?

Die Sprache C++ wurde gewählt, da sie zum Startzeitpunkt die verbreitetste objektorientierte Sprache war. Als zweite in Frage kommende Programmiersprache ist Java anzusehen, insbesondere, da Java bereits einen Mechanismus zur Verfügung stellt, der anhand eines Klassennamens (in Textform) die zugehörige Klasse suchen kann. Allerdings war bei Java die Ausführungsgeschwindigkeit noch nicht hoch genug (da die Sprache ebenfalls interpretiert wird) und es standen keine leistungsfähigen Just-In-Time Compiler zur Verfügung.

4.2 Registrierung der Klassen

Die Wahl von C++ macht es nötig, die Verbindung zwischen Klassennamen und Klasse von Hand zu implementieren. Dabei müssen alle in Abschnitt 3.3 gemachten Aussagen beachtet werden. Die Implementation dieser Referenz über Klassennamen ist wichtig für fast alle Bereiche der Arbeit, da erst dadurch eine flexible Steuerung von außen möglich ist (z.B. über ASCII-Steuerdateien).

Wie ist es also möglich, daß jede Klasse sich selbst in eine Liste von Klassen einträgt? Und welche Informationen benötigt man außer dem Namen, um eine Instanz der Klasse zu erzeugen? Ein einfaches Anlegen mit Hilfe des `new`-Operators von C++ ist leider nicht möglich, da hier im Quellcode bereits die zu erzeugende Klasse angegeben werden muß (z.B. `derWertFuenf = new int(5)`, der Typ `int` steht fest im Programmcode).

Die Lösung des ersten Problems ist durch einen Trick zu lösen. Im folgenden wird ein Ansatz schrittweise verbessert, bis man das Problem gelöst hat. Die Idee ist, daß jede Klasse ihren eigenen Namen kennen muß. Außerdem muß man die Klasse auch nach ihrem Namen fragen können. Es wird also in jeder Klasse eine Methode `getObjectName()` implementiert, die diesen Namen liefert. Ein erster Ansatz:

```
const char* MyClass::getObjectName() { return "MyClass"; }
```

Damit ist der Name der Klasse von außen sichtbar, der Name selbst steht fest im Programmcode der Klasse. Im zweiten Schritt ersetzt man den festen Text durch eine Klassenvariable, die den Namen enthält:

```
const char* MyClass::name = "MyClass";
const char* MyClass::getObjectName() { return MyClass::name; }
```

Die Klassenvariable `name` wird beim Start des Programms mit der Zeichenkette `"MyClass"` initialisiert, bei einem Aufruf von `getObjectName()` ist die Variable `MyClass::name` also immer korrekt gesetzt. Die Funktionalität des Programms hat sich im Vergleich zu der ersten Variante allerdings noch nicht geändert, ebenfalls ist noch keine Liste von Klassen ins Spiel gekommen.

Initialisiert man die Klassenvariable `name` jetzt allerdings nicht mit einem festen Text, sondern mit Hilfe einer Methode, so sieht der Programmcode so aus:

```
const char* MyClass::name = methode( "MyClass" );
const char* MyClass::getObjectName() { return MyClass::name; }
```

Die Methode `methode()` bekommt als Argument den Klassennamen und liefert diesen auch zurück. Innerhalb der Klasse `MyClass` ist also wieder keine neue Funktionalität hinzugekommen. Allerdings kann die Methode `methode()` nun den Namen in einer Liste von Namen speichern. Wenn jede Klasse den oben gezeigten Mechanismus implementiert, so wird die Methode `methode()` in allen Klassen aufgerufen und es entsteht eine Liste aller Klassennamen. Wird die Bibliothek um eine weitere Klasse erweitert, so muß man keine Änderungen am bestehenden Code vornehmen, die Klasse muß lediglich den oben dargestellten Code enthalten, um sich selbst in die Liste einzutragen. Dieser Vorgang wird im weiteren auch kurz **Registrierung** genannt.

Die Liste aller Klassennamen sollte zentral verwaltet werden, ebenso sollte die Methode zur Registrierung (im obigen Beispiel `methode()` genannt) in allen Klassen verfügbar sein. Die Lösung liegt in einer allgemeinen Basisklasse `Class`, von denen alle weiteren Klassen abgeleitet werden. Diese stellt alle nötigen Methoden zur Verfügung, um Klassen zu registrieren und registrierte Klassen zu suchen. Eine genaue Beschreibung der Klasse `Class` folgt später.

Das zweite Problem ist, daß man mit einer reinen Liste von Klassennamen nicht viel anfangen kann. Man hat zwar die Namen aller Klassen und kann entscheiden, ob eine Klasse eines bestimmten Namens existiert — man kann aber keine neue Instanz einer gefundenen Klasse erstellen. Zu diesem Zweck muß man außer dem Namen noch weitere Informationen zu einer Klasse speichern, d.h., es müssen noch weitere Informationen bei der Registrierung angegeben werden.

Man benötigt einen Mechanismus, der eine Instanz der Klasse erstellt. Mit der folgenden Methode ist dies möglich:

```
MyClass* MyClass::anew() { return new MyClass(); }
```

Ruft man die Methode `MyClass::anew()` auf, so erhält man eine neue Instanz der Klasse `MyClass` zurück. Hierbei treten allerdings noch zwei Probleme auf: Bei einer solchen Implementation liefert jede Klasse einen anderen Typ zurück, und es können nur Konstruktoren ohne Parameter aufgerufen werden. Das erste Problem kann man lösen, indem man einen Zeiger auf die zugrundeliegende Basisklasse liefert. Dies ist für alle abgeleiteten Klassen möglich:

```
Class* MyClass::anew() { return new MyClass(); }
```

Danach kann man mit Hilfe von Casts auf eine tiefergelegene Klasse kommen:

```
MyClass *object = (MyClass*)MyClass::anew()
```

Um parameterbehaftete Konstruktoren aufrufen zu können, muß man einen weiteren Trick anwenden: Die Methode `anew()` bekommt jetzt ein Objekt übergeben, welches die Parameter für einen Konstruktor enthält. Für diesen Zweck existiert eine Basisklasse `ClassInit`, die allerdings keine Parameter speichert. Von dieser Klasse können Unterklassen abgeleitet werden, die dann spezifische Parameter einer Klasse enthalten. Es empfiehlt sich, den Klassennamen der zugehörigen Klasse gefolgt von `Init` zu benutzen, so sollte die Initialisierungsklasse von `MyClass` etwa `MyClassInit` heißen. Die Methode `anew()` ist jetzt wie folgt definiert:

```
Class* MyClass::anew( ClassInit *init )
```

Innerhalb der Methode `anew()` wird das Objekt `init` ausgewertet und der passende Konstruktor aufgerufen.

Um eine Instanz der Klasse zu erzeugen, muß also jede Klasse die Methode `anew()` implementieren und bei der Registrierung einen Zeiger auf diese Methode mit angeben. Kann von einer Klasse keine Instanz erzeugt werden (z.B. bei abstrakten Klassen), so muß ein `NULL`-Zeiger übergeben werden.

Da wir ja bereits den Namen der Klasse in eine Liste eintragen, liegt es nahe, ebenfalls den Namen der Vaterklasse zu übergeben. Aus diesen beiden Informationen kann man dann einen kompletten Klassenbaum rekonstruieren. Damit kann man später prüfen, ob eine Klasse ein Nachfolger (direkt oder indirekt) einer anderen Klasse ist. Zu diesem Zweck muß jede Klasse noch die Methode `getParentName()` implementieren, die den Namen der Vaterklasse liefert. Am einfachsten geht dies, indem die Methode `getObjectName()` der Vaterklasse aufgerufen wird.

Alle drei Parameter (Name der Klasse, Name der Vaterklasse und `anew()`-Methode) reichen aus, um aus einem Namen eine Instanz der Klasse zu erstellen und grundlegende Prüfungen der Klassenhierarchie zuzulassen. Um den Mechanismus offen für Erweiterungen zu halten, werden allerdings noch zwei weitere Parameter übergeben. Es ist möglich, vor dem Ende des Programms eine Klassenmethode in jeder Klasse aufzurufen, die noch nötige Aufräumarbeiten durchführt (z.B. temporäre Dateien löscht):

```
void MyClass::cleanup( void )
```

Als letztes übergibt man eine allgemeine Methode, deren Funktionalität keiner Einschränkung unterliegt und abhängig von der umgebenden Klasse implementiert werden kann. Die Methode kann man als große `switch`-Anweisung ansehen: Sie bekommt einen `int`-Wert und eine variable Argumentliste, die abhängig vom übergebenen ersten Wert gestaltet werden kann:

```
int MyClass::hook( int, va_list args )
```

Wie auch schon bei `anew()` wird hier der Wert `NULL` übergeben, falls die Methoden in der Klasse nicht zur Verfügung stehen. Die Methode `hook()` wird zur Zeit nur in den Subklassen von `TSBase` benutzt (siehe Abschnitt 4.3.4).

Insgesamt sieht der Aufruf zum Registrieren einer Klasse also wie folgt aus:

```
const char* MyClass::name = registerClass(
    "MyClass",
    "MyParent",
    MyClass::anew,          // oder NULL
    MyClass::cleanup,       // oder NULL
    MyClass::hook);         // oder NULL
```

Die Klasse `Class` implementiert nun die Methode `registerClass()`: Die übergebenen Werte werden in einer Datenklasse `ClassData` gesichert und in einer Liste von Datenklassen verwaltet. Anhand eines Namens kann man nun entweder das zugehörige `ClassData`-Objekt aus der Liste anfordern (um beispielsweise die `hook()` Methode aufzurufen) oder auch direkt eine neue Instanz der zugehörigen Klasse erzeugen lassen.

Mit diesem Mechanismus kann man nun Klassen über den Namen suchen und erzeugen. Damit wäre die Grundlage einer textuellen Steuerung gegeben.

4.3 Das Zeitreihen-Objekt

Als nächstes benötigen wir ein allgemeines Zeitreihenobjekt. Da für manche Zwecke nur die Informationen einer Zeitreihe wichtig sind, liegt es nahe, den Kopf als allgemeinste Klasse zu formulieren.

Durch Ableiten der Kopf-Klasse erhält man zuerst eine allgemeine Zeitreihe, die zusätzlich zum Kopf auch Werte verwalten kann. Von diesem allgemeinen Zeitreihenobjekt kann man weitere Klassen ableiten, die einen speziellen Aufbau einer Zeitreihe voraussetzen. In der vorliegenden Arbeit wurde dies exemplarisch mit der Unterscheidung in boolsche (oder binäre) und numerische Zeitreihen dargestellt.

Für sämtliche Zeitreihenobjekte ist als Präfix des Klassennamens `TS` (für *TimeSeries*) vorgeschrieben.

4.3.1 Der Kopf als Objekt: `TSHead`

Der Kopf einer Zeitreihe wird in der Klasse `TSHead` abgebildet. Diese Klasse kann alle Kopf-Informationen einer Zeitreihe speichern, nicht jedoch die Werte der Zeitreihe.

Informationen, die in allen Arten von Zeitreihen benutzt werden, werden hier in Objektvariablen gespeichert. Dazu gehören Start- und Endzeitpunkt, Zeit- und Werteeinheit, Komponenten, Typ und Name der Zeiteinheit, Skalierungen und Basiszeitreihe, Typ der Aggregation und eine Aussage, ob die Zeitreihe aus einer Prognose entstanden ist. Diese Werte können über spezielle Methoden abgefragt werden (z.B. `getStart()`).

Die Klasse verwaltet zusätzlich eine Liste aller restlichen Kopf-Informationen, die aus einem Schlüssel, einem Wert und einem optionalen Kommentar bestehen. Eine einzelne Information wird in der Klasse `TSHeadInfo` gespeichert. Die Informationen können mit allgemeinen Methoden gesetzt, abgefragt und gelöscht werden (`getInfo()`, `setInfo()`).

Weiterhin besitzt die Klasse einen Projekt-Manager, der die referenzierten Projektobjekte verwaltet und in diesen auch suchen kann (siehe Abschnitt 4.8.3).

Die Basisklasse `PSBase` für die persistente Speicherung (siehe Abschnitt 4.4) ist in C++ als `friend`-Klasse angegeben, da diese die internen Strukturen des Kopfes ändern können muß. In C++ ist der `friend`-Mechanismus wie folgt geregelt:

Beispiel 4.1:

```
class ClassOne {
    friend class ClassTwo;
    protected: int secureInt;
    public:    int publicInt;
}
```

In diesem Fall kann auf die Variable `publicInt` von außen zugegriffen werden, die Variable `secureInt` ist nur in `ClassOne` und ihren Unterklassen sichtbar. Durch die `friend` Anweisung kann allerdings auch die Klasse `ClassTwo` auf die Variable `secureInt` zugreifen (nicht aber die Unterklassen von `ClassTwo`).

Initialisierungsobjekt für `TSHead`: `TSHeadInit`

Die Klasse `TSHeadInit` ist eine Subklasse von `ClassInit` und nimmt die Argumente auf, um einen neuen Zeitreihenkopf zu erzeugen (siehe Abschnitt 4.2):

```
TSHeadInit( TSHead *head, double def, int grow )
```

Wird mit Hilfe des Initialisierungsobjekts eine Instanz von `TSHead` erzeugt, so wird ein existierender Kopf `head` kopiert, dazu werden ein Standardwert `def` für unbelegte Ticks und ein Flag `grow` eingetragen, das angibt, ob die Zeitreihe selbständig wachsen soll.

Beispiel 4.2:
Die Zeitreihe ist im Bereich $t \in [50, 100]$ definiert. Beim Zugriff auf die Werte innerhalb des Definitionsbereichs werden die eingetragenen Werte geliefert. Ein Zugriff auf Werte außerhalb des Definitionsbereichs, beispielsweise $t' = 20$, liefert den Standardwert `def`. Ist das Flag `grow` auf `0` gesetzt, so wird die Zeitreihe nicht geändert. Mit `grow == 1` wird allerdings der Definitionsbereich erweitert, hier wäre die Zeitreihe jetzt im Bereich $t \in [t' = 20, 100]$ definiert, die Werte für $t \in [20, 49]$ werden mit dem Standardwert `def` initialisiert.

4.3.2 Erweiterung auf komplette Zeitreihe: `TSBase`

Eine allgemeine Zeitreihe entsteht aus `TSHead`, indem man Werte hinzufügt. Durch Erweitern der Klasse erhält man `TSBase`. Diese Klasse erbt alle Methoden von `TSHead` und implementiert selbst alle nötigen Methoden, um Werte zu speichern. `TSBase` selbst kann jedoch noch keine statistischen Aussagen über die Zeitreihe machen (wie zum Beispiel den Mittelwert).

Es ist möglich, daß manche Zeitreihen Lücken haben, etwa verursacht durch fehlende Meßwerte. Diese sogenannten *Missing-Werte* werden in der Zeitreihe mit dem Wert `TSBase::MISS` gekennzeichnet. Da hier kein numerischer Wert benutzt werden darf, ist `TSBase::MISS` die Darstellung für `NaN` (*Not A Number*). Beim Erzeugen einer Zeitreihe kann zusätzlich angegeben werden, welcher Wert als Initialisierung der Zeitreihe benutzt wird. Dieser Wert wird auch geliefert, falls man Werte außerhalb des Wertebereiches der Zeitreihe abfragt (siehe Beispiel 4.2). Dieser Standardwert kann, muß aber nicht `TSBase::MISS` sein.

Es wird nicht im einzelnen auf jede Methode eingegangen, insgesamt gibt es aber verschiedene Gruppen von Methoden in dieser Klasse:

Konstruktoren:
Es gibt Konstruktoren, die neue Zeitreihen erzeugen und deren Werte auf einen Standardwert setzen, und Konstruktoren, die existierende Zeitreihen oder Zeitreihenköpfe kopieren.

Methoden im Zusammenhang mit den Zeitreihenwerten:
Man kann alle Werte einer Zeitreihe abfragen (`getValue()`) oder verändern (`setValue()`). Die Abfrage liefert bei nicht gesetzten Werten den Standardwert der Zeitreihe zurück.

Neuberechnung von Informationen der Zeitreihe:
`recalc()` wird benutzt, um Informationen zu aktualisieren, die direkt mit den Werten der Zeitreihe in Beziehung stehen (z.B. die Anzahl der Missing-Werte). `recalc()` wird intern aufgerufen, wenn eine solche Information abgefragt wird, die Zeitreihe seit dem letzten Aufruf von `recalc()` aber geändert wurde. Eine Abfrage aller Informationen wird beispielsweise beim Speichern der Zeitreihe durchgeführt.

Methoden im Zusammenhang mit der persistenten Speicherung:
Es stehen Methoden `load()` und `save()` zur Verfügung, die eine Zeitreihe laden und speichern können. Diese Methode ruft als Hauptaufgabe die passende Methode im aktivierten Persistenzobjekt auf. Um Subklassen die Möglichkeit zu geben, nach dem Laden oder vor/nach dem Speichern noch eigene Informationen im Kopf zu setzen (oder andere Operationen auf die Zeitreihe anzuwenden), existieren die Methoden `afterLoad()`, `beforeSave()` und `afterSave()`, die in `TSBase` allerdings keine Funktion haben (d.h. der Methodenrumpf ist leer).

Weiterhin existieren Methoden, um auf das Vorhandensein einer Zeitreihe zu testen (`exists()`) oder eine gespeicherte Zeitreihe zu löschen (`destroy()`).

Methoden zur Aggregation:
Es gibt Methoden, die die ganze Zeitreihe auf eine andere Zeiteinheit umrechnen (`changeTime()`) oder die mehrere Werte auf einen aggregieren (`aggregate()`). Die Methode `aggregate()` wird allerdings zur Zeit nur in der Vorverarbeitung `MDAggregate` benutzt (siehe Abschnitt 7.6.2).

Sonstige Methoden:
Weiterhin existieren Methoden, die den Wertebereich verändern oder die Zeitreihe in eine andere Zeitreihenklasse konvertieren. Diese Konvertierung ist beispielsweise nötig, wenn man eine binäre Zeitreihe vorverarbeitet und dadurch eine numerische Zeitreihe erhält (siehe Abschnitt 4.3.3 und Beispiel 4.3).

Initialisierungsobjekt für `TSBase`: `TSBaseInit`

Die Klasse `TSBaseInit` ist eine Subklasse von `TSHeadInit` und nimmt die Argumente auf, um eine neue Zeitreihe zu erzeugen (siehe Abschnitt 4.2):

```
TSBaseInit( TSBase *zr )
```

Wird mit Hilfe des Initialisierungsobjekts eine Instanz von `TSBase` erzeugt, so wird eine existierende Zeitreihe `zr` (mit allen Kopf-Informationen und allen Werte) kopiert.

4.3.3 Erweiterungen: `TSNumeric` und `TSBinary`

Von `TSBase` kann man nun beliebig weitere Klassen ableiten, die spezielle Eigenschaften einer Zeitreihe implementieren. Als Beispiel wurden die Zeitreihen `TSNumeric` und `TSBinary` implementiert.

Die Klasse `TSBinary` verwaltet nur noch boolsche Werte, die allerdings in numerischer Form gespeichert sind. Dabei wird der Wert 0.0 mit dem boolschen Wert `FALSE` identifiziert, der Wert 1.0 repräsentiert den boolschen Wert `TRUE`.

Eine binäre Zeitreihe hat die Information `Binaer:1` im Kopf.

Als zusätzliche Information wird die Häufigkeit der `TRUE`- und `FALSE`-Werte gezählt (in `recalc()`). Diese können über `getTrues()` und `getFalses()` abgefragt werden.

Da boolsche Werte anders aggregiert werden (Summation und Maximumbildung entsprechen einer `ODER`-Verknüpfung, der Mittelwert entspricht einer `UND`-Verknüpfung, und die Minimumbildung wird durch eine `NICHT-UND`-Verknüpfung ersetzt), muß die Methode `aggregate()` aus der Klasse `TSBase` in dieser Klasse überschrieben werden.

Das Gegenstück zu `TSBinary` ist `TSNumeric`. Diese Klasse speichert nur noch numerische Werte, dafür können aber auch statistische Informationen wie Minimum, Maximum oder Mittelwert abgefragt werden.

Eine numerische Zeitreihe hat die Information `Binaer:0` im Kopf.

In der implementierten Methode `recalc()` werden Minimum, Maximum, Mittelwert, Standardabweichung und Varianz berechnet. Diese Werte können über spezielle Methoden abgefragt werden (z.B. `getMean()`).

Beispiel 4.3:
Manche Vorverarbeitungen bekommen einen Zeitreihentyp als Eingabe, liefern aber einen anderen Zeitreihentyp als Ausgabe. Weist eine Vorverarbeitung beispielsweise jedem Wahrheitswert einen numerischen Wert zu (nach einem beliebigen Algorithmus), so muß die eingehende Zeitreihe ein Objekt der Klasse `TSBinary` (oder auch einer ihrer später implementierten Subklassen) sein. Die Ausgabezeitreihe ist aber vom Typ `TSNumeric` (oder ebenfalls einer Subklasse), es muß also ein neues Zeitreihenobjekt angelegt werden. Die benötigte Methode stellt die Basisklasse `TSBase` zur Verfügung (siehe Abschnitt 4.3.2).

Fortsetzung ▷

▷ Fortsetzung **Beispiel 4.3**

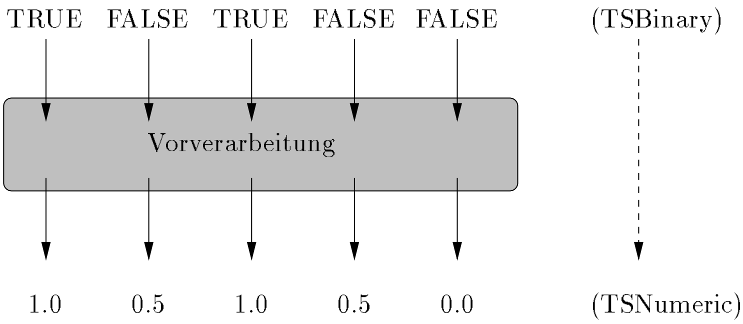

Abbildung 4.1: Konvertierung von `TSBinary` nach `TSNumeric`

4.3.4 Zeitreihen-Klasse beim Laden bestimmen

Beim Laden von Zeitreihen tritt folgendes Problem auf: Man benötigt ein Zeitreihenobjekt, um die geladenen Daten zu speichern. Allerdings kennt man vor dem Laden den benötigten Typ der Zeitreihe noch nicht, es ist z.B. unbekannt, ob es sich um binäre oder numerische Werte handelt. Demzufolge ist auch unbekannt, ob man ein Objekt der Klasse `TSBinary` oder `TSNumeric` erzeugen muß. Aus diesem Grund wird zuerst nur der Kopf der Zeitreihe in ein Objekt der Klasse `TSHead` geladen, daraufhin wird entschieden, zu welchem Typ die Zeitreihe gehört. Jetzt kann man ein Zeitreihenobjekt vom korrekten Typ erzeugen, den bereits gelesenen Kopf hineinkopieren und die Werte einlesen.

Wie kann man aus dem Kopf nun die zugehörige Zeitreihenklasse ermitteln? Die allgemeinste in Frage kommende Klasse ist `TSBase`, da diese Klasse bereits Kopf-Informationen und Werte verwalten kann. Aber wahrscheinlich kann man anhand von Kopf-Informationen eine speziellere Klasse von Zeitreihen finden, die zu dem Kopf paßt. Zur Zeit ist dies nur die Unterscheidung zwischen binären und numerischen Zeitreihen, eine Erweiterung ist aber ohne weiteres möglich.

Um die passende Zeitreihenklasse zu ermitteln, wird eine Methode `getTSClass()` der Klasse `TSHead` aufgerufen, die den Teilbaum der Klassenhierarchie mit `TSBase` als Wurzel traversiert und das zugehörige Zeitreihenobjekt zurückliefert (initialisiert mit dem übergebenen Kopf). Jede Zeitreihenklasse muß eine Methode `checkTSClass()` implementieren, die einen Zeitreihenkopf vom Typ `TSHead` bekommt und anhand der enthaltenen Informationen prüft, ob sie zuständig ist. Die Traversierung läuft nach folgendem Algorithmus ab:

1. Starte mit `Aktuell = "TSBase"` und dem geladenen Zeitreihenkopf `TSHead *head`.
2. Bestimme Liste `Soehne` aller direkten Nachfahren von `Aktuell`.
3. Ist Liste `Soehne` leer, setze Verfahren bei Schritt 7 fort.
4. Für jede Klasse `Nachfahre` aus `Soehne` prüfen, ob diese Klasse zuständig ist. Dies ermittelt man mit einem Aufruf der Methode `checkTSClass( head )` der Klasse `Nachfahre`.
5. Ist eine Klasse `Nachfahre` zuständig, so setze `Aktuell = Nachfahre` und setze das Verfahren bei Schritt 2 fort.
6. Ist keine Klasse `Nachfahre` zuständig, so setze Verfahren bei Schritt 7 fort.
7. Die Klasse `Aktuell` ist die zuständige Klasse.

Das Verfahren wird an einem Beispiel verdeutlicht:

Beispiel 4.4:
Die Zeitreihe `Tegut|AAA|229104|Wochen.ABV` soll geladen werden. Zuerst wird der Kopf der Zeitreihe in ein Objekt `TSHead head` eingelesen. Danach wird (beginnend bei `TSBase`) ein zuständiges Zeitreihenobjekt gesucht. Im ersten Schritt werden die direkten Nachfolger `TSBinary` und `TSNumeric` von `TSBase` getestet, dabei werden die Methoden `TSBinary::checkTSClass()` und `TSNumeric::checkTSClass()` aufgerufen. Der Unterschied zwischen einer binären und numerischen Zeitreihe ist in der Kopf-Information `Binaer` gespeichert. Diese Information wird in beiden Methoden geprüft. Ist die Information enthalten und deren Wert gleich `1`, so ist die Klasse `TSBinary` zuständig. Andernfalls ist `TSNumeric` die passende Klasse. Da die Zeitreihe die Information `Binaer:0` im Kopf enthält, wird in diesem Beispiel `TSNumeric` als zuständige Klasse gefunden. Da keine weiteren Klassen unterhalb von `TSNumeric` existieren, ist diese Klasse auch insgesamt die zuständige Klasse. Ein Ablauf dieser Traversierung ist in Abbildung 4.2 zu sehen.

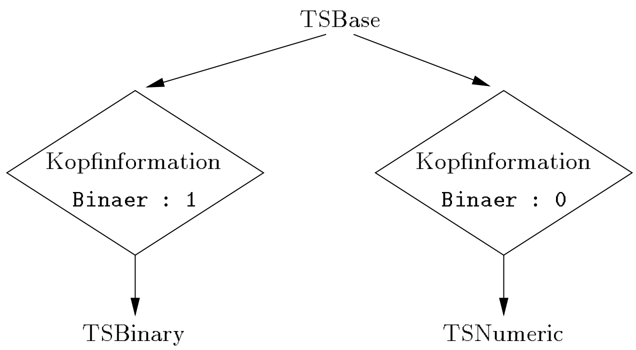

Abbildung 4.2: Beispiel: Test auf binäre oder numerische Zeitreihe

4.4 Persistente Objekte

Persistente Speicherung ist nötig, um im Hauptspeicher existierende Zeitreihenobjekte zu sichern, beispielsweise im Dateisystem. Da durchaus mehrere verschiedene Datenformate vorliegen können, existieren auch verschiedene Klassen für die Speicherung. Zur Laufzeit kann aus dieser Gruppe ein Objekt an die Zeitreihen gekoppelt werden. Ist es weiterhin möglich das persistente Objekt zur Laufzeit zu wechseln, so kann auf diese Art ein Konverter zwischen verschiedenen Datenformaten implementiert werden (siehe Abschnitt 5.3).

Die persistenten Objekte müssen die Daten aus dem vorliegenden Datenformat in das Format umrechnen, das intern in den Zeitreihenobjekten `TSHead` und `TSBase` benutzt wird. Bei den Werten ist dies einfach möglich, da hier einfach numerische Werte gespeichert werden. Alle Werte, die die Klasse `TSHead` selbst verwaltet (wie Start- und Endzeitpunkt), können mit entsprechenden Methoden der Basisklasse `PSBase` gesetzt werden. Schwieriger wird das bei den restlichen Kopf-Informationen, da diese in einer Schlüssel-Wert-Beziehung gespeichert sind. Diese Schlüssel werden beispielsweise bei der Suche einer zugehörigen Zeitreihenklasse benutzt (z.B. der Schlüssel `Binaer` im Beispiel 4.4). In der Tabelle 4.1 sind alle Schlüssel aufgelistet, die im benutzten Datenformat vorkommen und die in `TSHead` direkt unterstützt werden (d.h., der Schlüssel wird nicht benötigt, nur der Wert), die aufgeführten Methoden werden in der Klasse `PSBase` zur Verfügung gestellt (siehe unten). In der Tabelle 4.2 sind alle weiteren Schlüssel angeführt, die bisher benutzt worden sind. Diese müssen in genau dieser Form eingetragen werden, um die Funktionalität der Klassenbibliothek zu gewährleisten.

Alle Objekte, die für die persistente Speicherung zuständig sind, müssen das Präfix `PS` (für *PerSistent*) im Namen haben.

Tabelle 4.1: Schlüssel, die in `TSHead` nur als Wert gesetzt werden

Schlüssel	Methode in `PSBase`

`Start`
`Ende`
`Faktor`
	`PSBase::setTSInterval()`

`Aggregierung`	`PSBase::setTSAggreg()`
`Prognose`	`PSBase::setTSPrognose()`
`Komponenten`	`PSBase::setTSCompNo()`
`Name`	`PSBase::setTSName()`
`Basiszeitreihe`	`PSBase::setTSRoot()`
`Skalierungen`	`PSBase::addTSScale()`
`D-Beschreibung`, `E-Beschreibung`	`PSBase::setTSDesc()`
`D-ZeitEinheit`, `E-ZeitEinheit`	`PSBase::setTSTime()`
`D-KomponenteX`, `E-KomponenteX`	`PSBase::setTSComp()`, `PSBase::setTSCompComment()`
`D-Zeitreihentyp`, `E-Zeitreihentyp`	`PSBase::setTSType()`

Tabelle 4.2: Schlüssel, die intern benutzt werden

Schlüssel	Beschreibung des Schlüssels
`Binaer`	unterscheidet `TSBinary` und `TSNumeric`
`FALSE-Werte`, `TRUE-Werte`	zusätzliche Schlüssel von `TSBinary`
`Minimum`, `Maximum`, `Mittelwert`, `StdAbweichung`, `Varianz`	zusätzliche Schlüssel von `TSNumeric`
`D-Einheit`, `E-Einheit`, `Intervallbildung`	zusätzliche Schlüssel, unbenutzt im System (allerdings für Oberfläche wichtig !)
`UnskalierterTyp`	interner Schlüssel für externe Programme

4.4.1 Die Basisklasse: `PSBase`

Die Basisklasse `PSBase` stellt die Schnittstelle für alle Klassen zur Verfügung, die das Laden und Speichern der Zeitreihen implementieren. Außerdem sind in dieser Klasse einige Methoden vorhanden, um interne Daten der Zeitreihe zu ändern (siehe Tabelle 4.1).

Die vier Hauptmethoden:
Zwei Methoden sind verantwortlich für das Laden und Speichern der Zeitreihe: `load()` und `save()`. Beim Laden wird nur der Name der Zeitreihe angegeben, der abhängig vom persistenten Objekt gewählt werden darf. Im vorliegenden Dateiformat wäre als Name beispielsweise `Tegut|AAA|229104|Wochen.ABV` zulässig. Beim Speichern wird zusätzlich zum Namen noch ein Parameter übergeben, der einen Eintrag in einen Index von Zeitreihen erzwingen kann. Die Form der Indizierung ist wieder abhängig vom Datenformat. In unserem Fall entspricht die Indizierung einem Eintrag in die `INDEX`-Datei (siehe Abschnitt 2.1.5).
Mit `destroy()` kann eine bereits gespeicherte Zeitreihe wieder gelöscht werden, mit `exists()` kann man prüfen, ob eine Zeitreihe des übergebenen Namens existiert.

Projektanbindung:
Manche Objekte, z.B. die Prognoseobjekte, lagern weitere Dateien, wie z.B. Log-Dateien, im Dateisystem ab. Zu diesem Zweck kann jede persistente Speicherung ein Basisverzeichnis angeben, in dem dann weiter verzweigt werden kann. In der vorliegenden Struktur entspricht dies dem Verzeichnis `$PROJEKT_HOME`.

Verwaltung von Prognosen:
Ähnlich der Indizierung der Zeitreihen werden auch alle erzeugten Prognosen indiziert. Dies dient dazu, den Ursprung einer Prognosezeitreihe später wieder herauszufinden. Alle wichtigen Daten einer Prognose, wie die beteiligten Zeitreihen, Prognoseverfahren und Parameter, sind hier gespeichert. Die Methoden `getPrognose()` und `setPrognose()` stellen die Schnittstelle zu dieser Indizierung dar. Im benutzten Dateiformat entspricht dies einem Eintrag in die Datei `PROGNOSE.LST` (siehe Abschnitt 2.1.6).

Interne Daten des Zeitreihenkopfes ändern:
Beim Laden müssen Daten aus dem Kopf der Zeitreihe geändert werden, auf die keine außenstehende Klasse Zugriff hat. Dieser Schutz ist nötig, damit andere Klassen nicht durch direkten Zugriff Inkonsistenzen im Kopf erzeugen können. Aus diesem Grund stellt die Klasse `PSBase` Methoden zur Verfügung, die nur von Subklassen dieser Klasse benutzt werden können und die direkten Zugriff auf die geschützten Zeitreiheninformationen haben. Dies ist durch eine `friend`-Verbindung zwischen den Klassen `TSHead` und `PSBase` möglich (siehe Beispiel 4.1). Die Methoden sind mit `getTS*()` und `setTS*()` gekennzeichnet. Eine Liste aller `setTS*()` Methoden mit den zugehörigen Schlüsseln ist in Tabelle 4.1 zu sehen.

4.4.2 Speicherung im bestehenden Datenformat: `PSFile`

Die Klasse `PSFile` bildet das bestehende Datenformat so ab, daß alle existierenden Zeitreihen gelesen und geschrieben werden können. Beim Erzeugen eines Objekts der Klasse `PSFile` muß das Basisverzeichnis für die Projekte mit angegeben werden. Dies ist im bestehenden alten System in der Environment-Variablen `$PROJEKT_HOME` gespeichert.

`PSFile` überschreibt die Methoden zum Laden und Speichern, ebenso alle Methoden, die zur Indizierung der Zeitreihen benötigt werden. Kopf und Rumpf kommen in verschiedene Dateien, die Indizierung wird auf die `INDEX`-Dateien abgebildet.

4.4.3 Beispiel für ein weiteres Format: `PSPictex`

Als ein kleines Beispiel für die Erweiterungsfähigkeit bei der persistenten Speicherung ist eine weitere Klasse entstanden, die zwar keine Zeitreihen laden kann, beim Speichern allerdings das Format von PICTEX benutzt. Damit kann man leicht eine graphische Darstellung einer Zeitreihe in LATEX-Quellen einbinden.

Ein Beispiel für eine Ausgabe mit Hilfe der Klasse `PSPictex` war bereits in Abbildung 2.1 zu sehen.

4.5 Vorverarbeitungsobjekte

Vorverarbeitungen werden benötigt, um Umformungen oder Transformationen an den Zeitreihendaten vorzunehmen. Meist sind die unbearbeiteten Zeitreihen nicht direkt für eine Prognose zu gebrauchen, da ein Trend oder eine Saison auftritt oder die Daten nicht in einem für das Prognoseverfahren verständlichen Format vorliegen.

Alle Vorverarbeitungen müssen als Namenspräfix `MD` (für *MoDifier*) haben.

4.5.1 Die Basisklasse: `MDBase`

Die Basisklasse definiert die Schnittstelle für alle abgeleiteten Vorverarbeitungen. Es werden zwei Methoden definiert, die in dieser Klasse abstrakt bleiben:

`scale()`
: Ist die eigentliche Vorverarbeitung. Als Parameter werden die zu skalierende Zeitreihe, eine Liste weiterer eingehender Zeitreihen (oder `NULL`) und ein String mit den Parametern für die Skalierung übergeben.

`descale()`
: Ist die Rückrichtung der Vorverarbeitung. Im günstigsten Fall macht `descale()` die Vorverarbeitung komplett rückgangig. Es gibt aber auch Verfahren, die bei der Vorverarbeitung mit Verlusten arbeiten und daher nicht komplett rückskalierbar sind oder sogar Verfahren, die keine Rückrichtung besitzen. In diesem Fall muß die Methode zwar implementiert werden, kann aber einen Abbruch des Programms erzwingen oder eine Warnung ausgeben.

4.5.2 Abgeleitete Klassen

Im Rahmen der Projektarbeit entstanden eine Reihe von Vorverarbeitungen, die alle auch im entstandenen C++-System verfügbar sind.

Alle Vorverarbeitungen gehen nach demselben Prinzip vor: Zuerst werden aus der übergebenen Zeichenkette mit Hilfe der Klasse `StringParser` (siehe Abschnitt 7.1.3) die Parameter der Vorverarbeitung extrahiert. Danach wird die Vorverarbeitung durchgeführt und die zur Rückrichtung nötigen Parameter mit Hilfe von `MDBase::addPrefix()` in den Kopf der Zeitreihe geschrieben. Die Rückrichtung muß prinzipiell dasselbe tun, allerdings entfällt hier der Eintrag in den Zeitreihenkopf.

Im folgenden wird für jede Vorverarbeitung angegeben, welche Parameter die Vorverarbeitung und die Rückrichtung benötigt, und wie das Verfahren grundsätzlich implementiert wurde.

`MDAggregate`

Parameter der Vorverarbeitung: `<Zeiteinheit> <Faktor>`
Rechnet die Zeitreihe in die Zeiteinheit `<Zeiteinheit>` mit Faktor `<Faktor>` um. Dies wird durch einen Aufruf von `TSBase::changeTime()` durchgeführt.
Keine Rückrechnung vorhanden.

`MDAround0`

Parameter der Vorverarbeitung: `<Radius>`
Skaliert die Zeitreihe linear, der Nullpunkt bleibt erhalten. Dabei wird das Maximum des Betrages der Zeitreihenwerte auf `<Radius>` gesetzt, d.h., die vorverarbeiteten Werte liegen im Bereich [-`<Radius>`, `<Radius>`].
$x' = (x * Radius)/AbsMaximum$
Parameter der Rückrichtung: `<Radius> <AbsMaximum>`
Aus dem übergebenen Radius und dem Maximum des Betrages aller Zeitreihenwerte `<AbsMaximum>` können die Originalwerte wieder berechnet werden.
$x = (x' * AbsMaximum)/Radius$

`MDAverage`

Parameter der Vorverarbeitung: `<Werte>`
Berechnet den gleitenden Durchschnitt über `<Werte>` Zeitreihenwerte. Dabei wird nicht immer die Summe der letzten Werte neu berechnet, sondern nur die erste Summe berechnet und die letzten `<Werte>` Werte in einem Ringpuffer verwaltet. Bei jedem Schritt tritt nun ein Zeitreihenwert aus der Summe aus, ein weiterer tritt ein. Die entstandene Zeitreihe wird vorne um `<Werte>`-1 Werte kürzer.
Keine Rückrechnung vorhanden.

`MDBinary`

Parameter der Vorverarbeitung: `<False> <True> <Schranke>`
Bildet die `TRUE`- und `FALSE`-Werte einer binären Zeitreihe auf die Werte `<True>` und `<False>` einer numerischen Zeitreihe ab. Das Ergebnis der Vorverarbeitung ist immer eine Subklasse von `TSNumeric` (oder die Klasse `TSNumeric` selbst).
Parameter der Rückrichtung: `<False> <True> <Schranke> <Quellklasse>`
Bildet numerische Werte der Zeitreihe auf binäre Werte `TRUE` und `FALSE` ab. Dabei

werden alle Werte unterhalb des Wertes `<Schranke>` auf `FALSE` abgebildet. Alle anderen Werte bekommen den Wert `TRUE`. Da eine numerische Zeitreihe eingeht, aber eine binäre als Ergebnis erwartet wird, ist der Name der binären Zeitreihenklasse im Parameter `<Quellklasse>` angegeben. Somit wird immer die korrekte Klasse wiederhergestellt.

`MDCrop`

Parameter der Vorverarbeitung: `<Start> <Ende>`
Schneidet die Zeitreihe mit Hilfe der Methode `TSBase::changeInterval()` auf das angegebene Intervall zu. Dabei sind `<Start>` und `<Ende>` logische Datumsangaben.
Keine Parameterauswertung bei der Rückrichtung.
Die Rückrichtung gibt die Zeitreihe unverändert zurück.

`MDDays`

Parameter der Vorverarbeitung: `<TageProWoche> <Min> <Max>`
Ein Spezialfall der linearen Skalierung (siehe unten). Dabei werden die Zeitreihenwerte auf das Intervall [`<Min>`, `<Max>`] skaliert, wobei die Null auf den Wert `<Min>` und der Wert `TageProWoche` auf den Wert `<Max>` gesetzt wird. Dies wird angewandt, um eine Angabe von Tagen pro Woche in ein Intervall zu skalieren.
$x' = Min + (x/TageProWoche)$
Parameter der Rückrichtung: `<TageProWoche> <Min> <Max>`
Dir Rückrichtung entsteht einfach durch Umkehrung der Vorverarbeitungsvorschrift.
$x = (x' - Min) * TageProWoche$

`MDDiff`

Parameter der Vorverarbeitung: `<Tiefe> <Delta> <Modus>`
Berechnet die `<Tiefe>`-te Differenz der Zeitreihenwerte, dabei werden zur Differenz immer Werte mit dem Abstand `<Delta>` benutzt. Vor der Berechnung einer Differenz wird der Zustand einer Zeitreihe gesichert, der Name der gesicherten Zeitreihe wird an die Parameterliste angehängt. Der angegebene Parameter `<Modus>` ist nur bei der Rückrichtung von Bedeutung.
Parameter der Rückrichtung: `<Tiefe> <Delta> <Modus> <Zeitreihen>`
Macht die `<Tiefe>`-te Differenz mit Abstand `<Delta>` wieder rückgängig, dabei werden sämtliche gesicherte Zeitreihen aus `<Zeitreihen>` gebraucht. Der Parameter `<Modus>` steuert die Art der Rückskalierung. Der Modus `ONESTEP` ist genauer und benutzt zum Aufaddieren immer einen Originalwert aus einer gesicherten Zeitreihe. Der Modus `ADDITIVE` benutzt zum Aufaddieren dagegen einen zuvor rückskalierten Wert, der Fehler beim Rückrechnen des letzten Wertes geht also mit ein.

`MDExpbinary`

Parameter der Vorverarbeitung: `<Max> <Faktor>`
Rechnet die binären Werte `TRUE` und `FALSE` auf numerische Werte um. Dabei wird einem `TRUE`-Wert immer der Wert `<Max>` zugeordnet. Nach einem `TRUE`-Wert folgende `FALSE`-Werte nehmen nicht sofort einen Wert von 0 an, sondern der Wert fällt exponentiell mit dem Faktor `<Faktor>` und steigendem Abstand zum letzten `TRUE`-Wert.
Parameter der Rückrichtung: `<Max> <Faktor> <Quellklasse>`
Die Rückrichtung bildet alle Werte, die größer oder gleich dem Wert `<Max>` sind, auf einen `TRUE`-Wert ab, alles andere wird auf `FALSE` gesetzt. Da (wie bei `MDBinary`) eine numerische Zeitreihe eingeht, aber eine binäre als Ergebnis erwartet wird, ist der Name

der binären Zeitreihenklasse im Parameter `<Quellklasse>` angegeben. Somit wird immer die korrekte Klasse wiederhergestellt.

`MDExpo`

Parameter der Vorverarbeitung: `<Faktor>`

Wendet auf die Zeitreihe ein exponentielles Glätten mit dem Faktor `<Faktor>` an. Beim exponentiellen Glätten wird theoretisch eine unendliche Summe benutzt. In der Implementation wird bestimmt, ab welchem Exponenten der Faktor unterhalb der Maschinengenauigkeit liegt. Der gefundene Exponent begrenzt die Anzahl der Summanden. Damit nicht jedesmal die Koeffizienten neu berechnet werden müssen, werden diese einmalig berechnet und in einem Array abgelegt.

Keine Rückrechnung vorhanden.

`MDInteger`

Keine Parameter bei der Vorverarbeitung nötig.

Die Zeitreihe wird unverändert zurückgegeben.

Keine Parameter bei der Rückrichtung nötig.

Bei der Rückrichtung werden die Zeitreihenwerte auf ganze Zahlen gerundet.

`MDLinear`

Parameter der Vorverarbeitung: `<Min> <Max>`

Die Zeitreihenwerte werden linear auf das Intervall [`<Min>`, `<Max>`] skaliert. Zur Rückskalierung werden die Originalwerte von Minimum und Maximum gesichert.

$x' = Min + ((x - Minimum) * (Max - Min))/(Maximum - Minimum)$

Parameter der Rückrichtung: `<Min> <Max> <OrigMin> <OrigMax>`

Macht die lineare Skalierung rückgängig, dabei sind `<OrigMin>` und `<OrigMax>` die bei der Vorverarbeitung benutzten Werte für das Minimum und Maximum der ursprünglichen Zeitreihe.

$x = OrigMin + ((x' - Min) * (OrigMax - OrigMin))/(Max - Min)$

`MDMy_sigma`

Parameter der Vorverarbeitung: `<Faktor> [<VMin> <VMax>]`

Skaliert die Zeitreihe um den Mittelwert, dabei werden die Werte um den Faktor `<Faktor>`*Standardabweichung gestaucht. Wird ein Intervall [`<VMin>`, `<VMax>`] angegeben, so werden die Zeitreihenwerte so verschoben, daß der Mittelwert auf die Intervallmitte fällt, an den Intervallgrenzen werden die Werte abgeschnitten. Mit dieser Vorverarbeitung kann man Ausreißer eliminieren. Zur Rückskalierung werden die benutzten Werte des Mittelwertes und der Standardabweichung an die Parameterliste angehängt.

Parameter der Rückrichtung: `<Faktor> <VMin> <VMax> <Mwert> <StdAbw>`

Macht die Skalierung wieder rückgängig. Wurde bei der Vorverarbeitung auf ein Intervall [`<VMin>`, `<VMax>`] skaliert, so treten hier Verluste auf. Ohne Angabe eines Intervalls bei der Vorverarbeitung arbeitet das Verfahren verlustfrei, in diesem Fall ist für die Parameter `<VMin>` und `<VMax>` jeweils der Wert `"-"` eingetragen.

`MDNone`

Diese Vorverarbeitung ändert weder bei der Vorverarbeitung, noch bei der Rückrichtung die Zeitreihe. Diese Klasse kann als Grundgerüst für neue Vorverarbeitungen kopiert werden.

`MDShift`

Parameter der Vorverarbeitung: `<Einheiten>`
Verschiebt die Zeitreihe mit Hilfe der Methode `TSBase::shiftInterval()` um `<Einheiten>` Zeiteinheiten nach rechts.
Parameter der Rückrichtung: `<Einheiten>`
Verschiebt die Zeitreihe mit Hilfe der Methode `TSBase::shiftInterval()` um `<Einheiten>` Zeiteinheiten zurück nach links.

`MDTrend`

Parameter der Vorverarbeitung: `<Fallend> <Gleich> <Steigend>`
Betrachtet die Änderung der Zeitreihenwerte und setzt die Werte für fallende, steigende oder gleichbleibende Zeitreihenwerte.
Keine Rückrechnung vorhanden.

`MDUnsigned`

Keine Parameter bei der Vorverarbeitung nötig.
Die Zeitreihe wird unverändert zurückgegeben.
Keine Parameter bei der Rückrichtung nötig.
Bei der Rückrichtung werden negative Zeitreihenwerte auf 0 gesetzt, d.h., man erhält eine Zeitreihe mit nichtnegativen Werten.

4.6 Prognoseobjekte

Die Prognosen sind eine Hauptanwendung des gesamten Systems. Alle Vorverarbeitungen sind aus dem Grund entstanden, die eingehenden Zeitreihen für das Prognoseverfahren sinnvoll zur Verfügung zu stellen. Alle bisher verwendeten Verfahren (siehe Abschnitt 2.3) werden auch in C++ als Klasse implementiert.

Alle Prognosen müssen als Namenspräfix `PR` (für *PRediction*) haben.

4.6.1 Die Basisklasse: `PRBase`

Die Basisklasse stellt zwei Methoden zur Verfügung, die hier abstrakt bleiben:

`prognose()`

Führt das Prognose-Verfahren aus. Die Methode bekommt als Parameter die zu prognostizierende Zeitreihe, die einfließenden Zeitreihen (oder `NULL`) und einen String, der die Parameter für die Prognose enthält. Die Methode wendet das Verfahren an und liefert eine neue Prognose-Zeitreihe, die noch nicht rückskaliert ist.

`info()`

Dekodiert die Parameter, die an jede Zeitreihe gekoppelt sind. Dies kann die Anzahl der eingehenden Werte pro Prognosewert sein oder auch die Verfügbarkeit der Zeitreihe in der Zukunft. Diese Parameter sind abhängig vom Prognoseverfahren.

4.6.2 Prognosen mit Hilfe von Steuerdateien: `PRInfile`

Diese abstrakte Klasse ist Subklasse von `PSBase` und liefert eine einfache Schnittstelle für Prognosen, die durch eine ASCII-Datei gesteuert werden. In allen Prognoseverfahren gleiche Codeteile wurden in dieser Klasse bereits implementiert, verfahrensabhängige Teile können in den abgeleiteten Klassen mit Hilfe der hier definierten Methoden `prognose_*()` eingesetzt werden.

Als Beispiele für dateigesteuerte Prognosen sind die Klassen `PRBP`, `PRSNNS` und `PRGauss` vorhanden, die Implementationen benötigen zwischen 150 bis 180 Zeilen.

4.6.3 Ein Beispiel zur Erweiterung: `PRNaiv`

Als Beispiel, wie einfach ein neues Verfahren hinzugefügt werden kann, wird hier die naive Prognose angeführt, die immer den letzten Wert der zu prognostizierenden Zeitreihe vorhersagt.

Um die naive Prognose in das System einzubauen, muß man zuerst den vorgegebenen Rahmen für eine Prognoseklasse übernehmen (d.h., die Registrierung der Klasse und Programmcode zum Parsieren des Parametertextes). Danach kann man mit wenigen Code-Zeilen die naive Prognose implementieren. Insgesamt fallen etwa 60 Code-Zeilen an, von denen allerdings etwa 50 Zeilen Standardcode entspricht, der aus einer existierenden Prognoseklasse kopiert und angepaßt werden muß.

4.7 Zeiteinheitenobjekte

Die Zeitrechnung ist ein wichtiger Bestandteil der Zeitreihenverarbeitung, da hier alle Informationen über die zugrundeliegende Zeiteinheit enthalten sind. Eine Zeitreihe muß beispielsweise genaue Angaben über ihren Start- und Endzeitpunkt machen können.

Wie speichert man am flexibelsten einen Zeitpunkt? Man kann zur Beschreibung eines Tages das Datum speichern, bei feineren Einheiten muß ebenfalls die Uhrzeit angegeben werden. Bei einer wochenweisen Berechnung reicht bereits die Angabe der Kalenderwoche und des Jahres aus. Um flexibel zu bleiben, und um besser zwischen verschiedenen Zeiteinheiten umrechnen zu können, sollte man aber alle möglichen Werte abfragen können. Um unnötige doppelte Berechnungen zu sparen, sollte das Objekt alle abfragbaren Werte einer Zeiteinheit intern speichern: Uhrzeit, Datum, Wochentag, Kalenderwoche und das Quartal. Werden bei der Angabe eines Zeitpunktes nicht alle Angaben gemacht, so sollte das Objekt alle anderen Werte selbst berechnen, wie im nächsten Beispiel gezeigt:

Beispiel 4.5:
Wird ein Objekt mit einem Datum aufgerufen, so müssen danach noch Wochentag, Kalenderwoche und Quartal berechnet werden. Wird das Objekt allerdings mit einer Kalenderwoche gesetzt, so wird das Datum des Montags der Woche berechnet, der Wochentag auf Montag gesetzt und ebenfalls das Quartal berechnet.

Zusätzlich wird das Datum mit Hilfe eines logischen Ticks gesichert, der die Anzahl der Zeiteinheiten ab einem einmalig festgelegten Startdatum angibt (aktuell wird als Startdatum der 1.1.1945, 00:00:00 Uhr, benutzt). Zusätzlich wird ein Faktor angegeben, der angibt, wieviele Zeiteinheiten als ein Tick gerechnet werden, z.B. um einen 2-Wochen Rhythmus darzustellen. Auch hier werden Umrechnungen bei der Initialisierung vorgenommen, um mehrfache Berechnungen zu vermeiden.

Beispiel 4.6:
Wird ein Tage-Objekt mit einem Datum 5.1.1945 und Faktor 1 initialisiert, so wird als logischer Tick der Wert 5 berechnet. Initialisiert man das Objekt mit demselben Datum, allerdings mit Faktor 2, so wird intern immer noch als Tick 5 berechnet, bei einer Abfrage wird allerdings der korrekte Tick 3 geliefert ($3 = \frac{5}{2}$ aufgerundet).
Wird ein Tage-Objekt mit dem logischen Tick 6 und dem Faktor 1 initialisiert, so wird als Datum der 6.1.1945 berechnet. Initialisiert man mit Tick 3 und Faktor 2, so erhält man als Datum den 5.1.1945, da dies der erste Tag eines 2-Tage Rhythmus ist (1./2., 3./4., 5./6. Januar 1945).

Bei der Umrechnung in eine gröbere Einheit (z.B. Tage nach Wochen), geht Information verloren. Wird sofort danach zurückgerechnet, erhält man unter Umständen ein anderes Datum als das Ursprungsdatum. Um die Zeitrechnung in dieser Hinsicht verlustfrei zu halten, wird ein Datum immer doppelt gespeichert. Bei der Initialisierung wird ein Originaldatum berechnet, welches im folgenden nicht mehr geändert wird. Aus dem Originaldatum wird ein zulässiges Datum für diese Zeiteinheit berechnet. Wird nun in eine andere Zeiteinheit umgewandelt, so wird zu diesem Zweck das Originaldatum ausgelesen. Da bei einer Rückrechnung dasselbe Originaldatum benutzt wird, tritt hier kein Verlust mehr auf.

Das Originaldatum kann durch einen Aufruf von `setAsOriginal()` auf das zur aktuellen Zeiteinheit passende Datum umgestellt werden, falls dies gewünscht ist. Dadurch ist dieselbe Funktionalität wie ohne Originaldatum gegeben, Verluste können bei der Rückrechnung wieder auftreten.

Beispiel 4.7:
Abbildung 4.3 zeigt eine Umrechnung von Tagen nach Wochen und zurück. Die Zeiteinheit Wochen setzt das berechnete Datum auf den Montag der Woche. Die Umrechnung zurück nach Tagen ist mit und ohne das Überschreiben des Originaldatums dargestellt. In der oberen Zeile ist kein Verlust, da das Originaldatum wieder benutzt wurde. In der zweiten Zeile wird das Wochen-Objekt auf das berechnete Datum gesetzt (mit `setAsOriginal()`, s.u.), es tritt ein Verlust beim Zurückrechnen auf. Dieser Effekt kann aber benutzt werden, um von einem Tag auf den Anfang der Woche zu springen.

Fortsetzung ▷

▷ Fortsetzung **Beispiel 4.7**

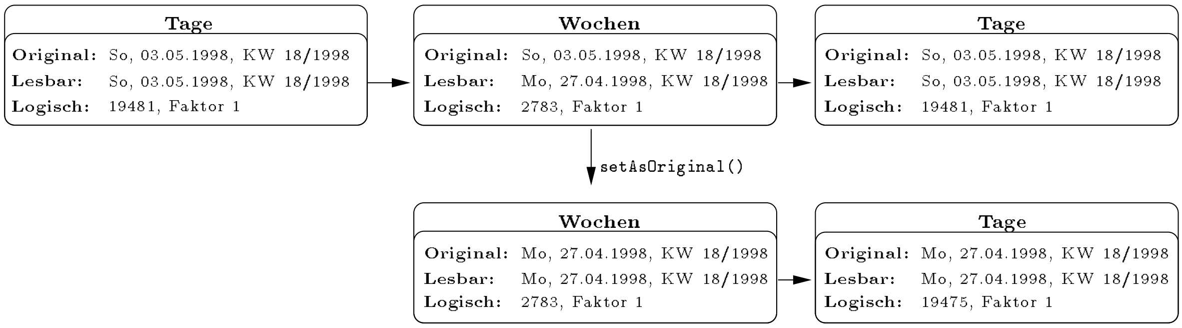

Abbildung 4.3: Beispiel: Zeit-Umrechnung Tage-Wochen-Tage

Weiterhin wird hier auch die Aggregation von Werten von und zu der Zeiteinheit implementiert, beispielsweise kann man eine tageweise Zeitreihe damit auf Wochenbasis aggregieren.

Alle Objekte für Zeiteinheiten müssen als Namenspräfix `TM` (für *TiMe*) haben.

4.7.1 Die Basisklasse: `TMBase`

Die Basisklasse implementiert alle grundlegenden Methoden zur Zeitrechnung. Weitergehende Umrechnungen müssen in den abgeleiteten Klassen zur Verfügung gestellt werden.

Konstruktoren:
Es existieren Konstruktoren, die ein Zeiteinheitenobjekt anhand eines übergebenen Datums setzen, aus einem Formatstring lesen oder aus einem anderen Zeiteinheitenobjekt kopieren. Wird das Zeiteinheitenobjekt aus einem anderen Zeiteinheitenobjekt initialisiert, findet bei unterschiedlichen Einheiten eine Umrechnung statt (z.B. wenn ein Tage-Objekt mit einem Jahres-Objekt initialisiert wird).

Datumstypen von Zeiteinheiten:
Beim Erzeugen von Zeiteinheitenobjekten kennt `TMBase` die folgenden Datumstypen: `DATUM`, `KWOCHE`, `MONAT`, `QUARTAL`, `TICKS` und `TIME`. Die Typen beschreiben, welche Parameter im Konstruktor übergeben werden müssen. Eine genaue Beschreibung der Typen folgt in Tabelle 4.3.

Abfragetypen von Zeiteinheiten:
Man kann jeden einzelnen Wert des Zeiteinheitenobjekts abfragen. Dafür stehen die folgenden Abfragetypen bereit: `QUERY_SEKUNDE`, `QUERY_MINUTE`, `QUERY_STUNDE`, `QUERY_TAG`, `QUERY_MONAT`, `QUERY_QUARTAL`, `QUERY_JAHR`, `QUERY_KWOCHE`, `QUERY_KJAHR` und `QUERY_WTAG`. Eine genaue Beschreibung der Abfragetypen folgt in Tabelle 4.4.

Methoden zum Bearbeiten der Zeiteinheit:
Man kann ein bestehendes Zeiteinheitenobjekt auf ein anderes Datum setzen. Dabei kann man ein logisches Datum (`setTick()`) oder einen Formatstring (`parseTime()`) übergeben. Alle Werte des Objekts können auch wieder ausgelesen werden: Das logische Datum bekommt man über `getTick()` und `getFactor()`, einen lesbaren String über `getReadable()`, und einzelne Werte des Datums, wie Tag oder Kalenderwoche, über die Methode `getValue()`.

Methoden zur Neuberechnung:
Nach dem Setzen des realen oder logischen Datums muß das jeweils andere Datum neu berechnet werden. `TMBase` stellt hierfür die nötigen Methoden `recalc_logical()` und `recalc_real()` zur Verfügung, die hier allerdings keine Funktionalität haben und nur als Schnittstelle definiert werden. Die abgeleiteten Klassen müssen hier die Umrechnungen implementieren. Die beiden Methoden sind nicht öffentlich, sondern können nur in Subklassen von `TMBase` aufgerufen werden.

Methoden zum Prüfen auf Gültigkeit:
Es existieren für alle Typen Methoden, die ein Datum auf Gültigkeit überprüfen: `gueltig_datum()` für Kalendertage (inklusive Uhrzeit), `gueltig_kwoche()` für Kalenderwochen und `gueltig_quartal()` für Quartalsangaben.

Aggregation auf andere Zeiteinheiten:
Die Aggregation ist ein aufwendigerer Schritt als die Umrechnung. Umrechnung bezeichnet nur das Umrechnen *eines* Datums in eine andere Zeiteinheit. Aggregation bedeutet aber, daß eine komplette Zeitreihe auf eine andere Zeiteinheit umgerechnet werden muß. Dabei werden mehrere Zeitreihenwerte zu einem einzigen Wert zusammengefaßt. Die Methoden `aggregateTo()` und `aggregateFrom()` implementieren eine Umrechnung zu oder von der zugehörigen Zeiteinheit. Ist die Aggregation nicht verfügbar, liefert die Methode `NULL`.

Sonstige Methoden:
Man kann den Namen der Zeiteinheit in mehreren Sprachen abfragen (`getDesc()`). Weiterhin kann man das Originaldatum auf den Wert des umgerechneten Datums setzen (`setAsOriginal()`, siehe Beispiel 4.7).

Verschiedene Datumstypen bei der Initialisierung/Ausgabe

Die Datumstypen geben bei der Initialisierung durch einen Konstruktor oder ein Objekt vom Typ `TimeInit` an, welche Parameter folgen. Eine Liste aller Datumstypen ist in Tabelle 4.3 zu sehen.

Tabelle 4.3: Datumstypen und zugehörige Parameter

Typ	Format	Parameter
`DATUM`	`TT.MM.JJJJ HH:MM:SS`	Faktor, Jahr, Monat, Tag, Stunde, Minute, Sekunde
`MONAT`	`MM \ JJJJ` `MM / JJJJ`	Faktor, Jahr, Monat, Tag, Stunde, Minute, Sekunde
`KWOCHE`	`MM \ JJJJ` `MM / JJJJ`	Faktor, Kalenderjahr, Kalenderwoche
`QUARTAL`	`QRT \ JJJJ` `QRT / JJJJ`	Faktor, Jahr, Quartal
`TICKS`	`TICK`	Faktor, Tick (>= 0)
`TIME`	`--`	nur intern benutzt bei Initialisierung durch `TimeInit`

Verschiedene Abfragetypen

Die Abfragetypen geben an, welchen Wert des Datums man abfragen will. Die Typen werden als Parameter an die Methode `getValue()` übergeben. Eine Liste aller Abfragetypen ist in Tabelle 4.4 zu sehen.

Tabelle 4.4: Liste aller Abfragetypen für Zeiteinheiten

Typ	zugehöriger Teil des Datums (Wertebereich)
`QUERY_SEKUNDE`	Sekunde (aus 0..59)
`QUERY_MINUTE`	Minute (aus 0..59)
`QUERY_STUNDE`	Stunde (aus 0..23)
`QUERY_TAG`	Tag im Monat (aus 1..31)
`QUERY_MONAT`	Monat im Jahr (aus 1..12)
`QUERY_QUARTAL`	Quartal im Jahr (aus 1..4)
`QUERY_JAHR`	Jahreszahl (>= 1945)
`QUERY_KWOCHE`	Kalenderwoche im Jahr (1..53)
`QUERY_KJAHR`	Kalenderjahr (>= 1945)
`QUERY_WTAG`	Wochentag (0..6, 0=Montag)

Initialisierungsobjekt für `TMBase`: `TimeInit`

Die Klasse `TimeInit` ist wie alle Initialisierungsobjekte eine Subklasse von `ClassInit` und nimmt Informationen über eine Zeiteinheit auf. Es können Zeiteinheitstyp mit zugehörigen Werten (vergleiche Tabelle 4.3) oder eine andere Zeiteinheit vom Typ `TMBase*` angegeben werden.

Beispiel 4.8:
Hier folgen einige Beispiele, auf welche Art man `TimeInit` erzeugen kann. Das Datum wird bei einem Aufruf von `anew()` an das Zeiteinheitenobjekt weitergegeben und zur Initialisierung benutzt.

`TimeInit( Time::DATUM, 1, 1998, 4, 1, 15, 34, 00 )`
  1.April.1998 15:34:00 Uhr (Faktor 1)
`TimeInit( Time::KWOCHE, 5, 1998, 36 )`
  KW 36/1998 (Faktor 5)
`TimeInit( Time::TICKS, 3, 105 )`
  Tick 105 (Faktor 3)
`TimeInit( thisdate, 2 )`
  Datum aus `thisdate`, auf Faktor 2 umgerechnet

4.7.2 Die abgeleiteten Klassen

Es gibt für alle gängigen Zeiteinheiten abgeleitete Klassen: `TMSekunden`, `TMMinuten`, `TMStunden`, `TMTage`, `TMWochen`, `TMMonate`, `TMQuartale` und `TMJahre` sowie die Klasse `TMTicks`, die nur logische Ticks verwaltet.

Die Umrechnungen sind dabei für alle Klassen bereits fertig, die Aggregationen sind allerdings nur für die Einheiten Tage, Monate und Quartale exemplarisch implementiert.

4.8 Projektobjekte

Das Projekt bestimmt die Beschreibung einer Zeitreihe sowie die Namen aller Typen und Komponenten, die in diesem Projekt auftreten. Weiterhin werden hier alle nötigen Daten für eine Indizierung der Zeitreihen zur Verfügung gestellt.

Alle Projektobjekte müssen als Namenspräfix `PJ` (für *ProJect*) haben.

4.8.1 Die Basisklasse: `PJBase`

Die Klasse `PJBase` kann anhand des Typs die Beschreibung und den Namen der Zeitreihe oder das Suffix des Zeitreihennamens bestimmen. Die Rückrichtung ist ebenfalls möglich: Die Klasse kann ebenfalls anhand des Namens oder Suffixes den Typ der Zeitreihe bestimmen. Ein Zeitreihentyp wird als Integer geliefert, die Verbindung zum Namen ist nur innerhalb des Projektobjekts definiert.

Die Suchmethoden sind bereits in `PJBase` implementiert, allerdings muß eine abgeleitete Klasse mehrere interne `get_*()` Methoden überschreiben.

Mit dem Typ kann man Name, Suffix und Beschreibung suchen lassen (`getTypeName()`, `getTypeToken()` und `getTypeDesc()`). Die Suche nach einem Typ kann anhand des Namens und der Beschreibung erfolgen (`getTypeFromName()` und `getTypeFromDesc()`).

Die Beschreibungen für die verschiedenen Komponenten einer Zeitreihe (d.h. für das Projekt selbst und u.U. weitere Subprojekte) kann man mit der Methode `getKompDesc()` ermitteln.

Beispiel 4.9:
In der Tabelle 4.5 sind alle für die Zeitreihe `Tegut|AAA|229104|Wochen.ABV` relevanten Informationen des Projektes angegeben.

Tabelle 4.5: Informationen, die vom Projekt verwaltet werden

Art der Information	zugehörige Methode	Beispielwert
Komponenten		`Tegut|AAA|229104|Wochen.ABV`
Typ	`getTypeFromDesc()`, `getTypeFromName()`	1
Name der Zeitreihe	`getTypeName()`	Abverkauf — sale
Beschreibung der Zeitreihe	`getTypeDesc()`	010 KOELLNFL.BLUE.ZART500 HAFERFLOCKEN
Suffix der Zeitreihe	`getTypeToken()`	ABV
Name der Komponente	`getKompDesc()`	Projekt, Filiale, Artikelnr.

Die Indizierung wird über die Methoden `getIndexHeading()` (für die Überschrift eines Index) `getIndexEntry()` (für die Beschreibung einer Zeitreihe im Index) und `getIndexType()` (für den Typ eines Eintrages im Index) gesteuert. Der Typ eines Eintrages bestimmt, ob es sich um eine Zeitreihe oder ein weiteres Subprojekt handelt. Ebenso unterscheidet der Typ zwischen globalen und Projekt-Zeitreihen. Alle Methoden zur Indizierung sind in der Basisklasse abstrakt und müssen in Subprojekten implementiert werden.

Für die projektspezifische Ablage von Daten, wie zum Beispiel Logfiles, kann das Projektobjekt mit Hilfe der Methode `gotoProjectDir()` auch in ein eigenes Verzeichnis wechseln (und dies bei Bedarf auch anlegen).

4.8.2 Beispiel eines Projektes: `PJTegut`

Es sind im Laufe der Diplomarbeit verschiedene Projekte bearbeitet worden, von denen das Tegut und das CO_2-Projekt exemplarisch implementiert wurden. Ebenfalls wurden die globalen Zeitreihen in ein Projektobjekt gefaßt. Die Methodik wird hier am Beispiel des Tegut-Projektes verdeutlicht.

Das Tegut-Projekt besteht aus 7 verschiedenen Typen, die in Tabelle 4.6 aufgelistet sind.

Tabelle 4.6: Tegut-Typen

Typ	Suffix	dt. Beschreibung	engl Beschreibung
0	`ABV`	Abverkauf	sale
1	`AKT`	Aktionen	advertisement
2	`AUM`	Artikelumsatz	article turnover
3	`KND`	Kunden	customers
4	`PRS`	Preis	price
5	`RUS`	Regalumsatz	shelf turnover
6	`UMS`	Gesamtumsatz	total turnover

Es existieren für jede Zeitreihe 3 Komponenten, die in Tabelle 4.7 beschrieben sind.

Tabelle 4.7: Tegut-Komponenten

Komponente	dt. Beschreibung	engl. Beschreibung
1	Projekt	project
2	Filiale	branch
3	Artikelnr.	artickeno.

Die Methode `get_token()` liefert zu einem Typ das Suffix, die Methode `get_name()` liefert den Namen des Typs. Die Methode `get_desc()` liefert die Beschreibung eines Artikels, die in einer Textdatei `ARTIKEL.TXT` steht (die Zuordnung geschieht anhand der Artikelnummer). `get_komp()` liefert die Beschreibung einer Komponente in einer oder allen Sprachen.

Die Indizierungs-Methoden liefern anhand der benutzten Komponenten verschiedene Ausgaben. `getIndexHeading()` gibt fest eingebaute Texte zurück (z.B. `"Projekt"` oder `"Artikel"`), `getIndexEntry()` liefert den Typ und die Zeiteinheit einer Zeitreihe als Ergebnis.

4.8.3 Verwaltung mehrerer Projekte: `ProjectManager`

Eine Zeitreihe ist immer mit einem Projekt verbunden. Der Name des Projektes ist gleichbedeutend mit dem Namen der ersten Komponente. Die Verwaltung mehrerer Projekte wird durch die Klasse `ProjectManager` vereinfacht.

Es kann ein primäres Projekt definiert werden. Das primäre Projekt ist bei Prognosen das Projekt der zu prognostizierenden Zeitreihe. Weiterhin wird eine Liste von sekundären Projekten verwaltet. Diese Liste wird automatisch beim Laden und Erzeugen von neuen Zeitreihen aktualisiert. Die dafür nötigen Methodenaufrufe sind in den Klassen `TSHead` und `TSBase` verankert.

Es existieren Methoden zum Setzen und Erweitern der Projektlisten (`setPrimary()` und `addSecondary()`) sowie zum Abfragen und Suchen innerhalb der Listen (`getPrimary()`, `getFirstSecondary()` und `getNextSecondary()` für die Suche von Hand, `getProjectForName()` für die automatische Suche). Bei der automatischen Suche wird das Projekt neu angelegt und in die Liste der sekundären Projekte aufgenommen, falls es nicht gefunden wurde.

Eine Methode, um in das projektspezifische Verzeichnis zu wechseln, wird mit `gotoProjectDir()` angeboten.

Der Projekt-Manager ist als Objektvariable in der Klasse `TSHead` enthalten und wird beim Start des Programms automatisch angelegt. Falls ein direkter Zugriff auf den Manager benötigt wird, kann dieser über `TSHead::getProjectManager()` referenziert werden.

4.9 Verwaltungsaufgaben: `Administration`

Die Klasse `Administration` stellt Methoden zur Verfügung, die den Umgang mit der Klassenhierarchie einfacher machen. Das Setzen und Ändern des persistenten Objekts wird hier genauso zur Verfügung gestellt wie das Skalieren und Prognostizieren einer Zeitreihe. Zusätzlich können Zeitreihen angegeben werden, die beim Programmende gelöscht werden müssen. Da alle Methoden der Klasse statisch sind, ist keine Instanz der Klasse nötig: Die Methoden können alle über `Administration::XXX()` aufgerufen werden.

4.10 Kommunikation zwischen den Objekten

Da die Aufgaben einer Zeitreihe in verschiedenen Klassen abgebildet wurde, müssen die Objekte bei verschiedenen Arbeitsschritten miteinander kommunizieren. Die Schnittstellen, die durch die Basisklassen gegeben sind, bieten hierfür alle nötigen Methoden an.

4.10.1 Aggregation

Die Aggregation ist eine sehr kommunikationsintensive Aufgabe. Da das Aggregieren einer Zeitreihe eine Vorverarbeitung ist, existiert ein Objekt `MDAggregate`, das mit einem String als Parameter aufgerufen wird. Dieser enthält die Zielzeiteinheit und den gewünschten Faktor. `Wochen 2` wandelt beispielsweise in eine Zeitreihe mit 2-wöchigen Werten um.

`MDAggregate` fragt nun das Zeitreihenobjekt (eine Subklasse von `TSHead`), in welcher Zeiteinheit es gerade vorliegt. Danach erzeugt es anhand der Namen der Quell- und Zielzeiteinheit die beiden zugehörigen Zeiteinheitenobjekte. Diese werden nun gefragt, ob sie die benötigte Aggregation implementiert haben. Findet sich in beiden Zeiteinheitenobjekten keine solche Aggregation, ist die Vorverarbeitung erfolglos und das Programm bricht ab.

Ist jedoch ein Zeiteinheitenobjekt in der Lage die Werte zu aggregieren, so geht die Kommunikation weiter: Das Zeiteinheitenobjekt bestimmt jetzt immer wieder eine Menge von Werten, die aggregiert werden müssen. Diese Werte werden wieder dem Zeitreihenobjekt übermittelt, da nur dieses weiß wie mehrere Werte aggregiert werden (Mittelwert, Summe, etc.). Das Zeitreihenobjekt liefert einen einzelnen Wert als Ergebnis. Dieser wird vom Zeiteinheitenobjekt übernommen und in die Zeitreihe geschrieben.

Beispiel 4.10:
Eine Zeitreihe ist tageweise im Bereich vom 1.4.1998 bis zum 15.4.1998 definiert, die Zeitreihe hat also 22 Werte. Diese Zeitreihe soll jetzt auf Wochenbasis umgerechnet werden. Ein Aufruf von `MDAggregate` mit dem Parameter `"Wochen 1"` läuft wie folgt ab:

Es wird versucht, eine Instanz der Klasse `TMWochen` zu erzeugen. Danach werden die neuen Start- und Endzeitpunkte berechnet, dabei entspricht der 1.4.1998 der Kalenderwoche 14/1998, der 15.4.1998 liegt in der Kalenderwoche 17/1998.

Danach wird zuerst geprüft, ob die Klasse `TMTage` die Umrechnung auf Wochen mit Hilfe der Methode `aggregateTo()` beherrscht. Dies ist hier der Fall (Andernfalls wird in der Klasse `TMWochen` auf eine Umrechnung mit Hilfe von `aggregateFrom()` getestet).

Die Methode `aggregateTo()` bekommt jetzt die Zeitreihenwerte, die Startzeitpunkte (Tag und Woche) als Zeiteinheitenobjekt und den Endzeitpunkt (ebenfalls Tag und Woche) als logischen Tick übergeben. Jetzt werden immer die Tage einer Woche zusammengefaßt und durch die Methode `aggregate()` der Klasse `TSBase` zu einem Wert aggregiert (siehe Abbildung 4.4). Als Ergebnis liefert die Methode ein Array mit den aggregierten Werten zurück.

Diese Werte werden, zusammen mit Start- und Endzeitpunkt in Wochen, in eine neue Wochen-Zeitreihe eingetragen und als Ergebnis der Vorverarbeitung zurückgegeben. Das alte Zeitreihenobjekt wird gelöscht.

Fortsetzung ▷

▷ Fortsetzung **Beispiel 4.10**

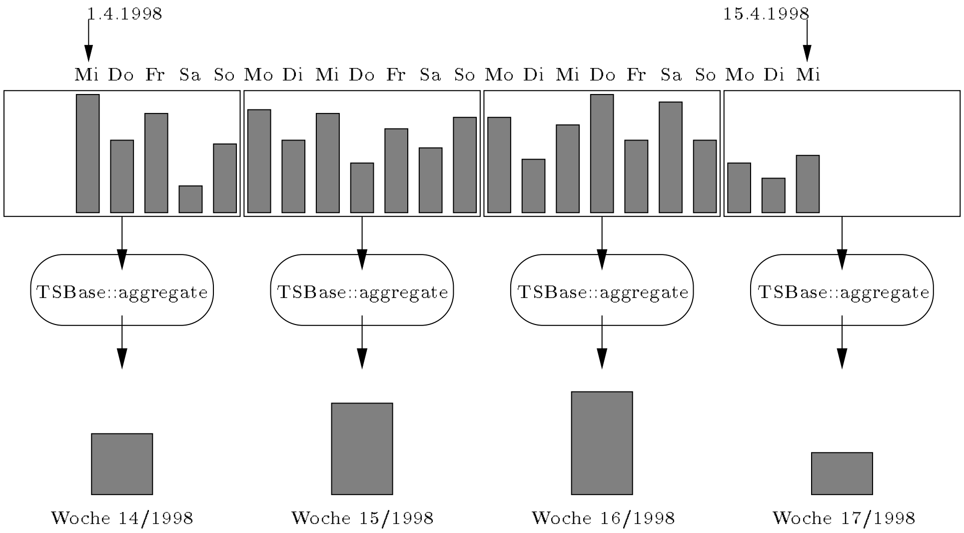

Abbildung 4.4: Beispiel: Aggregation von Tagen auf Wochen

Kapitel 5

Anwendung der Klassenbibliothek

In diesem Kapitel wird gezeigt, wie man die entwickelte Klassenbibliothek anwenden kann. Dabei werden sowohl ein Programm zur Steuerung von Prognosen, als auch andere kleine Hilfsprogramme angeführt.

Mit Hilfe der Beispiele kann man gut die Funktionalität der Klassen erkennen, die Anwendung in eigenen Programmen sollte danach keine Probleme bereiten.

5.1 Erzeugen von neuen Zeitreihen

Das erste Problem im Umgang mit Zeitreihen ist, diese aus bekannten Daten zu erzeugen. Weiterhin sollen ausreichend Informationen in den Kopf eingetragen werden, so daß die erzeugte Zeitreihe auch in der Oberfläche angezeigt werden kann.

5.1.1 Beispiel: Sinus-Zeitreihe

Als Beispiel wird eine Sinus-Schwingung erzeugt. Die Zeitreihe hat Ticks als Zeiteinheit (ein Tick entspricht einem Winkelgrad) und ist von 1 bis 360 definiert (d.h., es entsteht eine komplette Periode). Der Quellcode steht in der Datei `programs/Create.cpp`.

Der grobe Ablauf beim Erzeugen einer Zeitreihe:

1. Erzeugen der `Administration`-Klasse mit zugehörigem persistenten Objekt (hier `PSFile`) und Projekt (hier `Global`).
2. Erzeugen eines leeren Zeitreihenobjekts der richtigen Klasse (hier `TSNumeric`) im gewünschten Definitionsbereich (hier 1 bis 360).
3. Komponenten der Zeitreihe setzen (hier `Global` und `JahresSaison`), dabei können die Komponenten einzeln und mit Kommentar angegeben werden oder alle am Stück (dabei werden die Kommentare aus dem Projekt-Objekt abgefragt).
4. Zeiteinheit angeben (hier `Ticks`).
5. Zeitreihentyp angeben (hier `Jahressaison`).
6. Weitere Informationen für die Oberfläche angeben (Intervallbildung, Werteeinheit und Beschreibung).
7. Alle Werte in Definitionsbereich berechnen und eintragen.
8. Zeitreihe speichern und dabei in den `INDEX` eintragen lassen.
9. Objekte löschen und `cleanup`-Methode aller Klassen aufrufen (über `Class::cleanupAllClasses()`).

Alle Schritte sind in diesem Beispiel mit unter 100 Codezeilen implementiert worden. Der Aufruf des Programms erfordert keine Argumente; es wird genau eine Zeitreihe erzeugt und in den `INDEX` eingetragen.

```
// Initialisierungs-Objekt fuer PSFile, uebergibt $PROJEKT_HOME
PSFileInit *init = NEW(init, PSFileInit, (getenv("PROJEKT_HOME")));

// Administrations-Klasse: "PSFile" eintragen und "Global" als Projekt
Administration *admin = NEW(admin, Administration, ( (PSFile*)PSFile::anew(init),
    "Global" ));

// leere numerische ZR erzeugen, Def.-Bereich [1,360], Standardwert 0.0
TSNumeric zr(1, 360, 0.0);

// Komponenten setzen, dabei muessen die Kommentare vom Projekt-Objekt
// eingetragen werden.
zr.setComponents("Global|JahresSaison|Ticks.JSS");

// Zeiteinheit der Zeitreihe setzen
zr.setTime("Ticks");

// Typ der Zeitreihe setzen
zr.setType("Jahressaison");

// fuer Oberflaeche wichtige Eintraege (Intervallbildung, Einheit & Beschreibung)
zr.setInfo("Intervallbildung", 0L, NULL);
zr.setInfo("D-Einheit", "Anzahl", NULL);
zr.setInfo("E-Einheit", "times", NULL);
zr.setInfo("D-Beschreibung", "Jahressaison in Ticks", NULL);
zr.setInfo("E-Beschreibung", "yearly season in ticks", NULL);

// Werte fuer Winkel 1..360 berechnen und eintragen
for(int grad=1 ; grad<=360 ; ++grad)
    zr.setValue(grad, sin( (grad*3.1415)/180.) );

// Zeitreihe speichern, mit Eintrag in den INDEX
zr.save(NULL, 1);

// Aufraeumen nicht vergessen
if(admin) { DELETE(admin); admin=NULL; }
Class::cleanupAllClasses();
```

Datei `programs/Create.cpp`: Erzeugen einer Sinus-Zeitreihe (nur `main()`-Methode)

5.2 Prognosen mit ASCII-Parameterdateien

Mit dem Programm Training kann man einen kompletten Prognoselauf starten. Die Parameter werden dabei alle in einer Steuerdatei angegeben. Jeder Prognoselauf erhält zusätzlich einen Kommentar, der später benutzt werden kann, um verschiedene Läufe zu unterscheiden. Beim Aufruf des Programms muß man die Steuerdatei, den Kommentar und mindestens das Projekt angeben. Optional können noch mehr Komponenten übergeben werden:

```
Training Steuerdatei Kommentar Projekt [Komponente2 Komponente3 ...]
```

Im folgenden Beispiel ist der Auszug einer ASCII-Steuerdatei zu sehen:

```
# Trainings-Datei fuer C++-Applikation
INFILE  BP.infile

## BP-Prognose (mit BP.infile), Kommentar wird vom Programm angehaengt
TOOL    BP(BP.infile,BOTH)

## Prognose in Wochen, Umrechnung erfolgt automatisch
MEASURE Wochen

## Auf Intervall von 36/1994 bis 51/1995 zuschneiden
DEFINED 36/1994 51/1995

## Prognose-Wert wird 2 Wochen in die Zukunft vorhergesagt
ADVANCE 2

# die zu prognostizierende Zeitreihe
OUTPUT  $KOMP1|$KOMP2|$KOMP3|Wochen.ABV()                  \
            UNSIGNED | INTEGER | LINEAR 0.1 0.9

# die eingehenden Zeitreihen
INPUT   PRESENCE $KOMP1|$KOMP2|$KOMP3|Wochen.ABV(5)     \
            UNSIGNED | INTEGER | LINEAR 0.1 0.9

INPUT   FUTURE   $KOMP1|$KOMP2|$KOMP3|Tage.AKT(1)       \
            BINARY 0.1 0.9 0.5

INPUT   FUTURE   Global|Feiertage|Tage.Hessen.FTG(3)    \
            BINARY 0.1 0.9 0.5
```

```
# die folgende PARAM-Zeilen werden einfach in das Infile kopiert
PARAM   EPOCHEN     1000
PARAM   MINEPOCHEN  200
PARAM   ERR         40
     (hier folgen weitere Parameter)
```

Datei `BP.Infile`: ASCII-Steuerdatei für BP-Tool

Das Programm parsiert die Steuerdatei und kennt dabei die in Tabelle 5.1 aufgelisteten Befehle. Aus der übergebenen Steuerdatei wird eine zweite erzeugt, die an externe Programme weitergegeben wird.

Tabelle 5.1: Zulässige Befehle einer ASCII-Steuerdatei

Befehl	Parameter
`INFILE`	Dateiname
`TOOL`	Toolname(Parameter)
`DEFINED`	Von Bis
`OUTPUT`	Zeitreihe(Parameter) VPipeline
`INPUT`	`PRESENCE|FUTURE` Zeitreihe(Parameter) VPipeline
`MEASURE`	Zeiteinheit [Faktor]
`ADVANCE`	AnzahlTicks
`PARAM`	beliebig
`#`	beliebig
`##`	beliebig

Im folgenden wird eine Beschreibung der Befehle anhand von Beispielen angegeben:

`INFILE BP.infile`
: Die zweite Steuerdatei wird in `BP.infile` gespeichert.

`TOOL BP(BP.infile,BOTH)`
: Die Prognose verwendet das BackPropagation-Tool von Ulrich Middelberg, dabei wird als Steuerfile `BP.infile` benutzt und es werden Trainings- und Testzeitreihen erzeugt (da der Modus auf `BOTH` steht)

`MEASURE Wochen`
: Die beteiligten Zeitreihen sollen alle in Wochen benutzt werden. Zeitreihen in anderen Zeiteinheiten werden nach Wochen aggregiert.

`DEFINED 36/1994 51/1995`
: Die eingehenden Zeitreihen sind (maximal) im Bereich von der 36. Woche 1994 bis zur 51. Woche 1995 (einschließlich) definiert.

`ADVANCE 2`
: Der erste Prognosewert wird zwei Wochen nach dem letzten bekannten Wert angeschlagen (d.h., es entsteht eine Lücke von einer Woche)

`OUTPUT Tegut|AAA|229104|Wochen.ABV() LINEAR 0.1 0.9`
: Die Zeitreihe `Tegut|AAA|229104|Wochen.ABV` wird linear vorverarbeitet (`LINEAR 0.1 0.9`) und für die Output-Neuronen verwendet.

`INPUT FUTURE Tegut|AAA|229104|Wochen.PRS(5) TREND -1 0 1`
: Die Zeitreihe `Tegut|AAA|229104|Wochen.PRS` wird vorverarbeitet (mit `TREND -1 0 1`). Die Zeitreihe wird für 5 Input-Neuronen verwendet und ist zum Prognosezeitpunkt bekannt (`FUTURE`).

`PARAM EPOCHEN 2000`
: Die Zeile wird ungeprüft in die zweite Steuerdatei geschrieben. Der Parameter `EPOCHEN` setzt beispielsweise die maximale Anzahl der Trainingsepochen für das Neuronale Netz.

Die Befehle `INFILE`, `TOOL`, `DEFINED`, `OUTPUT` und `INPUT` werden nicht in die zweite Steuerdatei geschrieben, sondern nur intern benutzt. Die Befehle `MEASURE` und `ADVANCE` werden ausgewertet und in die zweite Steuerdatei geschrieben. Der Befehl `PARAM` wird nicht ausgewertet, sondern direkt in die zweite Steuerdatei übernommen. Einfache Kommentare (mit '`#`') werden ignoriert, doppelte Kommentare ('`##`') werden in die zweite Steuerdatei als Kommentar übernommen. Endet eine Zeile mit dem Zeichen '\', so wird diese mit der nächsten Zeile verbunden. Dies wird meist benutzt, um lange Verarbeitungspipelines lesbar einzutragen, dabei kann jetzt jede Vorverarbeitung auf einer Zeile stehen. Statt fester Komponentennamen kann auch der String `$KOMPn` eingetragen werden, wobei `n` der Nummer der Komponente entspricht. Diese Strings werden zur Laufzeit durch die übergebenen Komponenten ersetzt.

Nach dem Start wird die Steuerdatei gelesen und ausgewertet. Ein- und ausgehende Zeitreihen werden auf die mit `MEASURE` angegebene Zeiteinheit umgerechnet (mit `MDAggregate`), auf das in `DEFINED` angegebene Intervall abgeschnitten (mit `MDCrop`) und danach mit der bei der Zeitreihe angegebenen Vorverarbeitungspipeline vorverarbeitet. Danach wird die Prognose mit Hilfe des in `TOOL` angegebenen Verfahrens gestartet. Die entstandene Zeitreihe wird gesichert und im Index eingetragen.

5.3 Konverter

Das Konvertieren von Zeitreihen kann aus mehreren Gründen nötig sein. Der einfachste Fall ist, daß die Daten in einem anderen Format vorliegen als gewünscht. Man benötigt also ein Programm, das das vorliegende Format lesen kann und die Daten danach in dem gewünschten Format speichern kann. Das Problem kann man dadurch lösen, daß man die Einlesemethoden mit in das Hauptprogramm des Konverters einbezieht. Dann muß man aber für jedes Format einen Konverter schreiben oder alle Einlesemethoden im Hauptprogramm implementieren. Eleganter ist der Weg, für beide Formate ein persistentes Objekt zu implementieren und dann das eine zum Lesen und das andere zum Schreiben zu benutzen.

Eine andere Art der Konvertierung ist eine Speicherung der Zeitreihe nach einer Vorverarbeitung, z.B., wenn die Zeitreihe anders gewichtet oder geglättet vorliegen muß. Dazu implementiert man das gewünschte Verfahren als Vorverarbeitung und skaliert die Zeitreihe entsprechend.

Für beide Fälle sind hier einfache Beispiele angebracht:

5.3.1 Zeitreihen als PICTEX-Makros speichern

Dies entspricht dem Auswechseln des persistenten Objekts zur Laufzeit. Eine Zeitreihe wird hier mit Hilfe des Objekts `PSFile` geladen, d.h. aus dem Datenformat des bestehenden Systems, und danach mit Hilfe eines anderen Objekts, `PSPictex`, wieder gespeichert. Somit erhält man eine PICTEX-Datei, die die Zeitreihe graphisch darstellt (diese Art der Konvertierung wurde für Abbildung 2.1 benutzt). Der Quellcode steht in der Datei `programs/Convert.cpp`.

Der Ablauf beim Konvertieren einer Zeitreihe:

1. Erzeugen der `Administration`-Klasse mit zugehörigem persistenten Objekt (hier `PSFile`) und Projekt (hier `Tegut`).
2. Laden der Zeitreihe (hier `Tegut|AAA|229104|Wochen.PRS`).
3. Erzeugen eines persistenten Objekts für das Zielformat (hier `PSPictex`).
4. Persistentes Objekt auswechseln.
5. Zeitreihe unter neuem Namen speichern (hier `Tegut|tmp/demo`: `Tegut` beschreibt das Projekt, als Ausgabedatei wird `tmp/demo.tex` benutzt).
6. Objekte löschen und `cleanup`-Methode aller Klassen aufrufen (über `Class::cleanupAllClasses()`).

Die Implementation dieses Beispiels nimmt etwa 60 Zeilen in Anspruch. Der Aufruf des Programms erfordert keine Parameter. Die Zeitreihe wird hier in einem Verzeichnis `tmp` gespeichert, dieses Verzeichnis muß vor dem Programmstart existieren.

```
#define PROJECT "Tegut"
#define ZRNAME  "Tegut|AAA|229104|Wochen.PRS"
#define ZRSAVE  "Tegut|tmp/demo"

TSBase *zr;

// Initialisierungs-Objekt fuer PSFile mit Basisverzeichnis $PROJEKT_HOME
PSFileInit *init = NEW(init, PSFileInit, (getenv("PROJEKT_HOME")));
// mit dem Initialiesierungsobjekt eine Instanz von PSFile erzeugen
PSFile *pers = NEW(pers, PSFile, (init));

// Administrations-Klasse: "PSFile" eintragen und PROJEKT setzen
Administration *admin = NEW(admin,  Administration, (pers,PROJECT));

// Zeitreihe laden (mit Hilfe von PSFile)
if( (zr = TSBase::load(ZRNAME)) != NULL) {

    // PSPictex-Objekt erzeugen und anstatt von PSFile benutzen
    admin->changePersistent( NEW(tmp-var, PSPictex, ()) );

    // Zeitreihe speichern (mit Hilfe von PSPictex)
    zr->save(ZRSAVE);

    // Zeitreihenobjekt wieder freigeben
    if(zr) DELETE(zr);
}
else
    Message::error("Zeitreihe "ZRNAME" nicht gefunden.\n");

// Aufraeumen
if(admin) DELETE(admin);
Class::cleanupAllClasses();
```

Datei `programs/Convert.cpp`: Konvertieren in PICTEX-Makros (nur `main()`-Methode)

5.3.2 Zeitreihen bereinigt oder aggregiert speichern

Als Beispiel zum Umgang mit den Vorverarbeitungen wird hier eine Rohzeitreihe geladen, vorverarbeitet und unter einem neuen Namen wieder gespeichert. Auf diese Weise kann man benötigte Abwandlungen von Zeitreihen öffentlich zugänglich machen. Im folgenden Beispiel wird aus einer tageweisen Zeitreihe eine monatliche berechnet und gespeichert. Die entstandene Zeitreihe ist auch unter der Oberfläche sichtbar. Der Quellcode steht in der Datei `programs/Vorverarbeitung.cpp`.

Das Programm läuft in folgenden Schritten ab:

1. Erzeugen der `Administration`-Klasse mit zugehörigem persistenten Objekt (hier `PSFile`) und Projekt (hier `Tegut`).
2. Laden der Zeitreihe (hier `CO2|Tage.CO2`).
3. Vorverarbeiten der Zeitreihe (hier Aggregieren auf Monate).
4. Zeitreihe unter neuem Namen speichern (hier `CO2|Monate.CO2`).
5. Objekte löschen und `cleanup`-Methode aller Klassen aufrufen (über `Class::cleanupAllClasses()`).

Das Beispiel kommt mit etwa 50 Zeilen Quellcode aus. Beim Aufruf sind keine Parameter nötig.

```
#define SCALE   "AGGREGATE Monate 1"
#define PROJECT "CO2"
#define ZRNAME  "CO2|Tage.CO2"
#define ZRSAVE  "|Monate.CO2"

TSBase *zr;

// Applikations-Klasse: registriert Standard-Objekte

// Initialisierungs-Objekt fuer PSFile mit Basisverzeichnis $PROJEKT_HOME
PSFileInit *init = NEW(init, PSFileInit, (getenv("PROJEKT_HOME")));

// Administration-Klasse: "PSFile" eintragen und PROJECT setzen
Administration *admin = NEW(admin, Administration, ( (PSFile*)PSFile::anew(init),
    PROJECT ));

// Zeitreihe laden
if( (zr = TSBase::load(ZRNAME)) != NULL) {

    // Vorverarbeitung durchfuehren
    zr = admin->scale(zr, NULL, SCALE);

    // vorverarbeitete Zeitreihe speichern,
    //    mit Eintrag in den INDEX
    zr->save(ZRSAVE, 1);

    // Zeitreihenobjekt loeschen
    if(zr) { DELETE(zr); zr=NULL; }
}
else {
    Message::error("Zeitreihe "ZRNAME" nicht gefunden.\n");
}
```

```
// Aufraeumen
if(admin) { DELETE(admin); admin=NULL; }
Class::cleanupAllClasses();
```

Datei `programs/Vorverarbeitung.cpp`: Vorverarbeitete Zeitreihen speichern (nur `main()`-Methode)

Kapitel 6

Erweiterung der bestehenden Funktionen

Mit der bestehenden Klassenbibliothek lassen sich bereits viele Probleme lösen, die bei der Verwaltung von Zeitreihen entstehen. Falls aber besondere Methoden gebraucht werden oder einfach ein neues Projekt hinzukommt, kann die Hierarchie leicht erweitert werden. Dieses Kapitel geht im Detail auf die Erweiterung der bestehenden Klassenhierarchie ein.

Neue Hauptprogramme, die die Klassenbibliothek nur anwenden, unterliegen keinen festen Regeln. Es ist am einfachsten, das Grundgerüst aus einem Beispiel des letzten Kapitels zu übernehmen und die benötigte Funktionalität darin einzubetten.

6.1 Grundsätzliche Regeln für neue Klassen

Bevor man eine neue Klasse hinzufügt, müssen einige Regeln festgelegt werden. Dies betrifft sowohl Aspekte wie Klassennamen als auch fest vorgegebene Teile des Programmcodes. Ohne Wissen dieser Regeln kann die neue Klasse nicht korrekt in das System eingebettet werden.

6.1.1 Klassennamen

Da die Klassen nach Namen gesucht werden können, muß der Name der neuen Klasse einem Schema entsprechen, falls die Klasse in der Hierarchie unterhalb einer existierenden Basisklasse angelegt wird. Das Kürzel der Vaterklasse muß übernommen werden (eine Liste der Kürzel ist in Tabelle 3.1 verfügbar). Der Text dahinter ist frei wählbar, darf sich allerdings natürlich nicht mit einer anderen existierenden Klasse decken. Das erste Zeichen nach dem Kürzel muß groß geschrieben werden, der Rest klein (siehe Tabelle 6.1).

Tabelle 6.1: Beispiele für neue Klassennamen

neue Klasse	Vaterklasse	
`XXMillisekunden`	`TMBase`	falsch, da falsches Kürzel
`TMMilliSekunden`	`TMBase`	falsch, da Großbuchstabe 'S' in der Mitte
`TMMillisekunden`	`TMBase`	richtig

6.1.2 Registrierung der Klasse

Da die komplette Klassenhierarchie durch die Registrierung bekannt sein muß, muß sich auch jede neue Klasse an dieses Verfahren halten. Zu diesem Zweck muß jede neue Klasse folgende Methoden implementieren (siehe Abschnitt 4.2):

`static Class* anew( ClassInit* init )`
: Erzeugt eine neue Instanz der Klasse, dabei kann ein Initialisierungsobjekt übergeben werden.

`virtual const char* getParentName()`
: Liefert den Namen der Vaterklasse, am einfachsten über das Konstrukt `Vaterklasse::getObjectName()`.

`virtual const char* getObjectName()`
: Liefert den Namen der eigenen Klasse (s.u.)

Den Namen der eigenen Klasse speichert man in einer `private: static const char *name` Variablen und initialisiert diese wie folgt:

```
const char* MyClass::name = registerClass(
         "MyClass",
         "MyParent",
          MyClass::anew,
          MyClass::cleanup,
          MyClass::hook );
```

Dabei entspricht `MyClass` dem Namen der eigenen Klasse, `MyParent` ist der Name der Vaterklasse. Eine Beschreibung der Parameter ist in Abschnitt 4.2 nachzulesen.

6.2 Neue Zeitreihenobjekte

In der bestehenden Klassenhierarchie wurde bereits zwischen numerischen und boolschen Zeitreihen unterschieden. Um eine feinere Gliederung zu erhalten, muß eine der beiden Klassen abgeleitet werden. Es muß anhand von Informationen aus dem Kopf der Zeitreihe entscheidbar sein, welcher Untertyp zum Tragen kommt. Entscheidungen, die zur Vaterklasse geführt haben, müssen auch in abgeleiteten Klassen ihre Gültigkeit behalten (z.B. haben Subklassen von `TSBinary` auch alle nur boolsche Werte).

Folgende Methoden sind zur Bestimmung der korrekten Zeitreihenklasse beim Laden nötig:

`static int checkTSClass( TSHead* head )`
: Testet anhand von Informationen aus dem Kopf `head`, ob diese Klasse (oder eine ihrer Subklassen) zuständig ist. Liefert einen Wert ungleich 0 bei Erfolg. **Zwingend!**

`static int hook( int method, va_list args )`
: Diese Methode wird zur Bestimmung der richtigen Klasse beim Laden einer Zeitreihe benutzt. Dabei wird die Methode mit dem Parameter `method == ADD_CHECK_CLASS` aufgerufen, als zweiter Parameter wird ein Zeitreihenkopf vom Typ `TSHead*` übergeben. Die Methode muß prüfen, ob der Kopf zu dieser Klasse paßt. Weitere Funktionen können über andere Werte von `method` hinzugefügt werden. Außerdem muß die Methode beim Registrieren als letztes Argument mit angegeben werden (siehe Abschnitt 4.2). **Zwingend!**

`virtual void afterLoad()`
: Scannt nach dem Laden der Zeitreihe durch den Kopf und entfernt alle klasseninternen Zusätze. Beispielsweise sichert hier die Klasse `TSNumeric` alle statistischen Werte wie Minimum, Maximum, Mittelwert, etc. Die Informationen müssen hier auch aus dem Kopf entfernt werden. **Optional!**

`virtual void beforeSave()`
: Fügt die klasseninternen Zusätze in eine spezielle Liste von Informationen ein (`addon`-Liste). Diese Informationen werden zusätzlich zu allen anderen Informationen in den Kopf der Zeitreihe geschrieben. Falls diese Methode in einer Klasse überschrieben wird, so muß **immer** die Methode der Vaterklasse aufgerufen werden, da ansonsten keine korrekte Behandlung der `addon`-Liste stattfindet (in `TSBase`). **Optional!**

`virtual void afterSave()`
: Hier können Modifikationen an der Zeitreihe nach dem Speichern vorgenommen werden. Falls diese Methode in einer Klasse überschrieben wird, so muß auch hier **immer** die Methode der Vaterklasse aufgerufen werden, da ansonsten keine korrekte Behandlung der `addon`-Liste stattfindet (in `TSBase`). **Optional!**

`virtual void recalc()`
: Berechnet alle klasseninternen Daten der Zeitreihe neu. Es muß hier die `recalc()`-Methode der Vaterklasse aufgerufen werden, damit auch alle anderen internen Werte neu berechnet werden. **Optional!**

`virtual double aggregate( double values[], int len )`
: Aggregiert die ersten `len` Werte aus dem Wertearray `values` zu einem Wert. Dieser wird zurückgegeben. Die Methode muß nur überschrieben werden, falls eine andere Aggregation eingeführt wird. **Optional!**

Die Methoden `afterLoad()`, `beforeSave()`, `afterSave()` und `recalc()` müssen nur überschrieben werden, wenn neue Informationen über die Zeitreihe in dieser Klasse gesammelt werden. Andernfalls kann die Methode der Vaterklasse benutzt werden.

Die Methode `aggregate()` wird in den wenigsten Fällen neu implementiert. Die Klasse `TSNumeric` hat dies nicht getan, die Aggregation in der Basisklasse ist ausreichend. Die Klasse `TSBinary` hat dagegen eine eigene `aggregate()`-Methode implementiert, da boolsche Werte mit Hilfe von boolschen Operatoren aggregiert werden müssen (`UND`- und `ODER`-Verknüpfungen.

6.3 Neue Datenformate

Als persistente Objekte wurden bisher das bestehende Datenformat in `PSFile` und ein Export nach PICTEX in `PSPictex` implementiert. Denkbar wären aber auch Objekte, die direkt auf eine Datenbank zugreifen. Um eine neues persistentes Objekt zu erstellen, ist einfach die Basisklasse `PSBase` abzuleiten.

`virtual TSBase* load( const char* name )`
: Lädt eine Zeitreihe des Namens `name`. Normalerweise sollte der Name aus den Komponenten bestehen, getrennt durch ein Pipe-Symbol ('`|`'). Die Methode muß zuerst die Kopfinformationen laden, daraufhin entscheiden, zu welcher Klasse von Zeitreihen der Kopf gehört, eine Instanz dieser Klasse erzeugen, die Zeitreihenwerte laden und in dieser Instanz speichern. Liefert einen Zeiger auf das geladene Objekt oder `NULL` bei einem Fehler.

`virtual int save( TSBase* zr, const char* name, int index=0 )`
: Speichert eine Zeitreihe `zr` unter dem Namen `name`. Ist zusätzlich der optionale Parameter `index == 1` gegeben, so wird die Zeitreihe auch in einem Index vermerkt. Es müssen zwei Listen von Informationen gespeichert werden, auf die man über die Methoden `getTSInfos()` und `getTSAddon()` Zugriff bekommt. Die `addon`-Liste ist nur temporär und wird in den Methoden `beforeSave()` und `afterSave()` des Zeitreihenobjekts verwaltet.

`virtual int destroy( const char* name )`
: Löscht die gespeicherte Zeitreihe des Namens `name`. Bei erfolgreichem Löschen wird `1` zurückgeliefert, ansonsten `0`.

`virtual int exists( const char* name )`
: Prüft, ob eine Zeitreihe des Namens `name` existiert. Wenn dies der Fall ist, wird `1` zurückgegeben, ansonsten `0`.

`virtual const char* basedir()`
: Liefert das Basisverzeichnis, an das ein projektspezifisches Verzeichnis angehängt wird. In das resultierende Verzeichnis werden später Log-Dateien und ähnliches geschrieben.

`virtual int getPrognose( TSBase* zr, String* desc, String* data )`
: Liefert die Nummer einer Prognose. Dabei sind `zr` die prognostizierte Zeitreihe, `desc` eine Beschreibung der Parameter der Prognose und `data` die zugehörigen Parameter. Diese Methode ist zur Zeit nicht in `PSFile` implementiert.

`virtual int setPrognose( TSBase* zr, String* desc, String* data )`
: Trägt eine Prognose in eine Liste von Prognosen ein. Dabei entsprechen die Parameter denen der Methode `getPrognose()`. Auch diese Methode ist zur Zeit nicht in `PSFile` implementiert.

6.4 Neue Vorverarbeitungen

Falls das Angebot an Vorverarbeitungen nicht ausreichen sollte, kann dies ebenfalls auf einfache Weise erweitert werden. Es müssen lediglich zwei Methoden implementiert werden, die der Hin- und Rückrichtung der Vorverarbeitung entsprechen, die Methodenköpfe sind dabei von der Basisklasse `MDBase` gegeben:

`virtual TSBase* scale( TSBase* zr, List<TSBase*>* add, const char* desc )`
: Wendet die Vorverarbeitung auf die Zeitreihe `zr` an. Dabei können weitere Zeitreihen einfließen, diese werden in einer Liste `add` von Zeitreihen übergeben. Werden keine weiteren Zeitreihen benötigt, kann hier einfach `NULL` übergeben werden. Zusätzlich wird eine Zeichenkette übergeben, die die Parameter der Vorverarbeitung beinhaltet. Die Methode muß selbst die Art der Vorverarbeitung inklusive aller Parameter, die zur Rückrechnung nötig sind, mit Hilfe von `addPrefix()` in den Kopf der Zeitreihe eintragen. Als Rückgabewert wird ein Zeiger auf das entstandene Zeitreihenobjekt zurückgegeben, im Normalfall ist dies das Objekt `zr` selbst.

`virtual TSBase* descale( TSBase* zr, const char* desc )`
: Rechnet die Vorverarbeitung wieder zurück. Dabei ist die `zr` eine mit dieser Vorverarbeitung behaftete Zeitreihe, `desc` enthält die Parameter der Vorverarbeitung. Liefert einen Zeiger auf das resultierende Zeitreihenobjekt, im Normalfall ist dies `zr` selbst.

Da jede Vorverarbeitung die Parameter als Zeichenkette bekommt, ist eine Parsierung in beiden Methoden nötig. Zu diesem Zweck wird die Klasse `StringParser` benutzt (siehe Abschnitt 7.1.3), die auf einfache Weise Wörter und Klammerungen aus dem String liefern kann. Als Grundgerüst kann man jede bereits implementierte Vorverarbeitung nehmen und auf die gewünschte Art und Weise verändern.

Manche Vorverarbeitungen lassen sich nicht rückgängig machen, in diesem Fall bietet sich ein Aufruf von `Message::error()` an (siehe Abschnitt 7.1.6), der das Programm mit einer entsprechenden Meldung abbricht. Vorverarbeitungen, die in der Hinrichtung keine Änderungen machen (z.B. die Vorverarbeitung `MDInteger`, siehe Abschnitt 7.6.2), tragen nur die Vorverarbeitung in den Kopf der Zeitreihe ein und liefern das Zeitreihenobjekt `zr` ansonsten unverändert zurück.

6.5 Neue Zeiteinheiten

Ein wichtiger Bestandteil der Zeitreihen ist die zugrundeliegende Zeiteinheit. Da hier eine Umrechnung von einer Einheit in alle anderen möglich sein muß, ist die Erweiterung um eine neue Zeiteinheit schwieriger als bei anderen Klassen.

sämtliche Konstruktoren von `TMBase`
: Die Konstruktoren müssen die zugehörigen Konstruktoren von `TSBase` aufrufen und danach die nötigen Berechnungen des Datums vornehmen.
Beispiel: Die `TMTage`-Klasse setzt die Uhrzeit immer auf 00:00:00 Uhr um.

`virtual void recalc_logical()`
Berechnet anhand des realen Originaldatums das logische Datum `tm_count` neu.
Beispiel: `TMJahre` berechnet aus dem Jahr 1998 und dem Faktor 1 den logischen Tick 54.

`virtual void recalc_real()`
Berechnet anhand des logischen Datums `tm_count` das reale Datum neu (das Originaldatum bleibt dabei erhalten!).
Beispiel: `TMJahre` berechnet aus dem Tick 30 und dem Faktor 1 das Jahr 1974)

`virtual double* aggregateTo( const double* values, TSBase* zr, TMBase* start long end, TMBase* to_start, long to_end )`
Aggregiert die Werte in `values` von der Zeiteinheit dieser Klasse in eine andere. Dabei gibt `start` den Start der eingehenden Zeitreihe an (inklusive Zeiteinheit), `end` ist das logische Datum des letzten Wertes. Das erste Datum der Zieleinheit wird in `to_start` übergeben, das letzte logische Zieldatum in `to_end`.

`virtual double* aggregateFrom( const double* values, TSBase* zr, TMBase* start long end, TMBase* to_start, long to_end )`
Aggregiert genau wie `aggregateTo()`, nur daß nicht von der Zeiteinheit in eine andere aggregiert wird, sondern einer beliebigen Zeiteinheit in diese hinein.

Alle grundlegenden Zeiteinheiten sind bereits implementiert, so daß hier wahrscheinlich keine Erweiterungen nötig sind. Sollte eine Zeiteinheit hinzukommen, die sich nicht mit den zur Verfügung gestellten Variablen (Datum, Uhrzeit, Kalenderwoche, Wochentag, Quartal) darstellen läßt, so muß auch die Basisklasse `TMBase` geändert werden.

Die Aggregationen sind allerdings noch nicht für alle Zeiteinheiten implementiert worden, lediglich Tage und Monate können in alle gröberen Einheiten umgewandelt werden. Diese sind als erstes implementiert worden, da vorrangig diese Einheiten in den vorhandenen Projekten aggregiert werden mußten.

6.6 Neue Prognoseverfahren

Das Ziel der Projektarbeit war es, Zeitreihen zu prognostizieren. Dabei kommen verschiedene Verfahren zum Einsatz. Soll ein weiteres Verfahren angewandt werden, welches zur Zeit nicht implementiert ist, so kann dieses einfach in die Hierarchie eingegliedert werden.

`virtual TSBase* prognose( TSBase* zr, List<TSBase*>* add, const char* data )`
Diese Methode führt die eigentliche Prognose aus. Dabei ist `zr` die zu prognostizierende Zeitreihe, in der Liste `add` stehen weitere eingehende Zeitreihen (oder `NULL`). Der übergebene String `data` enthält alle Parameter für das Verfahren (analog zu den Vorverarbeitungen).

Soll das Verfahren die Parameter aus einer ASCII-Steuerdatei erhalten, so kann die neue Klasse auch von `PRInfile` abgeleitet werden. Diese Klasse bietet schon grundlegende Methoden zur Kommunikation mit einem Steuerfile an. Dabei muß nicht mehr die komplette Methode `prognose()`, sondern es müssen nur Teile der Funktionalität in den folgenden Methoden implementiert werden:

`virtual int info( TSBase* zr, void* data )`

Liefert zu einer Zeitreihe `zr` die prognosespezifischen Informationen (Diese sind textuell gespeichert und können mit `getObjectData()` abgefragt werden). Diese wird in einer Struktur `data` gesichert, die typenlos übergeben wird (d.h., hier ist ein Cast nötig).

Beispiel `PRBP`: Aus dem String `PRESENCE 5` wird die Struktur `PRBP_DATA` gefüllt.

`virtual void prognose_get_parameters( StringParser* str )`

Parsiert den Parameterstring `str` und extrahiert daraus alle Parameter der Zeitreihe. Der erste Parameter **muß** der Name der Steuerdatei sein. Dieser wird bereits vor dem Aufruf der Methode entfernt, so daß in `str` alle Parameter ab dem zweiten enthalten sind.

Beispiel `PRBP`: Aus dem String `TRAIN,tst` wird der Modus (`TRAIN` = Trainings-Zeitreihe erzeugen) und der Kommentar `tst` extrahiert und in Objektvariablen gesichert.

`virtual void prognose_test_parameters( String* bpfile )`

Testet alle Parameter der Prognose aus `prognose_get_parameters()` auf zulässige Werte. Dabei ist `bpfile` der Name der Steuerdatei.

`virtual void prognose_write_output( FILE* fp, TSBase* zr )`

Schreibt die Beschreibung für die Output-Zeitreihe `zr` in die Steuerdatei mit dem Dateihandle `fp`, dabei können die Informationen der Zeitreihe benutzt werden (aus `info()`).

Beispiel `PRBP`: Aus der Struktur `PRBP_DATA` und der übergebenen Zeitreihe wird eine Zeile in die Steuerdatei eingetragen: `OUTPUT Komp1 Komp2 Komp3 OUT`

`virtual void prognose_write_input( FILE* fp, TSBase* zr, int count )`

Schreibt die Beschreibung für eine Input-Zeitreihe `zr` in die Steuerdatei mit dem Dateihandle `fp`. Die laufende Nummer der Zeitreihe wird in `count` übergeben. Auch hier können Informationen aus `info()` benutzt werden.

`virtual void prognose_commands( String* bpfile, TSBase* zr )`

Führt die eigentliche Prognose aus. Normalerweise wird hier ein externes Programm ausgeführt, welches die Steuerdatei `bpfile` liest und eine zugehörige Prognose startet. Die zu prognostizierende Zeitreihe wird in `zr` übergeben.

`virtual void prognose_postprocessing( const char* zrfile, const char* zrname )`

Wird nach der Prognose ausgeführt und dient der Nachbearbeitung der Ergebnisse, beispielsweise kann hier rückskaliert werden. Dabei steht in `zrfile` der Dateiname der Zeitreihe, in `zrname` der Name der Zeitreihe inklusive aller Komponenten.

Beispiel: ZRName entspricht einem kompletten Zeitreihennamen inklusive Komponenten `Tegut|AAA|229104|Wochen.ABV`, ZRDateiname nur dem eigentlichen Dateinamen der Zeitreihe `Wochen.ABV`.

`virtual void prognose_cleanup( const char* zrname )`

Wird nach der Nachbearbeitung aufgerufen und kann zum Aufräumen verwendet werden (z.B. um temporäre Dateien der Prognose zu löschen). Dabei ist `zrname` der komplette Name der Prognosezeitreihe inklusive aller Komponenten.

Beispiel: ZRName entspricht einem kompletten Zeitreihennamen inklusive Komponenten `Tegut|AAA|229104|Wochen.ABV`.

6.7 Neue Projekte

Im Laufe der Projektarbeit sind mehrere Projekte hinzugekommen. Da auch weiterhin neue Projekte leicht aufgenommen werden sollen, ist eine Erweiterung auch hier mit wenigen Änderungen möglich. Dabei müssen neue Projekte von `PJBase` abgeleitet werden und die folgenden Methoden überschreiben:

`int getTypeCnt()`
: Liefert die Anzahl der verschiedenen Typen im Projekt.
Beispiel `PJTegut`: Es werden 7 Typen definiert: Abverkauf, Aktionen, Artikelumsatz, Kunden, Preis, Regalumsatz und Gesamtumsatz.

`int getKompCnt()`
: Liefert die Anzahl der Komponenten im Projekt.
Beispiel `PJTegut`: Es werden 3 Komponenten definiert: Projekt, Filiale und Artikelnummer.

`const char* get_token( int type )`
: Liefert zum Typ `type` das passende Suffix.
Beispiel `PJTegut`: Typ 1 liefert das Suffix `ABV`.

`const char* get_name( String komp[], int kanz, int type, int lang )`
: Liefert zu den angegebenen Komponenten `komp` (Anzahl Komponenten = `kanz`), dem Typ `type` und der Sprache `lang` den Typnamen der Zeitreihe.
Beispiel `PJTegut`: Für die Komponenten `Tegut|AAA|229104` (`kanz=3`), den Typ 1 und die Sprache `Language::DEUTSCH` erhält man den Typnamen `Abverkauf`.

`const char* get_desc( String komp[], int kanz, int type, int lang )`
: Liefert zu den angegebenen Komponenten `komp` (Anzahl Komponenten = `kanz`), dem Typ `type` und der Sprache `lang` die Beschreibung der Zeitreihe.
Beispiel `PJTegut`: Für die Komponenten `Tegut|AAA|229104` (`kanz=3`), den Typ 1 und die Sprache `Language::DEUTSCH` erhält man die Beschreibung `010 KOELLNFL.BLUE.ZART500 HAFERFLOCKEN` (aus der Datei `ARTIKEL.TXT`).

`const char* get_komp( int kompnr, int lang )`
: Liefert zur angegeben Komponentennummer `kompnr` und der Sprache `lang` den Namen der Komponente.
Beispiel `PJTegut`: Für die Komponente 2 und die Sprache `Language::DEUTSCH` erhält man den Namen `Filiale`.

`const char** get_komp( int kompnr )`
: Liefert zur angegebenen Komponentennummer `kompnr` ein Array mit den Komponentennamen in allen Sprachen.
Beispiel `PJTegut`: Für die zweite Komponente erhält man das Namen-Array
{ `"Filiale", "branch"` }.

`String* getIndexHeading( String komp[], int kanz )`
: Liefert für die übergebenen Komponenten `komp` (Anzahl Komponenten = `kanz+1`) die Überschrift der zugehörigen `INDEX`-Datei.
Beispiel `PJTegut`: Für Komponenten `Tegut|AAA` (`kanz`=1) bekommt man die Überschrift `Filialen`.

`String* getIndexEntry( TSHead* head )`
Liefert zu einem Zeitreihenkopf `head` den passenden Eintrag im `INDEX`-Datei.
Beispiel `PJTegut`: Hier wird einfach ein String der Form `Typ, Zeiteinheit`, also beispielsweise `Abverkauf, Wochen` zurückgegeben.

`String* getIndexType( int isprog )`
Liefert den Typ des Indexeintrages zurück. Dabei bezeichnet ein Typ 2 eine Zeitreihe, ein Typ von 6 eine globale Zeitreihe. Alle anderen Typen sind reserviert und werden nur intern benutzt (z.B. um eine Komponente zu kennzeichnen).

`int gotoProjectDir( TSHead* head, const char* comment, int mkdir )`
Wechselt in ein projektspezifisches Verzeichnis. In diesem können dann während der Prognose Log-Dateien oder ähnliches abgelegt werden. Dabei werden Informationen aus dem Kopf `head` und dem Kommentar `comment` benutzt. Bei Angabe von `mkdir == 1` wird ein nicht existierendes Verzeichnis angelegt.
Beispiel `PJTegut`: Es wird in das Verzeichnis
`Tegut/$PROGNOSE/$PROGNOSE_SUB/<Filiale>/<Artikel>/<Kommentar>` gewechselt.

Kapitel 7

Beschreibung der Klassen

In diesem Kapitel werden alle Klassen der Implementation aufgelistet. Zu jeder Klasse werden die Methoden angegeben, die gegenüber der Vaterklasse neu dazugekommen sind oder die Methoden der Vaterklasse überschreiben. Dieses Kapitel ist auch als Nachschlagehilfe bei der Erweiterung der Klassenhierarchie zu verwenden, da hier alle Methoden schnell gefunden werden können. Eine Übersicht über alle implementierten Klassen ist in Abbildung 7.1 zu sehen.

Im folgenden wird die C++-typische Schreibweise für Methodenköpfe benutzt, Tabelle 7.1 erklärt kurz die benutzte Syntax.

Tabelle 7.1: Bedeutungen der C++-Schreibweise

`void methode( int arg = wert )`	Der Parameter `arg` ist optional und muß beim Aufruf der Methode nicht übergeben werden. In diesem Fall nimmt der Parameter der angegeben Wert `wert` an.
`void methode() = 0`	Die Klasse definiert nur den Kopf der Methode, allerdings wird kein Methodenrumpf implementiert. Die entstandene Klasse ist **abstrakt**, und es kann keine Instanz der Klasse erzeugt werden. Die Methode muß in Subklassen definiert werden.
`virtual methode()`	Die Methode wird erst zur Laufzeit gebunden. Im Gegensatz zur statischen Bindung durch den Linker wird hier ein Zeiger auf die Methode intern an das Objekt gebunden. Dadurch wird immer die zum Objekt gehörende Methode aufgerufen, selbst wenn das Objekt einer Variablen vom Typ einer Basisklasse zugewiesen wurde.

```
Class
├─TSHead
│  └─TSBase
│     ├─TSBinary
│     └─TSNumeric
├─PSBase
│  ├─PSFile
│  └─PSPictex
├─PJBase
│  ├─PJC02
│  ├─PJGlobal
│  └─PJTegut
├─TMBase
│  ├─TMSekunden
│  ├─TMMinuten
│  ├─TMStunden
│  ├─TMTage
│  ├─TMWochen
│  ├─TMWochenTag
│  ├─TMMonate
│  ├─TMQuartale
│  ├─TMJahre
│  └─TMTicks
├─MDBase
│  ├─MDAggregate
│  ├─MDAround0
│  ├─MDAverage
│  ├─MDBinary
│  ├─MDCrop
│  ├─MDDays
│  ├─MDDiff
│  ├─MDExpbinary
│  ├─MDExpo
│  ├─MDInteger
│  ├─MDLinear
│  ├─MDMy_sigma
│  ├─MDNone
│  ├─MDShift
│  ├─MDTrend
│  └─MDUnsigned
├─PRBase
│  ├─PRInfile
│  │  ├─PRBP
│  │  ├─PRGauss
│  │  └─PRSNNS
│  └─PRNaiv
└─Administration

ClassInit
├─TSHeadInit
├─PSFileInit
└─TimeInit

Message
Observer
System
ProjectManager
List
└─PList
String
└─StringParser
Language
LanguageString
ClassData
```

Abbildung 7.1: Komplette Klassenhierarchie

7.1 Hilfsklassen außerhalb der Klassenhierarchie

Bei der Programmierung einer so großen Menge von Klassen treten immer wieder ähnliche Probleme auf. Für viele dieser Probleme sind dabei Hilfsklassen entstanden, die diese Probleme lösen sollen. Diese Hilfsklassen werden vor den eigentlich interessanten Klassen kurz angesprochen.

7.1.1 Verkettete Listen: `List` und `PList` (`std/List.inc`, `std/PList.inc`)

Um eine Liste von Objekten zu verwalten, benötigt man immer wieder den abstrakten Datentyp der Liste. Hier wurde eine doppelt verkettete Liste als Template-Klasse implementiert. Die Klasse benutzt dabei intern ein dynamisch wachsendes Array von Objekten. `List` und `PList` unterscheiden sich nur darin, daß bei `PList` die Objekte vor dem Entfernen gelöscht werden, d.h., `PList` ist für dynamisch angelegte Objekte entstanden. In `List` kann man dagegen durch das fehlende Löschen auch die eingebauten Datentypen wie `int` oder `double` verwalten. `PList` ist von der Klasse `List` abgeleitet und erbt somit bereits alle Methoden zur Verwaltung einer Liste.

Der Typ der verwalteten Objekte wird im folgenden mit `T` bezeichnet.

`public`-Methoden:

`List( T def, int startup=10, int increase=10 )`
: Erzeugt eine neue Liste. Die Liste hat am Anfang Platz für `startup` Objekte. Bei Platzmangel wird die Liste um mindestens `increase` Elemente vergrößert. Es muß ein Standardwert `def` für nicht existierende Objekte übergeben werden. Dieser wird geliefert, wenn `getAt()` einen nicht existierenden Index abfragt. Werden keine Werte für `startup` und/oder `increase` übergeben, so werden die Standardwerte `startup=10` und `increase=10` angenommen.

`virtual ~List()`
: Destruktor der Liste, ruft `clearAll()` auf.

`int isEmpty()`
: Liefert einen Wert ungleich 0, falls die Liste leer ist.

`int isFirst()`
: Liefert einen Wert ungleich 0, falls Position auf erstem Element der Liste ist.

`int isEOL()`
: Liefert einen Wert ungleich 0, falls Position hinter letztem Element der Liste ist.

`void first()`
: Setzt die Position auf das erste Element.

`void next()`
: Setzt die Position auf das nächste Element (maximal auf die Position hinter dem letzten Element).

`void prev()`
: Setzt die Position auf das vorige Element (maximal auf die Position des ersten Elementes).

`T getElem()`
: Liefert das aktuelle Element.

`T getFirst()`
: Setzt die Position auf das erste Element und liefert dieses zurück.

`T getNext()`
: Setzt die Position auf das nächste Element und liefert dieses zurück. Ist kein nächstes Element mehr vorhanden, wird `NULL` zurückgegeben.

`T getPrev()`
: Setzt die Position auf das vorige Element und liefert dieses zurück. Ist kein voriges Element mehr vorhanden, wird `NULL` zurückgegeben.

`T getAt( int index )`
: Liefert das Element an der Position `index`, wobei das erste Element den Index 0 hat. Bei nicht belegten Elementen oder einem ungültigen Index wird der Standardwert geliefert. Die aktuelle Position wird dabei nicht verändert.

`void changeAt( int index, T elem )`
: Setzt das Element an der Stelle `index` auf den Wert `elem`. Dabei wird die Liste verlängert, wenn der Index hinter dem Ende der Liste liegt (Die Elemente zwischen dem letzten und dem gesetzten Element werden auf den Standardwert für Objekte der Liste gesetzt). Die Position innerhalb der Liste wird dabei nicht verändert.

`int insert( T elem )`
: Fügt das Element `elem` vor der aktuellen Position ein. Die Position zeigt auf das eingefügte Element. Liefert einen Wert ungleich 0, falls das Einfügen erfolgreich war.

`int append( T elem )`
: Hängt das Element `elem` an die Liste an. Die Position innerhalb der Liste wird dabei nicht verändert.

`void clearAll()`
: Ruft für alle Elemente `clear()` auf und löscht anschließend das interne Array.

`int remove()`
: Löscht das aktuelle Element aus der Liste (inklusive Aufruf von `clear()`).

`int forall( int action(T) )`
: Führt für alle Elemente der Liste die Methode `action()` aus, die ein Objekt des verwalteten Typs als Parameter bekommt.

`protected`-Methoden:

`virtual void clear( T elem )`
: Löscht das Element `elem`. Wird vor dem Löschen aus der Liste aufgerufen. In `List` wird das Element auf `NULL` (oder `0`) gesetzt, in `PList` wird hier `delete` aufgerufen.

`int resize( int size )`
: Vergrößert die Liste auf `size` Elemente. Ist die Liste bereits mindestens `size` Elemente lang, so ändert ein Aufruf von `resize()` nichts. Die Liste wird bei einem Aufruf von `resize()` also nicht verkleinert.

`int shift( int start, int offset )`
: Verschiebt alle Elemente ab Index `start` um `offset` Elemente nach hinten. Ist `offset` negativ, wird nach vorne verschoben. Unter Umständen wird beim Verschie-

ben die Liste größer. Beim Verschieben nach vorne können Elemente aus der Liste herausfallen, diese werden gelöscht.

Ist `_INCLUDE_TEMPLATES_` gesetzt, wird der Code für einen Typ mit compiliert. Der Code darf für jeden Typ allerdings nur einmal compiliert werden, sonst bricht der Linker mit einer Fehlermeldung ab. Zu diesem Zweck sammelt man alle Template-Klassen in einer Quelle `admin/Templates.cpp`

7.1.2 Zeichenketten verwalten: `String` (in `std/String.cpp`)

Die Verwendung von Zeichenketten in C (und C++) mittels festen Arrays ist meist zu unflexibel, die Verwaltung von dynamischen Arrays ist dagegen flexibel, aber aufwendig und fehleranfällig. Daher ist eine Klasse zur Speicherung von Zeichenketten entstanden, die auch grundlegende Methoden zur Bearbeitung zur Verfügung stellt. Die Klasse `String` implementiert die komplette dynamische Verwaltung einer Zeichenkette, der Zugriff von außen kann die Integrität des Objekts nicht aufheben.

`String` verwaltet die Zeichenkette in einem `char* str`, die Länge des String wird in `int str_length` gespeichert (damit bei mehreren Abfragen der Zeichenkettenlänge nicht mehrmals gerechnet werden muß), und der aktuell verfügbare Speicherplatz der Zeichenkette ist in `int str_size` vermerkt. Beim Erzeugen oder Setzen eines Strings kann eine Mindestgröße vorgegeben werden, damit nicht zu oft die Stringgröße angepaßt werden muß (beispielsweise, wenn zeichenweise eingelesen wird, man aber in etwa die Länge des resultierenden Strings kennt).

`public`-Methoden:

`String()`
: Erzeugt einen leeren String.

`String( const char* str, int minlen=0 )`
`String( const String& str, int minlen=0 )`
: Erzeugt einen String aus einer übergebenen Zeichenkette `str`, bzw. aus einem übergebenen String `str`. Dabei kann optional eine Mindestgröße übergeben werden.

`String( int minlen, const char* format, ... )`
: Erzeugt einen String aus einem `printf`-kompatiblen Format. In dieser Form ist die Mindestgröße als erster Parameter anzugeben (ansonsten sind die Konstruktoren mehrdeutig).

`virtual ~String()`
: Destruktor, gibt die Zeichenkette frei.

`const char* get()`
: Liefert einen Zeiger auf die Zeichenkette.

`const char* getFrom( int index )`
: Liefert einen Zeiger auf die Zeichenkette ab dem `index`-ten Zeichen. Das erste Zeichen steht bei Index `0`, ein Aufruf von `getFrom(0)` ist also gleichbedeutend mit `get()`.

`virtual String* substr( int from, int to )`
: Liefert einen neuen String, der einen Teilstring der gespeicherten Zeichenkette enthält. Der Teilstring wird durch Start- und Endindex angegeben, das erste Zeichen hat den Index `0`. Liefert `NULL`, wenn eine der Indexgrenzen nicht innerhalb der Zeichenkette liegt.

`char charAt( int index )`
: Liefert das Zeichen an der `index`-ten Stelle, das erste Zeichen hat den Index `0`.

`int length()`
: Liefert die Länge der gespeicherten Zeichenkette.

`int size()`
: Liefert den aktuell verfügbaren Speicherplatz für die Zeichenkette. Dieser Wert ist niemals kleiner als die Länge der Zeichenkette.

`int contains( const char* substr, int caseignore=0 )`
: Liefert Wert ungleich `0`, falls die Zeichenkette `str` in der gespeicherten Zeichenkette enthalten ist. Bei `caseignore == 1` wird nicht zwischen Groß-/Kleinschreibung unterschieden.

`int equals( const char* str, int caseignore=0 )`
`int equals( const String* str, int caseignore=0 )`
: Boolscher Vergleich mit der übergebenen Zeichenkette `str`, bzw. dem übergebenen String `str`. Liefert `1` bei Identität, sonst `0`. Bei `caseignore == 1` wird nicht zwischen Groß-/Kleinschreibung unterschieden.

`int compare( const char* str, int caseignore=0 )`
`int compare( const String* str, int caseignore=0 )`
: Lexikographischer Vergleich mit der übergebenen Zeichenkette `str`, bzw. dem übergebenen String `str`. Liefert `-1`, wenn `str` kleiner ist, `0` bei Identität und `1` falls `str` größer als die gespeicherte Zeichenkette ist. Bei `caseignore == 1` wird nicht zwischen Groß-/Kleinschreibung unterschieden.

`virtual set( const char* str, int minlen=0 )`
`virtual set( const String* str, int minlen=0 )`
`virtual set( const String& str, int minlen=0 )`
: Setzt den String auf die übergebene Zeichenkette `str`, bzw. den übergebenen String `str`. Dabei kann optional eine Mindestgröße übergeben werden.

`void vset( int minlen, const char* format, ... )`
: Setzt den String aus einem `printf`-kompatiblen Format. In dieser Methode ist die Mindestgröße als erster Parameter zwingend anzugeben.

`virtual int fgetline( FILE* fp, int append=0, int minlen=0 )`
: Liest eine Zeile über den gegebenen Dateihandle `fp` bis zum nächsten Zeilenumbruch ein. Der optionale Parameter `append` gibt an, ob der gespeicherte String überschrieben werden soll (`append == 0`) oder ob die Zeile angehängt wird (`append == 1`). Weiterhin kann optional eine Mindestgröße übergeben werden. Liefert `0` zurück, wenn beim Lesen das Ende der Datei erreicht wurde (ansonsten `1`).

`virtual void append( const char* str, int minlen=0 )`
`virtual void append( const String* str, int minlen=0 )`
`virtual void append( const String& str, int minlen=0 )`
: Hängt die übergebene Zeichenkette `str` bzw. den übergebenen String `str` an die gespeicherte Zeichenkette an. Dabei kann optional eine Mindestgröße übergeben werden.

`virtual int subst( const char* token, char replace, int many=0 )`
: Ersetzt alle vorkommenden Zeichen aus einer Zeichenliste `token` durch das Zeichen `replace`. Mehrere aufeinanderfolgende Token können durch ein einzelnes Zeichen ersetzt werden, indem als optionaler Parameter `many == 1` übergeben wird. Liefert die Anzahl der Ersetzungen.

`virtual int replace( const char* str1, const char* str2 )`
: Ersetzt jedes Vorkommen von `str1` in der Zeichenkette durch `str2`. Liefert die Anzahl der Ersetzungen.

`virtual int cutAfter( char chr, String* here, int front=0, int rmchr=1 )`
: Schneidet den String hinter dem ersten Vorkommen des Zeichens `chr` ab. Wird für `here` ein String übergeben, wird der abgeschnittene Teil hier gesichert (ansonsten ist `here == NULL` zu übergeben). Standardmäßig wird von hinten gesucht, optional kann mit `front == 1` auch von vorne begonnen werden. Weiterhin wird das gesuchte Zeichen mit abgeschnitten, optional kann dies mit `rmchr == 0` verhindert werden. Liefert `1` zurück, wenn das Abschneiden Erfolg hatte.

`virtual void cutTrailing( const char* group=" \t\r\n" )`
: Schneidet von hinten alle Zeichen des Strings ab, die in der übergebenen Gruppe liegen. Ohne Parameter werden alle Whitespaces entfernt.

`protected`-Methoden:

`virtual void resize( int n )`
: Vergrößert den Platz für die Zeichenkette auf `n` Zeichen. Ist der String bereits größer als `n` Zeichen, wird er jedoch nicht verkleinert.

`int compare( const char* str1, const char* str2, int caseignore )`
: Interne Methode zum Vergleichen zweier Zeichenketten `str1` und `str2`, dabei kann mit `caseignore == 1` Groß-/Kleinschreibung ignoriert werden.

7.1.3 Zeichenketten parsieren: `StringParser` (in `std/StringParser.cpp`)

Die Klasse `String` kann zwar dynamisch Strings verwalten, aber das Parsieren eines Strings, z.B. um nach Wörtern zu suchen, ist mit dieser Klasse nicht ohne Aufwand möglich. Zu diesem Zweck gibt es die Klasse `StringParser`, eine Subklasse von `String`, die einen String untersuchen kann.

Zusätzlich zum Zeiger `str` auf den Anfang der Zeichenkette (aus `String`) wird hier noch ein weiterer Zeiger `char* pos` auf die aktuelle Position in der Zeichenkette mitgeführt. Damit kann man jetzt innerhalb der Zeichenkette navigieren.

`public`-Methoden:

`StringParser(), ...`
: Konstruktoren wie bei `String`.

`virtual ~StringParser()`
: Destruktor wie bei `String`, zusätzlich wird `pos` zurückgesetzt.

`int endOfString( const char* blanks=" \t\r\n" )`
: Liefert `1`, wenn die Position am Ende des Strings angekommen ist. Dabei werden alle Zeichen aus der übergebenen Gruppe ignoriert. Ohne Parameterübergabe werden alle Whitespaces überlesen.

`virtual String* nextWord( const char* trenner, int many=1 )`
: Gibt den String von der aktuellen Position bis zum nächsten Vorkommen eines Zeichens aus der übergebenen Gruppe `trenner` zurück. Ist kein solcher Trenner vorhanden, wird der ganze String geliefert. Der optionale Parameter gibt an, ob mehrere gleiche Trenner als ein einziger interpretiert werden sollen. Diese Methode setzt die aktuelle Position auf das Zeichen hinter dem Trenner.

`virtual String* untilChar( char chr )`
: Gibt den String von der aktuellen Position bis zum nächsten Vorkommen des Zeichens `chr` zurück. Ist kein solches Zeichen vorhanden, wird `NULL` geliefert. Diese Methode setzt die aktuelle Position auf das Zeichen hinter dem gefundenen Zeichen.

`virtual String* nextBrace( String* before, const char* braces )`
: Sucht die nächste komplette Klammerung ab der aktuellen Position. Wird in `before` ein String übergeben, so wird hier der Text zwischen aktueller Position und Klammerbeginn gesichert. Die Zeichenkette `braces` gibt die öffnende und schließende Klammer an. Der Text innerhalb der Klammer wird zurückgegeben (bei fehlender öffnender Klammer wird `NULL`, bei fehlender schließender Klammer wird der String bis zum Ende zurückgegeben).

Zusätzlich werden alle Methoden von `String` überschrieben, die die Zeichenkette ändern können, da hierbei geprüft werden muß, ob die Position `pos` innerhalb der Zeichenkette noch zulässig ist. Im einzelnen sind dies die folgenden Methoden: `resize()`, `set()` (in allen Varianten), `vset()`, `format()`, `subst()`, `replace()`, `cutAfter()` und `cutTrailing()`. Der Einfachheit halber wurden die Parameterlisten hier weggelassen.

7.1.4 Sprachen verwalten: `Language` (in `std/Language.cpp`)

Zur Verwaltung mehrerer Sprachen wurden bereits im alten System Sprachcodes eingeführt (`D-` und `E-`). In der Klasse `Language` werden die Codes nun einer Sprache zugeordnet. Zur Zeit existieren die Werte `Language::DEUTSCH`, `Language::ENGLISH` und `Language::NONE` (für sprachunabhängige Texte). Der Wert `Language::LANGUAGES` gibt die Anzahl der Sprachen an (ohne `Language::NONE`). Da die unten angeführte Methode statisch definiert ist, braucht die Klasse `Language` nicht instanziiert zu werden.

`protected`-Methoden:

> `static int getLang( const char** text, int ignore )`
> Liefert anhand einer übergebenen Zeichenkette `text` die passende Sprache zurück. `text` ist ein doppelter Zeiger, da bei `ignore == 1` der einleitende Sprachcode der Zeichenkette übergangen wird, d.h. `*text` zeigt auf das erste Zeichen nach dem Sprachcode.

7.1.5 Mehrsprachige Strings: `LanguageString` (in `std/LanguageString.cpp`)

Um mehrsprachige Zeichenketten zu verwalten, wurde `LanguageString` implementiert. Ein Objekt dieser Klasse speichert zu jeder Sprache ein `String`-Objekt. `LanguageString` implementiert alle Methoden von `String`, allerdings ist in den Parameterlisten ein neues Argument an erster Stelle: `int lang`. Dies ist die gewünschte Sprache aus der Klasse `Language` (bsp. `Language::DEUTSCH`). Zusätzlich zu diesen Methoden existieren noch einige mehr:

`public`-Methoden:

> `LanguageString()`
> Erzeugt einen leeren Sprachstring.
>
> `virtual  LanguageString()`
> Destruktor, löscht alle Zeichenketten.
>
> `void set ( const char** multi )`
> Setzt die Strings für alle Sprachen, dabei muß `multi` ein Array von Strings sein, die Anzahl der Strings muß der Anzahl der Sprachen entsprechen (d.h. `Language::LANGUAGES`).
>
> `void set( LanguageString& lstr )`
> `void set( LanguageString* lstr )`
> Setzt den Sprachstring anhand eines anderen Sprachstrings `lstr`.

`protected`-Methoden:

> `int valid_lang( int lang )`
> Prüft, ob die Sprache zulässig ist.

7.1.6 Nachrichten ausgeben: `Message` (in `admin/Message.cpp`)

Eine einfache Klasse zur Ausgabe von Fehlermeldungen wurde in `Message` implementiert. Die Klasse besitzt nur statische Methoden, braucht also nicht als Instanz erzeugt zu werden.

`public`-Methoden:

> `static void error( const char* format, ... )`
> Ruft `Class::cleanupAllClasses()` auf, gibt die Fehlermeldung aus und bricht das Programm ab.
>
> `static void warn( const char* format, ... )`
> Gibt die Fehlermeldung aus, ohne das Programm zu beenden.

7.1.7 Systemaufrufe: `System` (in `admin/System.cpp`)

Um bestimmte Systemaufrufe zu vereinfachen, ist die Klasse `System` entstanden. Hier kann man externe Programme aufrufen und in C++ nicht implementierte Systemaufrufe starten. Da auch hier alle Methoden statisch definiert sind, muß auch von `System` keine Instanz erzeugt werden.

`public`-Methoden:

`static int command( int abort, const char* prefix, const char* format, ... )`
: Ruft ein externes Programm auf. `abort` gibt an, ob das laufende Programm abbrechen soll, falls das externe Programm nicht gestartet werden kann oder einen Fehlercode zurückliefert. Tritt ein Fehler auf, wird der Text aus `prefix` vor die Fehlermeldung geschrieben. Das externe Programm mit Argumenten wird am Ende in einer `printf`-kompatiblen Form übergeben. `command()` liefert den Fehlercode des Programms zurück.

`static int changedir( const char* dir, int mkdir=0 )`
: Wechselt in das übergebene Verzeichnis `dir`. Ist dies noch nicht vorhanden, kann mittels dem optionalen Parameter `mkdir == 1` das Verzeichnis erzeugt werden. `changedir()` liefert `1` zurück, falls der Wechsel (und u.U. das Anlegen) erfolgreich war.

`static void copyFile( const char* file, const char* srcdir, const char* dstdir )`
: Kopiert eine Datei des Namens `file` vom Verzeichnis `srcdir` in das Verzeichnis `dstdir`.

7.1.8 Speicherschutz: `Observer` (in `admin/Observer.cpp`)

Das verwendete System zum Auffinden von nicht freigegebenem Speicher und ungültigen Zugriffen wird in Anhang B genauer besprochen.

7.2 Administration: `Class` und `Administration`

Die gesamte Verwaltung der Klassenhierarchie ist in der Klasse `Class` implementiert, dazu werden weitere Hilfsklassen eingeführt.

7.2.1 Klassendaten: `ClassData` (in `admin/Class.cpp`)

Die Klasse `ClassData` kann alle Daten speichern, die bei der Registrierung geliefert werden. Der Zugriff wird mit Hilfe von Methoden lesend auf alle Komponenten gewährt (siehe Abschnitt 4.2).

`public`-Methoden:

`ClassData( const char* name, const char* parentname,`
`Class* anew(ClassInit*), void cleanup(void)`
`int hook(int, va_list) )`
: Erzeugt ein neues Objekt der Klasse `ClassData` und trägt alle Informationen ein, die bei der Registrierung übergeben wurden (siehe Abschnitt 4.2).

`const char* getName()`
: Liefert den Namen der Klasse, deren Informationen im Objekt gesichert sind.

`const char* getParent()`
: Liefert den Namen der Vaterklasse der Klasse, deren Informationen im Objekt gesichert sind.

`Class* create( ClassInit* init )`
: Erzeugt eine neue Instanz der Klasse, deren Informationen im Objekt gesichert sind. Dabei muß ein Initialisierungsobjekt (oder `NULL`) übergeben werden.

`void cleanup()`
: Ruft die Methode der eingetragenen Klasse auf, die für das Aufräumen zuständig ist.

`int hasHook()`
: Liefert als Ergebnis, ob eine Methode `hook()` bei der Registrierung übergeben wurde.

`int hook( int type, ... )`
: Ruft die `hook()`-Methode auf, die beim Registrieren der Klasse angegeben wurde.

7.2.2 Initialisierung: `ClassInit` (in `admin/Class.cpp`)

Die Klasse `ClassInit` bildet die Basisklasse für alle Initialisierungsobjekte. Diese werden benutzt, um beim Erzeugen einer Klasse anhand ihres Namens die nötigen Argumente an den Konstruktor weiterzuleiten. Die Basisklasse nimmt keine Argumente auf, sondern definiert nur die Schnittstelle für abgeleitete Klassen.

`public`-Methoden:

`virtual const char* getName()`
: Liefert den Namen der Initialisierungsklasse, hier wird also `"ClassInit"` zurückgegeben.

`virtual int instanceOf( const char* name )`
: Prüft, ob das Initialisierungobjekt zum übergebenen Klassennamen `classname` paßt. Dazu wird getestet, ob `classname` mit dem Klassennamen `"ClassInit"` übereinstimmt.

7.2.3 Klassen verwalten: `Class` (in `admin/Class.cpp`)

Die Klasse `Class` verwaltet eine Liste von `ClassData`-Objekten. Jede Klasse trägt sich selbständig in die Liste ein, Informationen über Klassen können hier angefordert werden.

Ein Klassenname wird dabei immer über ein Prefix/Name-Paar angefordert (Beispiel: MD und Linear).

public-Methoden:

static ClassData* getClassData(const char* prefix, const char* name)
: Liefert das zum Klassennamen prefix/name gehörige Datenobjekt vom Typ ClassData.

static Class* newObject(const char* prefix, const char* name, ClassInit* init)
: Erzeugt anhand des Klassennamens prefix/name eine neue Instanz der Klasse. Dabei wird zur Initialisierung das Objekt init benutzt.

static void cleanupClass(const char* prefix, const char* name)
: Ruft für die übergebene Klasse prefix/name die Methode zum Aufräumen auf.

static void cleanupAllClasses()
: Ruft für alle registrierten Klassen die Methode zum Aufräumen auf.

static List<ClassData*> getChildren(const char* prefix, const char* name)
: Liefert eine Liste aller direkten Nachfolger der Klasse prefix/name.

static int isChildOf(const char* prefix, const char* name, const char* root_prefix, const char* root_name)
: Gibt zurück, ob die Klasse prefix/name im Teilbaum mit der Wurzel root_prefix/root_name enthalten ist. Sind beide Klassen identisch, wird ebenfalls ein Erfolg zurückgegeben.

virtual const char* getObjectName()
: Definiert die Schnittstelle für die Abfrage des eigenen Klassennamens. Hier wird statisch "Class" zurückgegeben.

virtual const char* getParentName() = 0
: Definiert die Schnittstelle, über die man den Namen der Vaterklasse abfragen kann.

protected-Methoden:

static const char* registerClass(const char* name, const char* parent Class* func(ClassInit*), void cleanup(void), int hook(int, va_list))
: Fügt ein weiteres ClassData-Objekt in die Liste ein. Die Argumente beschreiben die Klasse name, eine genaue Beschreibung aller Argumente ist in Abschnitt 4.2 nachzulesen.

7.2.4 Administration: Administration (in admin/Administration.cpp)

Die Klasse Administration übernimmt einige Verwaltungsaufgaben, die nicht mit der Klassenhierarchie zusammenhängen und deshalb nicht von Class implementiert werden. Diese Klasse macht das Arbeiten mit den Zeitreihenobjekten handlicher.

`public`-Methoden:

`Administration( PSBase* persist, const char* project )`
: Erzeugt eine neue Instanz der Klasse `Administration`. Dabei wird sofort das persistente Objekt `persist` aktiviert und das Projekt `project` als primäres Projekt eingetragen.

`virtual ~Administration()`
: Löscht das `Administration`-Objekt und führt dabei die `cleanup()`-Methode aus.

`static void markUnlink( String* name )`
: Fügt die Zeitreihe mit dem Namen `name` in eine Liste zu löschender Zeitreihen ein.

`static void cleanup()`
: Bei einem Aufruf von `cleanup()` wird für jeden mit `markUnlink()` eingetragenen Namen die Methode `TSBase::destroy()` aufgerufen und die Zeitreihe damit gelöscht.

`void changePersistent( PSBase* newpersist )`
: Aktiviert ein neues persistentes Objekt `newpersist`, das bisher aktive persistente Objekt wird dabei freigegeben.

`TSBase* scale( TSBase* zr, List<TSBase*>* add, const char* desc )`
: Skaliert eine Zeitreihe `zr`, dabei wird in `desc` die Art der Vorverarbeitung mit allen Argumenten angegeben, eine komplette Vorverarbeitungspipeline ist ebenfalls möglich. In `add` können weitere eingehende Zeitreihen übergeben werden (oder `NULL`). Liefert einen Zeiger auf die skalierte Zeitreihe zurück.

`TSBase* prognose( const char* name, TSBase* zr, List<TSBase*>* add, const char* data )`
: Startet eine Prognose mit dem Verfahren `name`, dabei wird die Zeitreihe `zr` prognostiziert, eingehende Zeitreihen werden in `add` übergeben und die Parameter stehen in `desc`. Liefert die prognostizierte Zeitreihe zurück.

7.3 Zeitreihen: `TSHead` und Subklassen

Hier werden `TSHead` und alle Subklassen aufgeführt.

7.3.1 Einzelne Kopfinformation: `TSHeadInfo` (in `series/TSHead.cpp`)

Die Klasse `TSHeadInfo` speichert ein Schlüssel-Wert-Kommentar Paar in drei Variablen vom Typ `String`.

`public`-Methoden:

`TSHeadInfo( const char* key, const char* value, const char* comment )`
: Erzeugt eine neue Kopfinformation mit den übergebenen Werten.

`TSHeadInfo( TSHeadInfo& source )`
: Erzeugt eine Kopie einer Kopfinformation `source`.

`virtual ~TSHeadInfo()`
Destruktor: Gibt die drei Strings wieder frei.

`const char* getKey()`
Liefert den Schlüssel der Kopfinformation.

`const char* getValue()`
Liefert den Wert der Kopfinformation.

`const char* getComment()`
Liefert den Kommentar der Kopfinformation.

`void setValue( const char* value, const char* comment )`
Setzt den Wert und den Kommentar der Kopfinformation auf die übergebenen Werte.

7.3.2 Initialisierung: `TSHeadInit` (in `series/TSHead.cpp`)

Siehe auch Abschnitt 4.3.1

`public`-Methoden:

`TSHeadInit( TSHead* source, double def, int autoexpand=1 )`
Erzeugt ein neues Initialisierungsobjekt für Zeitreihenköpfe. Dabei wird ein existierender Kopf übergeben, ein Standardwert für nicht gesetzte Werte und die Angabe, ob die Zeitreihe selbständig wachsen soll.

`int getMode()`
Liefert den Modus des Objekts. Möglich sind hier `TSINIT_TSHEAD` und `TSINIT_TSBASE`. Diese Klasse setzt den Modus immer auf `TSINIT_TSHEAD`. Siehe auch Abschnitt 7.3.4.

`TSHead* getHeadPtr()`
Liefert einen Zeiger auf den Zeitreihenkopf, der kopiert werden soll.

`double getDefault()`
Liefert den Standardwert für nicht gesetzte Werte oder Werte außerhalb des Definitionsbereiches.

`int getAutoexpand()`
Liefert die Angabe, ob die Zeitreihe selbständig wachsen soll.

`protected`-Methoden:

`virtual const char* getName()`
Liefert den Namen des Initialisierungsobjekts, hier also `"TSHeadInit"`.

`virtual int instanceOf( const char* classname )`
Prüft, ob das Initialisierungobjekt zum übergebenen Klassennamen `classname` paßt. Dazu wird getestet, ob `classname` mit dem Klassennamen `"TSHeadInit"` übereinstimmt oder der Test mit Hilfe der Methode `instanceOf()` in der Vaterklasse `ClassInit` Erfolg hatte.

7.3.3 Zeitreihenkopf: `TSHead` (in `series/TSHead.cpp`)

Siehe auch Abschnitt 4.3.1

`public`-Methoden:

`static ProjectManager* getProjectManager()`
: Liefert einen Zeiger auf den Projekt-Manager.

`static void setPrimaryProject( const char* projectname, ClassInit* init )`
: Setzt im Projekt-Manager das primäre Projekt auf den Namen `projectname`. Ist ein solches Projekt noch nicht eingetragen, so wird ein neues Objekt erzeugt, dabei wird das Initialisierungsobjekt `init` benutzt.

`static TSHead* getTSClass( TSHead* source, TSHeadInit* init, const char* root = "Base" )`
: Liefert die für den Zeitreihenkopf `source` zuständige Zeitreihenklasse. Dabei wird mit der Suche bei der Klasse `root` begonnen (mit vorangestelltem `TS`, Standard ist also `TSBase`). Das erzeugte Zeitreihenobjekt wird mit dem Initialisierungsobjekt `init` erzeugt.

`static int checkTSClass( TSHead* check )`
: Prüft, ob diese Klasse zum übergebenen Zeitreihenkopf `check` paßt.

`static void cleanup()`
: Methode zum Aufräumen der Klasse: Löscht den Projekt-Manager.

`TSHead()`
: Erzeugt einen leeren Zeitreihenkopf.

`TSHead( TSHeadInit* init )`
: Erzeugt einen Zeitreihenkopf, dabei wird das übergebene Initialisierungsobjekt benutzt.

`TSHead( TSHead& source )`
: Erzeugt eine Kopie des übergebenen Zeitreihenkopfes.

`virtual ~TSHead()`
: Destruktor: Gibt alle Objektvariablen des Kopfes frei.

`long getStart()`
: Liefert das logische Datum des Startzeitpunktes der Zeitreihe.

`long getEnd()`
: Liefert das logische Datum des Endzeitpunktes der Zeitreihe.

`int getFactor()`
: Liefert den Faktor für die Zeiteinheiten der Zeitreihe.

`int getType()`
: Liefert den Typ der Zeitreihe, der Typ ist projektabhängig.

`int getAggregType()`
: Liefert den Typ der Aggregation der Zeitreihe (0=Mittelwert, 1=Summe, 2=Maximum, 3=Minimum).

`int getPrognoseType()`
: Liefert 1, falls die Zeitreihe aus einer Prognose entstanden ist, ansonsten 0.

`int getIndexType()`
Liefert den Typ des Indexeintrages der Zeitreihe.

`const char* getTypeString( int lang )`
Liefert den Typnamen der Zeitreihe, dieser wird vom Projekt definiert.

`const char* getTime()`
Liefert den Namen der zugrundeliegenden Zeiteinheit.

`int getComponentNo()`
Liefert die Anzahl der Komponenten.

`const char* getComponent( int nr )`
Liefert die `nr`-te Komponente des Zeitreihenkopfes, das Projekt entspricht immer der ersten Komponente.

`const char* getComponentComment( int nr )`
Liefert den `nr`-ten Komponentenkommentar des Zeitreihenkopfes, das Projekt entspricht immer der ersten Komponente.

`void getComponents( String* dest, int full=0, const char* trenner="|" )`
Liefert alle Komponenten in einem String `dest`. Die Komponenten werden durch den String `trenner` getrennt, Standard ist hier das Pipe-Symbol (`|`). Mit `full==1` wird hinter die Komponenten auch noch der Name der Zeitreihe angehängt.

`const char* getScalingRoot()`
Liefert den Namen der Basiszeitreihe. Dies ist die Zeitreihe, die als Input für die Verarbeitungspipeline benutzt wurde.

`const char* getScaling()`
Liefert die Verarbeitungspipeline, die auf die Basiszeitreihe angewandt wurde. Dabei sind die Vorverarbeitungen so angegeben, daß aus den Parametern eine Rückrichtung möglich ist (normalerweise sind dies mehr Informationen, als beim Aufruf der Vorverarbeitung).

`const char* getName()`
Liefert den Namen der Zeitreihe.

`void setCompNo( int nr )`
Setzt die Anzahl der Komponenten der Zeitreihe auf `nr`.

`void setComp( int nr, const char* value )`
Setzt die `nr`-te Komponente der Zeitreihe auf `value`.

`void setCompComment( int nr, int lang, const char* comment )`
Setzt den `nr`-te Komponentenkommentar der Zeitreihe in der Sprache `lang` auf `comment`.

`void setComponents( const char* name )`
Setzt alle Komponenten anhand des kompletten Namens der Zeitreihe. Dabei sind die Komponenten durch das Pipe-Symbol (`|`) getrennt.

`void setTime( const char* timename )`
Setzt den Namen der zugrundeliegenden Zeiteinheit der Zeitreihe. Der Aufruf der Methode rechnet die Zeitreihenwerte jedoch nicht auf die Zeiteinheit um.

`void setType( const char* typename )`
Setzt den Typ der Zeitreihe.

`void noScaling()`
Löscht alle eingetragenen Vorverarbeitungen sowie den Namen der Basiszeitreihe.

`void addScaling( const char* value )`
Hängt eine Vorverarbeitung an die Liste der bereits angewandten Vorverarbeitungen an. Ist dies die erste Vorverarbeitung, so wird auch die Basiszeitreihe eingetragen.

`int removeLastScaling( String* scale )`
Löscht die letzte Vorverarbeitung aus der Liste aller angewandten Vorverarbeitungen und liefert diese im String `scale` zurück (bei `scale != NULL`).

`void setName( const char* value )`
Setzt den Namen der Zeitreihe auf den Wert `value`.

`void setDesc( int lang, const char* value )`
Setzt die Beschreibung der Zeitreihe in der Sprache `lang` auf den Wert `value`.

`void appendDesc( int lang, const char* value )`
Hängt den Wert `value` an die Beschreibung der Zeitreihe in der Sprache `lang` an.

`const char* getDescription( int lang )`
Liefert die Beschreibung der Zeitreihe in der Sprache `lang`.

`String* getIndexName()`
Liefert den `INDEX`-Eintrag für diesen Zeitreihenkopf.

`const char* getIndexHeading()`
Liefert die Überschrift für die `INDEX`-Datei, die diese Zeitreihe enthält.

`int compareKeys( const char* key1, const char* key2 )`
Vergleicht zwei Schlüssel `key1` und `key2`. Liefert 1 bei Übereinstimmung, 0 sonst.

`void setInfo( const char* key, const char* value, const char* comment, int list=TSHEAD_STANDARD )`

`void setInfo( const char* key, long value, const char* comment, int list=TSHEAD_STANDARD )`

`void setInfo( const char* key, double value, const char* comment, int list=TSHEAD_STANDARD )`
Setzt die Kopfinformation mit dem Schlüssel `key` auf den Wert `value` und den Kommentar `comment`. `value` kann dabei eine Zeichenkette, eine Ganzzahl oder eine Fließkommazahl sein. Der optionale Parameter `list` gibt an, ob die Information in die Standardliste der Informationen (`TSHEAD_STANDARD`) oder in die `addon`-Liste (`TSHEAD_ADDON`) eingetragen wird.

`int rmInfo( const char* key, int list=TSHEAD_STANDARD )`
Löscht die Kopfinformation mit dem Schlüssel `key` aus der Liste aller Kopfinformationen (Parameter `list` wie oben).

`const char* getInfo( const char* key, int list=TSHEAD_STANDARD )`
Liefert den Wert der Kopfinformation mit dem Schlüssel `key` (Parameter `list` wie oben).

`const char* getInfoComment( const char* key, int list=TSHEAD_STANDARD )`
Liefert den Kommentar der Kopfinformation mit dem Schlüssel `key` (Parameter `list` wie oben).

`void setObjectData( const char* data )`
: Setzt die Zeichenkette `data` als Objektdatum an die Zeitreihe.

`const char* getObjectData()`
: Liefert das Objektdatum der Zeitreihe.

`protected`-Methoden:

`virtual void recalc()`
: Berechnet alle klassenspezifischen Werte der Zeitreihe neu. In dieser Klasse ohne Funktion.

`const char* createKeyWithLanguage( const char* key, char* newkey )`
: Erzeugt einen korrekten Schlüssel, da für Schlüssel ohne Sprachcode intern immer ein `"--"` vorangestellt wird. Sollte der Schlüssel `key` keinen korrekten Sprachcode besitzen, wird in `newkey` der Schlüssel mit dem internen Code `"--"` abgelegt, ansonsten enthält `newkey` genau den Schlüssel `key`.

`PList<TSHeadInfo*>* getList( int list )`
: Liefert die Liste der Kopfinformationen. Der Parameter `list` gibt dabei die gewünschte Liste an (`TSHEAD_STANDARD` oder `TSHEAD_ADDON`, siehe oben).

`private`-Methoden:

`void init_TSHead()`
: Initialisiert einen Zeitreihenkopf mit Standardwerten, es werden keine Informationen eingetragen.

`void init_TSHead( TSHead& source )`
: Initialisiert den Kopf mit dem übergebenen Initialisierungsobjekt `init`.

7.3.4 Initialisierung: `TSBaseInit` (in `series/TSBase.cpp`)

Siehe auch Abschnitt 4.3.2

`public`-Methoden:

`TSBaseInit( TSBase* source )`
: Diese Klasse setzt den Modus immer auf `TSINIT_TSBASE`. Siehe auch Abschnitt 7.3.2.

`TSBase* getBasePtr()`
: Liefert einen Zeiger auf das Zeitreihenobjekt, das kopiert werden soll.

`virtual const char* getName()`
: Liefert den Namen des Initialisierungsobjekts, hier also `"TSBaseInit"`.

`virtual int instanceOf( const char* classname )`
: Prüft, ob das Initialisierungobjekt zum übergebenen Klassennamen `classname` paßt. Dazu wird getestet, ob `classname` mit dem Klassennamen `"TSBaseInit"` übereinstimmt oder der Test mit Hilfe der Methode `instanceOf()` in der Vaterklasse `TSHeadInit` Erfolg hatte.

7.3.5 Allgemeine Zeitreihe: `TSBase` (in `series/TSBase.cpp`)

Siehe auch Abschnitt 4.3.2

`public`-Methoden:

`static void setPersistentHandler( PSBase* handler )`
: Aktiviert das übergebene persistente Objekt. Bei allen nachfolgenden Aktionen, die vom persistenten Objekt abhängen (z.B. `load()` und `save()`), wird dieses Objekt benutzt. Das alte Objekt wird nicht automatisch freigegeben.

`static int hook( int method, va_list args )`
: Die Methode führt bei Aufruf mit `method==ADD_CHECK_CLASS` die Methode `checkTSClass()` aus (siehe Abschnitt 4.2).

`static int checkTSClass( TSHead* head )`
: Prüft, ob diese Klasse zum übergebenen Zeitreihenkopf `head` paßt.

`static int exists( const char* name, TSBase* relativeTo=NULL )`
: Prüft, ob die angegebene Zeitreihe existiert. Wenn der Name `name` der zu prüfenden Zeitreihe nicht vollständig ist (z.B. bei der Angabe von `|Wochen.ABV`), so werden die Komponenten der Zeitreihe `relativeTo` vorangestellt.

`static int destroy( const char* name, TSBase* relativeTo=NULL )`
: Löscht die angegebene gespeicherte Zeitreihe. Wenn der Name `name` der zu löschenden Zeitreihe nicht vollständig ist (z.B. bei der Angabe von `|Wochen.ABV`), so werden die Komponenten der Zeitreihe `relativeTo` vorangestellt.

`static void descale( TSBase** zr, int all=1 )`
: Führt eine Rückskalierung der Zeitreihe `zr` durch. Wenn der Parameter `all` gleich 0 ist, so wird nur die letzte Vorverarbeitung rückgängig gemacht, ansonsten wird die gesamte Verarbeitungspipeline rückwärts abgearbeitet.

`static void cleanup()`
: Methode zum Aufräumen, löscht das persistente Objekt.

`static TSBase* load( const char* name )`
: Lädt die Zeitreihe des Namens `name`.

`TSBase( TSHeadInit* init )`
: Erzeugt eine neues Zeitreihenobjekt mit Hilfe des übergebenen Initialisierungsobjekts.

`TSBase( long starttick, long endtick, double def, int autoexpand=1 )`
: Erzeugt eine neues Zeitreihenobjekt im Bereich $[starttick, endtick]$, die Werte werden mit dem Standardwert `def` initialisiert. Der Parameter `autoexpand` gibt an, ob die Zeitreihe von selbst wächst (siehe Beispiel 4.2).

`TSBase( TSHead& source, double def, int autoexpand=1 )`
: Erzeugt eine neues Zeitreihenobjekt, der übergebene Kopf wird kopiert, die Werte werden auf den Standardwert `def` gesetzt (Parameter `autoexec` wie oben).

`TSBase( TSBase& source )`
: Erzeugt eine neues Zeitreihenobjekt, die übergebene Zeitreihe `source` wird komplett kopiert.

`int hasMissings()`
Liefert 1 zurück, falls in der Zeitreihe ein Missing-Wert auftritt, 0 sonst.

`double getValue( long tick )`
Liefert den Wert für das logische Datum `tick` zurück.

`void setValue( long tick, double value )`
Setzt den Wert für das logische Datum `tick` auf den Wert `value`.

`void setAllValues( TSBase& source )`
Setzt alle Werte der Zeitreihe auf die Werte der übergebenen Zeitreihe `source`.

`int isMiss( double value )`
Liefert 1 zurück, wenn der Wert für das logische Datum `tick` ein Missing-Wert ist, 0 sonst.

`virtual double aggregate( double values[], int length )`
Aggregiert die ersten `length` Werte des Wertearrays `values` zu einem einzigen und liefert diesen zurück. Die Aggregierung ist abhängig vom Aggregierungstyp (siehe Abschnitt 7.3.3).

`void changeTime( const char* timename, int factor )`
Aggregiert auf die Zeiteinheit `timename` mit dem Faktor `factor`. Wird keine passende Aggregierung gefunden, so bricht das Programm ab.

`void changeInterval( long starttick, long endtick )`
Setzt den Start- und Endzeitpunkt neu. Dabei werden bisher nicht im Definitionsbereich liegende Werte auf den Standardwert gesetzt.

`void shiftInterval( long ticks )`
Verschiebt den Start- und Endzeitpunkt um `ticks` Einheiten nach hinten.

`int convertTo( const char* from, const char* to, TSBase** tozr )`
Konvertiert die Zeitreihe `tozr` von der Klasse `from` in die Klasse `to`. Nach Ende der Methode ist das Zeitreihenobjekt vom Typ `to`.

`int save( const char* name=NULL, int toindex=0 )`
Speichert das Zeitreihenobjekt mit Hilfe des eingetragenen persistenten Objekts. Der Name und die Komponenten der Zeitreihe werden vor dem Speichern angepaßt. Bei Angabe von `"-"` wird auf die Standardausgabe geschrieben, ansonsten kann der Name aus dem Namen der Zeitreihe mit oder ohne Komponenten bestehen. Ohne Komponenten werden die in der Zeitreihe eingetragenen Komponenten benutzt. Eine Angabe von `NULL` ist ebenfalls möglich, dabei werden Komponenten und Namen aus der Zeitreihe benutzt.
Beispiel: `Tegut|AAA|229104|Wochen.ABV` oder `|Wochen.ABV`.

`protected`-Methoden:

`virtual void recalc()`
Berechnet die Anzahl der Missing-Werte in der Zeitreihe. Diese Methode wird beim Aufruf der Methode `hasMissings()` und nach dem Laden einer Zeitreihe aufgerufen.

`virtual void afterLoad()`
Die Methode wird nach dem Laden aufgerufen. Wird benutzt, um für die Klasse relevante Daten aus den Kopfinformationen zu suchen. Die Methode hat in dieser Klasse allerdings keine Funktion.

`virtual void beforeSave()`
: Die Methode erzeugt eine leere `addon`-Liste, in die klasseninterne Informationen eingetragen werden können. Die `addon`-Liste wird beim Speichern abgearbeitet und nach allen Kopfinformationen der Standardliste ebenfalls in den Kopf geschrieben. Wird diese Methode in Subklassen von `TSBase` überschrieben, so muß in der überschriebenen Methode unbedingt diese Methode mit `TSBase::beforeSave()` aufgerufen werden.

`virtual void afterSave()`
: Die Methode löscht die `addon`-Liste wieder. Wird diese Methode in Subklassen von `TSBase` überschrieben, so muß dort unbedingt diese Methode mit `TSBase::afterSave()` aufgerufen werden.

`private`-Methoden:

`void init_TSBase( TSHead& source, double def, int autoexpand=1 )`
: Initialisiert das Zeitreihenobjekt mit dem übergebenen Kopf `source`. Der Standardwert wird auf `def` gesetzt, der Wert `autoexpand` gibt an, ob die Zeitreihe selbständig wächst.

`void init_TSBase( TSBase& source )`
: Initialisiert das Zeitreihenobjekt mit der übergebenen Zeitreihe, d.h., es werden Kopfinformationen und Werte kopiert.

7.3.6 Binäre Zeitreihe: `TSBinary` (in `series/TSBinary.cpp`)

Siehe auch Abschnitt 4.3.3

`public`-Methoden:

`static int hook( int method, va_list args )`
: Die Methode führt bei Aufruf mit `method==ADD_CHECK_CLASS` die Methode `checkTSClass()` aus (siehe Abschnitt 4.2).

`static int checkTSClass( TSHead* head )`
: Prüft, ob diese Klasse zum übergebenen Zeitreihenkopf `head` paßt. Dazu wird nachgesehen, ob die Information `Binaer:1` im Kopf enthalten ist.

`TSBinary( TSHeadInit* init )`
`TSBinary( long starttick, long endtick, double def, int autoexpand=1 )`
`TSBinary( TSHead& source, double def, int autoexpand=1 )`
`TSBinary( TSBinary& source )`
: Konstruktoren wie bei `TSBase`. Zusätzlich wird in den Kopf der Zeitreihe die Information `Binaer:1` geschrieben und die Anzahl der `TRUE`- und `FALSE`-Werte initialisiert.

`virtual double aggregate( double values[], int length )`
: Aggregiert die ersten `length` Werte des Wertearrays `values` zu einem einzigen und liefert diesen zurück. Die Aggregierung ist abhängig vom Aggregierungstyp (siehe Abschnitt 7.3.3). Der Unterschied zur Aggregierung in `TSBase` ist, daß hier nur noch mit boolschen Operatoren gerechnet wird (Summation entspricht `ODER`-Verknüpfung,

das Minimum entspricht einer NICHT-UND-Verknüpfung, alles andere entspricht UND-Verknüpfungen).

long getTrues()
: Liefert die Anzahl der TRUE-Werte der Zeitreihe.

long getFalses()
: Liefert die Anzahl der FALSE-Werte der Zeitreihe.

protected-Methoden:

virtual void recalc()
: Berechnet die Anzahl der TRUE- und FALSE-Werte neu und ruft TSBase::recalc() auf.

virtual void afterLoad()
: Entfernt die Informationen über die Anzahl der TRUE- und FALSE-Werte aus den Kopfinformationen und sichert diese in den Objektvariablen trues und falses.

virtual void beforeSave()
: Ruft TSBase::beforeSave() auf und schreibt die Anzahl der TRUE- und FALSE-Werte in die addon-Liste.

private-Methoden:

void init_TSBinary(TSHead& source, double def, int autoexpand=1)
: Initialisiert das Zeitreihenobjekt mit dem übergebenen Kopf source. Der Standardwert wird auf def gesetzt, der Wert autoexpand gibt an, ob die Zeitreihe selbständig wächst. Fügt außerdem die Information Binaer:1 in den Kopf ein.

void init_TSBinary(TSBinary& source)
: Initialisiert das Zeitreihenobjekt mit der übergebenen Zeitreihe, d.h., es werden Kopfinformationen und Werte kopiert. Fügt außerdem die Information Binaer:1 in den Kopf ein.

7.3.7 Numerische Zeitreihe: TSNumeric (in series/TSNumeric.cpp)

Siehe auch Abschnitt 4.3.3

public-Methoden:

static int hook(int method, va_list args)
: Die Methode führt bei Aufruf mit method==ADD_CHECK_CLASS die Methode checkTSClass() aus (siehe Abschnitt 4.2).

static int checkTSClass(TSHead* head)
: Prüft, ob diese Klasse zum übergebenen Zeitreihenkopf head paßt. Dazu wird nachgesehen, ob die Information Binaer:0 im Kopf enthalten ist.

`TSNumeric( TSHeadInit* init )`
`TSNumeric( long starttick, long endtick, double def, int autoexpand=1 )`
`TSNumeric( TSHead& source, double def, int autoexpand=1 )`
`TSNumeric( TSNumeric& source )`
: Konstruktoren wie bei `TSBase` weiter oben. Zusätzlich wird in den Kopf der Zeitreihe die Information `Binaer:0` geschrieben und die Werte für Minimum, Maximum, Mittelwert, Varianz und Standardabweichung initialisiert.

`double getMinimum()`
: Liefert das Minimum der Zeitreihenwerte.

`double getMaximum()`
: Liefert das Maximum der Zeitreihenwerte.

`double getMean()`
: Liefert den Mittelwert der Zeitreihenwerte.

`double getVariance()`
: Liefert die Varianz der Zeitreihenwerte.

`double getStdDerivation()`
: Liefert die Standardabweichung der Zeitreihenwerte.

`protected`-Methoden:

`virtual void recalc()`
: Berechnet Minimum, Maximum, Mittelwert, Varianz und Standardabweichung der Zeitreihenwerte und ruft `TSBase::recalc()` auf.

`virtual void afterLoad()`
: Entfernt die Informationen über Minimum, Maximum, Mittelwert, Varianz und Standardabweichung aus den Kopfinformationen und sichert diese in den Objektvariablen `minim`, `maxim`, `mean`, `variance` und `derivation`.

`virtual void beforeSave()`
: Ruft `TSBase::beforeSave()` auf und schreibt Minimum, Maximum, Mittelwert, Varianz und Standardabweichung in die `addon`-Liste.

`private`-Methoden:

`void init_TSNumeric( TSHead& source, double def, int autoexpand=1 )`
: Initialisiert das Zeitreihenobjekt mit dem übergebenen Kopf `source`. Der Standardwert wird auf `def` gesetzt, der Wert `autoexpand` gibt an, ob die Zeitreihe selbständig wächst. Fügt außerdem die Information `Binaer:0` in den Kopf ein.

`void init_TSNumeric( TSNumeric& source )`
: Initialisiert das Zeitreihenobjekt mit der übergebenen Zeitreihe, d.h., es werden Kopfinformationen und Werte kopiert. Fügt außerdem die Information `Binaer:0` in den Kopf ein.

7.4 Persistenz: PSBase und Subklassen

Hier werden `PSBase` und alle Subklassen aufgeführt. Diese Klassen sind zuständig für die persistente Speicherung von Zeitreihenobjekten

7.4.1 Basisklasse: PSBase (in persist/PSBase.cpp)

Siehe auch Abschnitt 4.4.1

`public`-Methoden:

`virtual TSBase* load( const char* name ) = 0`
: Dieser Methodenkopf definiert die Schnittstelle zum Laden einer Zeitreihe. Der Name der zu ladenden Zeitreihe wird in `name` übergeben, die geladene Zeitreihe wird als Ergebnis geliefert.

`virtual int save( TSBase* zr, const char* name, int toindex=0 ) = 0`
: Dieser Methodenkopf definiert die Schnittstelle zum Speichern einer Zeitreihe. Als Argumente müssen die Zeitreihe `zr`, der Name der Zeitreihe `name` und der Parameter `toindex`, der angibt, ob die Zeitreihe im Index eingetragen werden soll, übergeben werden.

`virtual int destroy( const char* name ) = 0`
: Dieser Methodenkopf definiert die Schnittstelle zum Löschen einer gespeicherten Zeitreihe. Der Name der zu löschenden Zeitreihe wird in `name` übergeben.

`virtual int exists( const char* name ) = 0`
: Dieser Methodenkopf definiert die Schnittstelle, die prüft ob eine Zeitreihe des Namens `name` existiert.

`virtual const char* basedir() = 0`
: Dieser Methodenkopf definiert die Schnittstelle für die Angabe eines Basisverzeichnisses. Von diesem Verzeichnis aus wird abhängig vom Projekt weiterverzweigt. Das resultierende Verzeichnis kann z.B. Log-Dateien des Prognoseverfahrens aufnehmen.

`virtual int getPrognose( TSBase* zr, String* desc, String* data ) = 0`
: Definiert die Schnittstelle, die die Nummer der Prognose liefert, durch die die Zeitreihe `zr` entstanden ist. Dabei stehen in `desc` die Beschreibungen der Parameter, in `data` die eigentlichen Parameterwerte.

`virtual int setPrognose( TSBase* zr, String* desc, String* data ) = 0`
: Definiert die Schnittstelle zum Eintragen einer Prognose in eine Liste von Prognosen. Dabei ist `zr` die prognostizierte Zeitreihe, `desc` und `data` entsprechen den Parameterbeschreibungen und den Parameterwerten.

`protected`-Methoden:

`List<TSHeadInfo*>* getTSInfos( TSHead* head )`
: Liefert die Standardliste aller Kopfinformationen von `head`.

`List<TSHeadInfo*>* getTSAddon( TSHead* head )`
: Liefert die `addon`-Liste mit Kopfinformationen von `head`.

`void setTSType( TSHead* head, const char* type )`
: Setzt den Typ der Zeitreihe `head` auf den Wert `value` (siehe `setType()` in Abschnitt 7.3.3).

`void setTSTime( TSHead* head, const char* time )`
: Setzt die Zeiteinheit der Zeitreihe `head` auf den Wert `value` (siehe `setTime()` in Abschnitt 7.3.3).

`void setTSInterval( TSHead* head, int factor, long start, long end )`
: Setzt Start- und Endzeitpunkt der Zeitreihe `head` auf die Werte `start` und `end`, dabei wird als Faktor `factor` benutzt (siehe `setInterval()` in Abschnitt 7.3.3).

`void setTSCompNo( TSHead* head, int value )`
: Setzt die Anzahl der Komponenten der Zeitreihe `head` auf den Wert `value` (siehe `setCompNo()` in Abschnitt 7.3.3).

`void setTSComp( TSHead* head, int nr, const char* value )`
: Setzt die `nr`-te Komponente der Zeitreihe `head` auf den Wert `value` (siehe `setComp()` in Abschnitt 7.3.3).

`void setTSCompComment( TSHead* head, int nr, int lang, const char* comment )`
: Setzt den `nr`-ten Komponentenkommentar der Zeitreihe `head` auf den Wert `value` (siehe `setCompComment()` in Abschnitt 7.3.3).

`void setTSName( TSHead* head, const char* value )`
: Setzt den Namen der Zeitreihe `head` auf den Wert `name` (siehe `setName()` in Abschnitt 7.3.3).

`void setTSDesc( TSHead* head, int lang, const char* value )`
: Setzt die Beschreibung der Zeitreihe `head` auf den Wert `name` (siehe `setDesc()` in Abschnitt 7.3.3).

`void setTSRoot( TSHead* head, const char* value )`
: Setzt die Basiszeitreihe der Zeitreihe `head` auf den Wert `name`.

`void setTSChanged( TSHead* head, int value )`
: Setzt eine Flagge, ob die Zeitreihe `head` als geändert angesehen wird, auf den Wert `value`. Bei geänderten Zeitreihen (`value != 0`) wird intern `recalc()` aufgerufen, um Informationen der Zeitreihe neu zu berechnen.

`void setTSAggreg( TSHead* head, int value )`
: Setzt den Typ der Aggregierung der Zeitreihe `head` auf den Wert `value` (0=Mittelwert, 1=Summe, 2=Maximum, 3=Minimum).

`void setTSPrognose( TSHead* head, int value )`
: Setzt eine Flagge, die angibt, ob die Zeitreihe `head` aus einer Prognose entstanden ist (`value != 0`).

`void addTSScale( TSHead* head, const char* value )`
: Fügt die Vorverarbeitung `value` zu den im Kopf der Zeitreihe `head` eingetragenen Vorverarbeitungen hinzu.

7.4.2 Initialisierung: `PSFileInit` (in `persist/PSFile.cpp`)

Dies ist das Initialisierungsobjekt für die Klasse `PSFile`. Als Argument kann die Klasse `PSFileInit` das Basisverzeichnis speichern, von dem in die Projekte verzweigt wird.

`public`-Methoden:

`virtual const char* getName()`
: Liefert den Namen des Initialisierungsobjekts, hier also `"PSFileInit"`.

`virtual int instanceOf( const char* classname )`
: Prüft, ob das Initialisierungobjekt zum übergebenen Klassennamen `classname` paßt. Dazu wird getestet, ob `classname` mit dem Klassennamen `"PSFileInit"` übereinstimmt oder der Test mit Hilfe der Methode `instanceOf()` in der Vaterklasse `ClassInit` Erfolg hatte.

`PSFileInit( const char* dir )`
: Speichert den übergebenen Verzeichnisnamen.

`const char* getRoot()`
: Liefert den Verzeichnisnamen.

7.4.3 Datenformat: `PSFile` (in `persist/PSFile.cpp`)

Siehe auch Abschnitt 4.4.2

`public`-Methoden:

`PSFile( PSFileInit* init )`
: Erzeugt ein persistentes Objekt der Klasse `PSFile`, dabei wird das übergebene Initialisierungsobjekt `init` benutzt.

`PSFile( const char* rootdir )`
: Erzeugt ein persistentes Objekt der Klasse `PSFile`, dabei wird das Basisverzeichnis `rootdir` benutzt.

`virtual TSBase* load( const char* name )`
: Lädt die Zeitreihe des Namens `name`. Dabei muß die Zeitreihe in dem während der Projektarbeit entstandenen Datenformat (siehe Kapitel 2) vorliegen.

`virtual int save( TSBase* zr, const char* name, int toindex=0 )`
: Speichert die Zeitreihe `zr` unter dem Namen `name`. Ist `toindex` ungleich 0, so wird die Zeitreihe zusätzlich auch in einer `INDEX`-Datei eingetragen.

`virtual int destroy( const char* name )`
: Löscht die gespeicherte Zeitreihe des Namens `name`, dabei wird ein existierender Eintrag in einer `INDEX`-Datei ebenfalls gelöscht.

`virtual int exists( const char* name )`
: Prüft, ob die Zeitreihe `name` existiert.

`virtual const char* basedir()`
: Liefert das Basisverzeichnis, das bei der Erzeugung des Objekts angegeben wurde.

`virtual int getPrognose( TSBase* zr, String* desc, String* data )`
`virtual int setPrognose( TSBase* zr, String* desc, String* data )`
Diese Methoden wurden nicht implementiert, da der Eintrag in die Prognoseliste von den externen Programmen durchgeführt wird.

`protected`-Methoden:

`void filenames( const char* name, char* head, char* body, char* type="ZR" )`
Erzeugt aus dem Zeitreihennamen `name` die zugehörigen Dateinamen für den Kopf der Zeitreihe (in `head`) und den Rumpf der Zeitreihe (in `body`). In `type` wird eine Zeichenkette übergeben, die als Verzeichnis hinter die Komponenten eingesetzt wird. **Beispiel:** Aus `CO2|Tage.CO2` wird mit `type="ZR"` der Kopf-Dateiname `"CO2/ZR/Head/Tage.CO2"` erzeugt.

`void decode( TSHead* head, const char* line )`
Dekodiert die Kopfzeile `line` und speichert die gelesenen Informationen direkt oder als Schlüssel-Wert-Paar im Zeitreihenkopf `head`.

`const char* getIndexFile( TSBase* zr )`
Liefert den Dateinamen der `INDEX`-Datei, die zu der übergebenen Zeitreihe `zr` gehört.

`const char* parseLine( char* line, int* type, char** key, char** desc )`
Parsiert eine Zeile `line` der `INDEX`-Datei. Dabei wird in `type` der Typ des Eintrages, in `key` der Schlüssel (z.B. der Name der Zeitreihe) und in `desc` die Beschreibung des Eintrages eingetragen.

`int checkIndex( TSBase* zr )`
`int checkIndex( const char* name )`
Prüft, ob die zur Zeitreihe `zr` (bzw. zur Zeitreihe des Namens `name`) gehörende `INDEX`-Datei existiert.

`int setIndex( TSBase* zr, int type, const char* key, const char* desc )`
Trägt die Zeitreihe `zr` in einer `INDEX`-Datei ein. Der Typ des Eintrages wird in `type` übergeben, der Schlüssel und die Beschreibung in `key` und `desc`.

`int rmIndex( const char* name, int substring )`
Löscht einen Eintrag aus einer `INDEX`-Datei. Dabei gibt `name` den Namen einer Zeitreihe an, die ausgetragen werden soll. Der Parameter `substring` entscheidet, ob der Zeitreihenname genau passen muß oder ob alle Einträge entfernt werden, die den Zeitreihennamen als Teilstring enthalten.

`int newIndex( TSBase* zr, const char* head )`
Erzeugt eine neue `INDEX`-Datei für die Zeitreihe `zr`. Dabei wird `head` als Überschrift eingetragen, die `INDEX`-Datei hat keine weiteren Einträge.

`private`-Methoden:

`void init_PSFile( const char* rootdir )`
Initialisiert das persistente Objekt der Klasse `PSFile` mit dem eingetragenen Verzeichnis `rootdir`.

7.4.4 Datenformat: `PSPictex` (in `persist/PSPictex.cpp`)

Siehe auch Abschnitt 4.4.3

`public`-Methoden:

`virtual int save( TSBase* zr, const char* name, int toindex=0 )`
: Speichert die Zeitreihe `zr` als PICTEX-Makro. Dabei muß der Name folgende Form haben: Die erste Komponente gibt das Projekt an, die zweite den Dateinamen der LATEX-Datei (ohne .tex Kürzel).
Beispiel: `Tegut|latex/ausgabe` verwendet das Projekt Tegut und erzeugt eine Datei `latex/ausgabe.tex`.

`virtual const char* basedir()`
: Liefert `"."` als Verzeichnis, es wird also vom aktuellen Verzeichnis weiterverzweigt.

`virtual TSBase* load( const char* name )`
`virtual int destroy( const char* name )`
`virtual int exists( const char* name )`
`virtual int getPrognose( TSBase* zr, String* desc, String* data )`
`virtual int setPrognose( TSBase* zr, String* desc, String* data )`
: Liefert Fehlermeldung, da diese Methoden nicht implementiert sind. `PSPictex` ist eine reine Export-Klasse, d.h., es wird nur das Speichern von Zeitreihen unterstützt.

7.5 Zeiteinheiten: `TMBase` und Subklassen

7.5.1 Initialisierung: `TimeInit` (in `time/TMBase.cpp`)

`public`-Methoden:

`virtual const char* getName()`
: Liefert den Namen des Initialisierungsobjekts, hier also `"TimeInit"`.

`virtual int instanceOf( const char* classname )`
: Prüft, ob das Initialisierungobjekt zum übergebenen Klassennamen `classname` paßt. Dazu wird getestet, ob `classname` mit dem Klassennamen `"TimeInit"` übereinstimmt oder der Test mit Hilfe der Methode `instanceOf()` in der Vaterklasse `ClassInit` Erfolg hatte.

`TimeInit( int type, int factor, long p1=0, long p2=0, long p3=0, long p4=0, long p5=0, long p6=0 )`
: Speichert eine Datumsangabe. Dazu gehören der Typ `type` der Angabe (siehe Tabelle 4.3), der Faktor `factor` und bis zu sechs weitere Argumente (abhängig vom gewählten Typ).

`TimeInit( TMBase* source, int factor=1 )`
: Kopiert das Datum des übergebenen Zeiteinheitenobjekts `source` und rechnet das Datum auf den Faktor `factor` um. Als Datumstyp wird `TMBase::TIME` eingetragen.

`int getType()`
Liefert den Datumstypen.

`int getFactor()`
Liefert den Faktor der Zeiteinheit.

`long getParam( int index )`
Liefert das `index`-te Argument des ersten Konstruktors (`p1` ... `p6`). Nur bei Typen ungleich `TMBase::TIME` definiert.

`TMBase* getPtr()`
Liefert den Zeiger auf die zu kopierende Zeiteinheit. Nur bei einem Typ `TMBase::TIME` definiert.

7.5.2 Allgemeine Zeitrechnung: `TMBase` (in `time/TMBase.cpp`)

Es werden zwei verschiedene Datumsformate gespeichert: Ein logisches Datum, das nur die Einheiten seit dem festgelegten Nullpunkt der Zeitrechnung angibt, und zweimal ein lesbares Datum, das die gebräuchlichen Werte speichert (Datum, Uhrzeit, Woche, etc.). Es werden zwei lesbare Daten gespeichert, davon ist eins ein Originaldatum und wird bei Konvertierungen meist nur lesend benutzt, das andere repräsentiert den Zeitpunkt, der zur Zeiteinheit paßt (z.B. wird einem Jahres-Objekt immer der 1.1. als Tag zugeordnet). Näheres zur Zeitrechnung ist in Abschnitt 4.7 zu lesen.

`public`-Methoden:

`static int gueltig_datum( int tt, int mon, int jjjj, int hh, int mm, int ss )`
Prüft, ob das übergebene Datum korrekt ist.

`static int gueltig_kwoche( int kwoche, int kjahr )`
Prüft, ob die übergebene Kalenderwoche gültig ist.

`static int gueltig_quartal( int quartal, int jahr )`
Prüft, ob die übergebene Quartalsangabe gültig ist.

`TMBase()`
Setzt das Zeiteinheitenobjekt auf das aktuelle Datum.

`TMBase( TimeInit* init )`
Setzt das Zeiteinheitenobjekt auf das im Initialisierungsobjekt `init` gespeicherte Datum.

`TMBase( int type, int factor=1, long p1, int p2, int p3, int p4, int p5, int p6 )`
Setzt das Zeiteinheitenobjekt auf das übergebene Datum (siehe Tabelle 4.3).

`TMBase( int type, const char* format )`
Setzt das Zeiteinheitenobjekt auf den Wert aus der Zeichenkette `format`. Dabei gibt `type` an, in welchem Format der Text vorliegt (siehe Tabelle 4.3).

`TMBase( TMBase& source, int factor=1 )`
Setzt das Zeiteinheitenobjekt auf den Wert des übergebenen Zeiteinheitenobjekts und rechnet das Datum auf den Faktor `factor` um.

`virtual void setAsOriginal()`
Setzt das Originaldatum auf das berechnete Datum.

`virtual void parseTime( const char* format, int type=-1 )`
Setzt das Zeiteinheitenobjekt auf den Wert aus der Zeichenkette `format`. Dabei gibt `type` an, in welchem Format der Text vorliegt (siehe Tabelle 4.3). Der Wert `type==-1` gibt an, daß der Typ zur Laufzeit mit Hilfe der Methode `stdType()` ermittelt werden soll.

`void setTick( int factor, long tick )`
Setzt das Zeiteinheitenobjekt auf das übergebene logische Datum.

`void setTick( const char* format )`
Setzt das Zeiteinheitenobjekt auf das im Text `format` angegebene logische Datum.

`long getValue( int value )`
Liefert einen speziellen Wert des berechneten Datums, der Parameter `value` gibt dabei den gewünschten Wert an (siehe Tabelle 4.4).

`virtual const char* getDesc( int lang ) = 0`
Definiert die Schnittstelle, über den den Namen der Zeiteinheit in der Sprache `lang` ermittelt werden kann.

`const char* getReadable( int type=-1 )`
Liefert das gespeicherte Datum in einer lesbaren Form. Dabei gibt `type` die Art der Ausgabe an, ein Wert von `type==-1` ermittelt zur Laufzeit den gewünschten Typ mit Hilfe der Methode `stdType()`.

`virtual void output( int type=-1 )`
Gibt das gespeicherte Datum in einer lesbaren Form aus, vergleiche dazu `getReadable()`.

`virtual int stdType() = 0`
Definiert die Schnittstelle, mit der die abgeleiteten Klassen ihren Standardtyp (zur Ein-/Ausgabe) festlegen können.

`virtual double* aggregateFrom( const double* values, TSBase* zr, TMBase* from_start, long from_end, TMBase* to_start, long to_end )`

`virtual double* aggregateTo( const double* values, TSBase* zr, TMBase* from_start, long from_end, TMBase* to_start, long to_end )`
Aggregiert die Werte im Array `values`. Die Werte gehören zur Zeitreihe `zr`. Dabei werden Start- und Endzeitpunkt in der Quell- und Zielzeiteinheit übergeben. Das Startdatum ist dabei ein komplettes Zeiteinheitenobjekt, das Enddatum ist einfach ein logisches Datum. Der Unterschied zwischen den beiden Methoden ist der folgende: `aggregateFrom()` aggregiert aus der Zeiteinheit der Klasse in eine andere Einheit, `aggregateTo()` aggregiert in die Zeiteinheit der Klasse hinein.

`protected`-Methoden:

`void init_TMBase()`
Initialisiert das Zeiteinheitenobjekt mit dem aktuellen Datum und der aktuellen Uhrzeit.

`void init_TMBase( int type, int factor, long p1, int p2, int p3, int p4, int p5, int p6 )`
: Initialisiert das Zeiteinheitenobjekt anhand des übergebenen Datums.

`void init_TMBase( TMBase& source, int factor )`
: Initialisiert das Zeiteinheitenobjekt mit Hilfe des übergebenen Zeiteinheitenobjekts `source` und rechnet auf den Faktor `factor` um.

`void init_TMBase_DATUM( int jjjj=AD_JAHR, int mon=1, int tt=1, int hh=0, int mm=0, int ss=0 )`
: Initialisiert das Objekt mit dem übergebenen Datum. Danach werden Kalenderwoche, Quartal und Wochentag berechnet.

`void init_TMBase_KWOCHE( int kjahr=AD_JAHR, int kwoche=1 )`
: Initialisiert das Objekt mit der übergebenen Kalenderwoche. Danach wird das Datum auf den Montag der Woche und die Uhrzeit auf 00:00:00 Uhr gesetzt, das Quartal wird anhand des Montags der Woche berechnet.

`void init_TMBase_QUARTAL( int jjjj=AD_JAHR, int qq=1 )`
: Initialisiert das Objekt mit dem übergebenen Quartal. Danach wird das Datum auf den ersten Tag des Quartals und die Uhrzeit auf 00:00:00 Uhr gesetzt. Der Wochentag wird anhand des ersten Tages des Quartals berechnet.

`virtual void recalc_logical()`
: Wird in Unterklassen benutzt, um das gespeicherte Originaldatum in ein logisches Datum umzurechnen. In dieser Klasse ist die Methode leer.

`virtual void recalc_real()`
: Wird in Unterklassen benutzt, um das gespeicherte logische Datum in ein reales Datum umzurechnen.

7.5.3 Zeitrechnung: Von `TMBase` abgeleitete Klassen (in `time/*.cpp`)

Alle abgeleiteten Zeiteinheiten überschreiben nur einige der Umrechnungs- und Aggregationsmethoden von `TMBase`. Alle Konstruktoren von `TMBase` werden auch in den Subklassen implementiert, allerdings wird hier nur der Konstruktor von `TMBase` aufgerufen und danach das logische oder reale Datum neu berechnet. Die Methode `getDesc()` liefert den Namen der Zeiteinheit in der angegebenen Sprache zurück.

Das reale Datum wird wie folgt aus dem Originaldatum erzeugt:

`TMSekunden`
: Das Originaldatum wird ohne Änderungen übernommen.

`TMMinuten`
: Das Originaldatum wird kopiert und die Sekunden auf 0 gesetzt.

`TMStunden`
: Das Originaldatum wird kopiert und die Sekunden und Minuten auf 0 gesetzt.

`TMTage`
: Das Originaldatum wird kopiert und die Uhrzeit auf 00:00:00 gesetzt.

`TMWochen`
: Das Originaldatum wird übernommen, die Uhrzeit auf 00:00:00 und der Tag wird auf den Montag der Woche gesetzt.

`TMMonate`
: Das Originaldatum wird übernommen, die Uhrzeit auf 00:00:00 und der Tag wird auf den Ersten des Monats gesetzt.

`TMQuartale`
: Das Originaldatum wird übernommen, die Uhrzeit auf 00:00:00 und der Tag wird auf den Ersten Tage des Quartals gesetzt.

`TMJahre`
: Das Originaldatum wird übernommen, die Uhrzeit auf 00:00:00 und der Tag wird auf den 1.1. des Jahres gesetzt.

Jede Klasse muß außerdem die Umrechnung vom logischen zum realen Datum implementieren, dazu kommen einige Methoden der Zeitrechnung zum Tragen [Deu75, Kra]. Die genaue Verfahrensweise ist im Quellcode ersichtlich.

Eine Liste der bereits implementierten Aggregationen (in der Methode `aggregateTo()`) ist in Tabelle 7.2 zu sehen.

Tabelle 7.2: Implementierte Aggregationen auf andere Zeiteinheiten

Klasse	Aggregation in folgende Einheiten
`TMTage`	Wochen, Monate, Quartale, Jahre
`TMMonate`	Quartale, Jahre

7.5.4 Zeitrechnung: `TMTicks` (in `time/TMTicks.cpp`)

Die Zeiteinheit `Ticks` ist anders zu implemetieren, da hier einige Sonderfälle auftreten. Die Einheit Ticks kann weder mit Hilfe anderer Einheiten initialisiert werden, noch können Ticks in andere Einheiten umgerechnet werden. Aus diesem Grund brechen alle Konstruktoren das Programm ab, falls sie nicht mit dem Typ `TMBase::TICKS` aufgerufen wurden. Ebenfalls bricht ein Aufruf von `setAsOriginal()` das Programm ab, da nur ein logisches Datum verwaltet wird.

Alle anderen Methoden werden nicht überschrieben, da hier die Funktionalität der Basisklasse `TMBase` ausreicht.

7.6 Vorverarbeitungen: `MDBase` und Subklassen

7.6.1 Basisklasse: `MDBase` (in `modifiers/MDBase.cpp`)

`public`-Methoden:

`virtual TSBase* scale( TSBase* zr, List<TSBase*>* add, const char* desc ) = 0`
: Definiert die Schnittstelle zum Vorverarbeiten einer Zeitreihe. Dabei soll die Zeitreihe `zr` bearbeitet werden, die Liste `add` enthält weitere eingehende Zeitreihen, der Text `desc` enthält alle Parameter des Verfahrens.

`virtual TSBase* descale( TSBase* zr, const char* desc ) = 0`
: Definiert die Schnittstelle zur Rückskalierung einer Zeitreihe. Dabei wird die Zeitreihe `zr` bearbeitet, die Parameter der Vorverarbeitung sind in `desc` gespeichert.

`protected`-Methoden:

`void addPrefix( TSBase* zr, const char* desc )`
: Hängt die Beschreibung der Vorverarbeitung `desc` an die bereits im Kopf der Zeitreihe `zr` eingetragene Vorverarbeitungspipeline an.

7.6.2 Vorverarbeitung: Von `MDBase` abgeleitete Klassen (in `modifiers/*.cpp`)

Alle von `MDBase` abgeleiteten Klassen überschreiben die Methode `scale()` und `descale()`. Ein Überblick über alle Vorverarbeitungen ist in Abschnitt 4.5.2 zu sehen.

7.7 Prognosen: `PRBase` und Subklassen

7.7.1 Basisklasse: `PRBase` (in `progno/PRBase.cpp`)

`public`-Methoden:

`virtual TSBase* prognose( TSBase* zr, List<TSBase*>* add, const char* data ) = 0`
: Definiert die Schnittstelle zur Prognose einer Zeitreihe. Dabei wird die Zeitreihe `zr` unter dem Einfluß der Zeitreihen in `add` prognostiziert, die Parameter des Verfahrens werden in `data` übergeben.

`private`-Methoden:

`virtual int info( TSBase* zr, void* data )`
: Definiert die Schnittstelle zum Parsieren des an der Zeitreihe hängenden Objektdatums. Dieses kann mit der Methode `getObjectData()` ermittelt werden. Es wird eine klassenabhängige Struktur `data` gefüllt, die Informationen der Struktur werden nur innerhalb der Klasse benutzt.

7.7.2 Skriptgesteuerte Prognosen: `PRInfile` (in `progno/PRInfile.cpp`)

`public`-Methoden:

`virtual TSBase* prognose( TSBase* zr, List<TSBase*>* add, const char* data )`
: Startet die Prognose, dabei können durch die in dieser Klasse definierten Schnittstellen `prognose_*()` Anpassungen in den Subklassen gemacht werden. Genauere Beschreibungen für jede einzelne Methode sind weiter unten verfügbar. Da die prognostizierte Zeitreihe meist unter einem temporären Namen gespeichert wird, muß im Kopf der Prognosezeitreihe der echte Name der Zeitreihe stehen. Dazu wird nach dem Schlüssel `UnskalierterTyp` gesucht.
Beispiel: Die Zeitreihe wird vom Skript immer unter `Wochen.RESULT` gespeichert, der echte Name der Zeitreihe steht aber im Kopf unter `UnskalierterTyp:Wochen.BP_001`.

`protected`-Methoden:

`virtual void prognose_get_parameters( StringParser* sp ) = 0`
: Definiert die Schnittstelle, in der aus dem String `sp` die Parameter der Prognose parsiert werden. Diese Parameter müssen in Objektvariablen gesichert werden.

`virtual void prognose_test_parameters( String* infile ) = 0`
: Definiert die Schnittstelle zum Testen der eingelesenen Parameter. Um die Methode flexibel zu halten, wird in `infile` der Name der Steuerdatei mit übergeben.

`virtual void prognose_write_output( FILE* fp, TSBase* zr ) = 0`
: Definiert die Schnittstelle, mit der eine Zeile in die Steuerdatei mit dem Dateihandle `fp` eingefügt werden kann. Diese Zeile beschreibt die zu prognostizierende Zeitreihe `zr`.

`virtual void prognose_write_input( FILE* fp, TSBase* zr, int cnt ) = 0`
: Definiert die Schnittstelle, mit der eine Zeile in die Steuerdatei mit dem Dateihandle `fp` eingefügt werden kann. Diese Zeile beschreibt die `cnt`-te einfließende Zeitreihe `zr`.

`virtual void prognose_commands( String* infile, TSBase* zr ) = 0`
: Definiert die Schnittstelle zum Start der Prognose. Hier werden meistens externe Programme aufgerufen. Der Name der Steuerdatei wird in `infile` übergeben, die zu prognostizierende Zeitreihe steht in `zr`.

`virtual void prognose_postprocessing( const char* zrfile, const char* zrname ) = 0`
: Definiert die Schnittstelle für beliebige Arten der Nachbearbeitung. Diese Methode wird direkt nach der Prognose aufgerufen und kann weiterführende Arbeiten verrichten, z.B. zusätzlich zu der prognostizierten Zeitreihe enstandene Werte bearbeiten. Als Parameter wird der reine Dateiname der Zeitreihe ohne Komponenten in `zrfile` und der komplette Name der Zeitreihe inklusive Komponenten in `zrname` übergeben.

`virtual void prognose_cleanup( const char* zrname ) = 0`
: Definiert die Schnittstelle für Aufräumarbeiten, die nach der Prognose durchgeführt werden müssen. Hier können beispielsweise temporäre Dateien gelöscht werden, die von externen Programmen erzeugt wurden.

7.7.3 Verfahren BP und SNNS: `PRBP` und `PRSNNS` (in `progno/PRBP.cpp`, `progno/PRSNNS.cpp`)

Die Verfahren sind fast identisch, es werden nur unterschiedliche externe Programme aufgerufen. Als Parameter können ein Kommentar und ein Modus übergeben werden. Der Modus gibt an, ob nur die prognostizierte Zeitreihe oder auch Trainings- und/oder Testwerte als Zeitreihe angelegt werden.

`public`-Methoden:

`PRBP(), PRSNNS()`
: Initialisiert Kommentar und Modus.

`virtual ~PRBP(), virtual ~PRSNNS()`
: Löscht die Variablen für Kommentar und Modus.

`protected`-Methoden:

`virtual void prognose_get_parameters( StringParser* sp )`
: Liest aus dem übergebenen String `sp` die Parameter ein, der erste Parameter entspricht dem Modus, der zweite dem Kommentar.

`virtual void prognose_test_parameters( String* infile )`
: Testet die Parameter auf korrekte Werte. Der Modus kann die Werte `"NONE"`, `"TRAIN"`, `"TEST"` und `"BOTH"` annehmen (entscheidet, ob Trainings- und/oder Testzeitreihen geschrieben werden). Der Kommentar ist beliebig und wird nicht getestet.

`virtual void prognose_write_output( FILE* fp, TSBase* zr )`
: Schreibt die Beschreibung der zu prognostizierenden Zeitreihe `zr` in das Steuerfile mit dem Dateihandle `fp`. Eine Zeile hat dabei folgende Form:
`OUTPUT $KOMP1 $KOMP2 ... $KOMPn OUT`
Die Zeitreihe muß den Namen `Wochen.OUT` haben.

`virtual void prognose_write_input( FILE* fp, TSBase* zr, int cnt )`
: Schreibt die Beschreibung `cnt`-ten eingehenden Zeitreihe `zr` in das Steuerfile mit dem Dateihandle `fp`. Dabei werden zusätzliche Informationen der Zeitreihe (mit `info()`) benutzt. Eine Zeile hat dabei folgende Form:
`INPUT USE|UNKNOWN <Tiefe> $KOMP1 $KOMP2 ... $KOMPn IN<cnt>`
Die Zeitreihe muß den Namen `Wochen.IN<cnt>` haben.

`virtual void prognose_commands( String* infile , TSBase* zr )`
: Erzeugt die Pattern für die neuronalen Netze mit Hilfe des Perl-Skripts `fe2bp.pl`. Der Name der Steuerdatei wird in der Variablen `infile`, die zu prognostizierende Zeitreihe in `zr` übergeben. `start_training` startet das Training und die Abfrage des Neuronale Netzes. Danach wird mit dem Perl-Skript `create_prognose.pl` die Prognosezeitreihe erzeugt. Dabei wird der Dateiname der zu prognostizierenden Zeitreihe benutzt, allerdings wird das Kürzel `OUT` durch `RESULT` ersetzt, die Trainingszeitreihe bekommt das Kürzel `RESULT.TRAIN`, die Testzeitreihe das Kürzel `RESULT.TEST`.

`virtual void prognose_postprocessing( const char* zrfile, const char* zrname )`
Lädt (je nach Modus) die Trainings- und/oder Testzeitreihe, führt die Rückskalierung durch und speichert die entstandenen Zeitreihen.

`virtual void prognose_cleanup( const char* zrname )`
Löscht die von den externen Skripten geschriebenen Trainings- und Testzeitreihen `RESULT.TRAIN` und `RESULT.TEST`.

`private`-Methoden:

`virtual int info( TSBase* zr, void* data )`
Parsiert die Zeichenkette, die als Objektdatum an der Zeitreihe hängt. Diese wird mit `getObjectData()` ermittelt und muß folgende Form haben:
`PRESENCE|FUTURE <Tiefe>` Damit wird die Struktur `data` vom Typ `PRBP_DATA` bzw `PRSNNS_DATA` gefüllt.

7.7.4 Verfahren Gauss: PRGauss (in progno/PRGauss.cpp)

`public`-Methoden:

`PRGauss()`
Initialisiert den Kommentar und das zu verwendende Perl-Skript für die Prognose.

`virtual ~PRGauss()`
Löscht die Variablen für Kommentar und Perl-Skript.

`protected`-Methoden:

`virtual void prognose_get_parameters( StringParser* sp )`
Liest aus dem übergebenen String `sp` die Parameter ein, der erste Parameter entspricht dem Skriptnamen, der zweite dem Kommentar.

`virtual void prognose_test_parameters( String* infile )`
Keine Funktion, da weder Kommentar noch Perl-Skriptname geprüft werden.

`virtual void prognose_write_output( FILE* fp, TSBase* zr )`
Schreibt die Beschreibung der zu prognostizierenden Zeitreihe `zr` in das Steuerfile mit dem Dateihandle `fp`. Eine Zeile hat dabei folgende Form:
`OUTPUT $KOMP1 $KOMP2 ... $KOMPn OUT`
Die Zeitreihe muß den Namen `Wochen.OUT` haben.

`virtual void prognose_write_input( FILE* fp, TSBase*zr , int cnt )`
Schreibt die Beschreibung `cnt`-ten eingehenden Zeitreihe `zr` in das Steuerfile mit dem Dateihandle `fp`. Dabei werden zusätzliche Informationen der Zeitreihe (mit `info()`) benutzt. Eine Zeile hat dabei folgende Form:
`INPUT USE|UNKNOWN <Tiefe> $KOMP1 $KOMP2 ... $KOMPn IN<cnt>`
Die Zeitreihe muß den Namen `Wochen.IN<cnt>` haben.

`virtual void prognose_commands( String* infile, TSBase* zr )`
Startet die Gauss-Prognose mit Hilfe des übergebenen Perl-Skriptes. Zur Zeit sind die Skripte `arima.pl` und `regression.pl` implementiert. Der Name der Steuerdatei wird in der Variablen `infile`, die zu prognostizierende Zeitreihe in `zr` übergeben.

`virtual void prognose_postprocessing( const char* zrfile, const char* zrname )`
Lädt die Trainingszeitreihe, führt die Rückskalierung durch und speichert die enstandene Zeitreihe.

`virtual void prognose_cleanup( const char* zrname )`
Löscht die von den externen Skripten geschriebene Trainingszeitreihe `RESULT.TRAIN`.

`private`-Methoden:

`virtual int info( TSBase* zr, void* data )`
Parsiert die Zeichenkette, die als Objektdatum an der Zeitreihe hängt. Diese wird mit `getObjectData()` ermittelt und muß folgende Form haben:
`PRESENCE|FUTURE <Tiefe>` Damit wird die Struktur `data` vom Typ `PRGAUSS_DATA` gefüllt.

7.7.5 Verfahren Naiv: `PRNaiv` (in `progno/PRNaiv.cpp`)

`public`-Methoden:

`virtual TSBase* prognose( TSBase* zr, List<TSBase*>* add, const char* data )`
Erstellt eine naive Prognose der Zeitreihe `zr`. Es dürfen keine zusätzlichen Zeitreihen eingehen, d.h., für den Parameter `add` muß `NULL` übergeben werden. In `data` steht als Parameter die Anzahl der zu prognostizierenden Werte. Die Methode nimmt den letzten Wert der Zeitreihe `zr` und prognostiziert diesen Wert in die Zukunft.

`private`-Methoden:

`virtual int info( TSBase* zr, void* data )`
Liefert nur einen Erfolg, wenn keine Daten an der Zeitreihe hängen, d.h., `getObjectData()` muß einen String der Länge 0 enthalten.

7.8 Projekte: `PJBase` und Subklassen

7.8.1 Projekte verwalten: `ProjectManager` (in `project/ProjectManager.cpp`)

`public`-Methoden:

`ProjectManager()`
Erzeugt eine leere Liste für die sekundären Projekte.

`virtual ~ProjectManager()`
Löscht das eingetragene primäre Projekt, alle sekundären Projekte in der Liste und die Liste selbst.

`PJBase* setPrimary( const char* project, ClassInit* init )`
Löscht das eingetragene primäre Projekt und ersetzt dieses durch das Projekt des Namens `project`. Bei der Initialisierung wird das Objekt `init` benutzt.

`PJBase* addSecondary( const char* name, ClassInit* init )`
Hängt das sekundäre Projekt des Namens `project` an die Liste aller sekundären Projekte an. Bei der Initialisierung wird das Objekt `init` benutzt. Die Methode prüft nicht, ob das Projekt bereits in der Liste vorhanden ist.

`PJBase* getPrimary()`
Liefert das primäre Projekt.

`PJBase* getFirstSecondary()`
Liefert das erste sekundäre Projekt in der Liste.

`PJBase* getNextSecondary()`
Liefert das nächste sekundäre Projekt in der Liste. Ist kein weiteres Projekt vorhanden, so wird `NULL` zurückgegeben.

`PJBase* getProjectForName( const char* name, ClassInit* init )`
Liefert das Projekt zu dem übergebenen Namen `name`. Ist dies noch nicht als primäres oder sekundäres Projekt vorhanden, so wird mit Hilfe des Initialisierungsobjekts `init` ein neues Projekt erzeugt und an die Liste der sekundären Projekte gehängt.

`int gotoProjectDir( const char* basedir, TSHead* zr, const char* comment, int mk_dir )`
Liefert das zum primären Projekt gehörende Basisverzeichnis. Dabei wird zuerst in das Verzeichnis `basedir` gewechselt, von dort aus in das projektspezifische Verzeichnis (abhängig von der Zeitreihe `zr` und dem Kommentar `comment`). Der Schalter `mk_dir` gibt an, ob das Verzeichnis erzeugt werden soll, falls es noch nicht existiert.

7.8.2 Basisklasse: `PJBase` (in `project/PJBase.cpp`)

`public`-Methoden:

`virtual const char* getTypeToken( int type )`
Liefert das Suffix zum Zeitreihentyp `type`. Intern wird `get_token()` aufgerufen.

`virtual const char* getTypeName( String komp[], int kanz, int type, int lang )`
Liefert den Namen der Zeitreihe, von der `kanz` Komponenten in `komp[]` übergeben werden. Außerdem wird der Typ der Zeitreihe in `type` angegeben, die gewünschte Sprache steht im Parameter `lang`. Intern wird `get_name()` aufgerufen.,

`virtual const char* getTypeDesc( String komp[], int kanz, int type, int lang )`
Liefert die Beschreibung der Zeitreihe, von der `kanz` Komponenten in `komp[]`

übergeben werden. Außerdem wird der Typ der Zeitreihe in `type` angegeben, die gewünschte Sprache steht im Parameter `lang`. Intern wird `get_desc()` aufgerufen.

`virtual int getTypeFromName( String komp[], int kanz, const char* name )`
Liefert anhand der Komponenten `komp[]` und des Namens `name` der Zeitreihe den Zeitreihentyp zurück. Wird kein Typ gefunden, so wird -1 zurückgegeben. Intern wird `get_name()` aufgerufen.

`virtual int getTypeFromDesc( String komp[], int kanz, const char* desc )`
Liefert anhand der Komponenten `komp[]` und der Beschreibung `desc` der Zeitreihe den Zeitreihentyp zurück. Wird kein Typ gefunden, so wird -1 zurückgegeben. Intern wird `get_desc()` aufgerufen.

`const char** getKompDesc( int cnt )`
Liefert die Beschreibung der `cnt`-ten Komponente. Als Ergebnis erhält man ein Array von Zeichenketten, die die Beschreibungen in den in `Language` angegebenen Sprachen enthalten. Intern wird `get_komp()` aufgerufen.

`const char* getKompDesc( int cnt, int lang )`
Liefert die Beschreibung der `cnt`-ten Komponente in der Sprache `lang`. Intern wird `get_komp()` aufgerufen.

`virtual String* getIndexHeading( String komp[], int cnt ) = 0`
Definiert die Schnittstelle, über die die Überschrift für einen Index abgefragt werden kann. Dabei werden die ersten `cnt` Komponenten in `komp[]` angegeben.

`virtual String* getIndexEntry( TSHead* zr ) = 0`
Definiert die Schnittstelle, über die man einen Eintrag im Index für die Zeitreihe `zr` abgefragt werden kann.

`virtual int getIndexType( int isprog ) = 0`
Definiert die Schnittstelle, über die man den Typ des Indexeintrages für die Zeitreihe `zr` abgefragt werden kann.

`virtual int gotoProjectDir( TSHead* zr, const char* comment, int mk_dir ) = 0`
Definiert die Schnittstelle, die in ein projektabhängiges Verzeichnis wechselt. Das Verzeichnis kann dabei von der übergebenen Zeitreihe `zr` und dem Kommentar `comment` abhängen. Der Parameter `mk_dir` gibt an, ob das Verzeichnis erzeugt werden soll, falls es noch nicht existiert.

`protected`-Methoden:

`virtual int getTypeCnt() = 0`
Definiert die Schnittstelle, die die Anzahl der Zeitreihentypen des Projektes angibt.

`virtual int getKompCnt() = 0`
Definiert die Schnittstelle, die die Anzahl der Komponenten des Projektes angibt.

`virtual const char* get_token( int type ) = 0`
Definiert die Schnittstelle, die zum übergebenen Zeitreihentyp `type` das Suffix des Zeitreihennamens zurückgibt.

`virtual const char* get_name( String komp[], int kanz, int type, int lang ) = 0`
Definiert die Schnittstelle, die zu den übergebenen Komponenten in `komp` (es sind

`kanz` Komponenten enthalten), dem Zeitreihentyp `type` und der Sprache `lang` den Namen der Zeitreihe liefert.

`virtual const char* get_desc( String komp[], int kanz, int type, int lang ) = 0`
: Definiert die Schnittstelle, die zu den übergebenen Komponenten in `komp` (es sind `kanz` Komponenten enthalten), dem Zeitreihentyp `type` und der Sprache `lang` die Beschreibung der Zeitreihe liefert.

`virtual const char* get_komp( int kompnr, int lang ) = 0`
: Definiert die Schnittstelle, die die Beschreibung der `kompnr`-ten Komponente in der Sprache `lang` liefert.

`virtual const char** get_komp( int kompnr ) = 0`
: Definiert die Schnittstelle, die die Beschreibungen der `kompnr`-ten Komponente in allen in `Language` definierten Sprachen liefert.

7.8.3 Globale Zeitreihen: `PJGlobal` (in `project/PJGlobal.cpp`)

Die Klasse `PJGlobal` ist für alle globalen Zeitreihen zuständig, zur Zeit sind dies Feiertage, Signifikanzen und Jahressaisons. Es werden also drei Zeitreihentypen definiert (`getTypeCnt()` liefert 3), jede globale Zeitreihe hat zwei Komponenten (`getKompCnt()` liefert 2).

`public`-Methoden:

`virtual String* getIndexHeading( String[], int )`
: Liefert als Überschriften `"Projekt"` (für `cnt == 0`) und `"globale Zeitreihen"` (für `cnt == 1`).

`virtual String* getIndexEntry( TSHead* head )`
: Liefert den Eintrag `JahresSaison` für Zeitreihen mit dem Subprojekt `JahresSaison` und den Eintrag `Feiertage` für Zeitreihen mit dem Subprojekt `Feiertage`

`virtual int getIndexType( int isprog )`
: Liefert immer den Wert 6 (für globale Zeitreihen).

`virtual int gotoProjectDir( TSHead* head, const char* comment, int mk_dir )`
: Wechselt in das Verzeichnis `Global/$KOMP2/$PROGNOSE/$PROGNOSE_SUB/<comment>`. Dabei ist `$KOMP2` die zweite Komponente im Zeitreihenkopf `head`. `mk_dir` gibt an, ob das Verzeichnis erzeugt werden soll, falls es noch nicht existiert.

`protected`-Methoden:

`virtual const char* get_token( int type )`
: Liefert die Suffixe `FTG`, `SIGN` oder `JSS`.

`virtual const char* get_name( String komp[], int kanz, int type, int lang )`
: Liefert nur anhand des Typs `type` und der Sprache `lang` die Namen `"Feiertage"` (`"feasts"`), `"Signifikanz"` (`"significance"`) oder `"Jahressaison"` (`"yearly-season"`).

`virtual const char* get_desc( String komp[], int kanz, int type, int lang )`
: Liefert das gleiche Ergebnis wie `getTypeName()`, da sich Namen und Beschreibungen bei globalen Zeitreihen nicht unterscheiden.

`virtual const char* get_komp( int kompnr, int lang )`
: Liefert den Namen der `kompnr` Komponente in der Sprache `lang`. Die erste Komponente bekommt den Namen `"Projekt"` (`"project"`), die zweite den Namen `"Subprojekt"` (`"subproject"`).

`virtual const char** get_komp( int kompnr )`
: Liefert zu den angegebenen Komponentennummer `kompnr` die Namen der Komponente in allen Sprachen (Werte siehe oben).

7.8.4 Projekt CO_2: `PJCO2` (in `project/PJCO2.cpp`)

Das CO_2-Projekt wird analog zum Projekt für globale Zeitreihen definiert. Hier werden 20 Zeitreihentypen und 1 Komponente eingeführt. Das Verzeichnis, in das gewechselt wird, ist bis auf das Projekt (`Global` wird durch `CO2` ersetzt) ebenfalls identisch.

7.8.5 Projekt Tegut: `PJTegut` (in `project/PJTegut.cpp`)

Auch das Tegut-Projekt wird analog zu den beiden anderen Projekten implementiert. Allerdings werden die Beschreibungen der Artikel nicht explizit in die Klasse geschrieben, sondern es wird bei einem Aufruf von `get_desc()` die Datei `ARTIKEL.TXT` parsiert. In dieser Datei werden die Artikelnummern mit dem Namen des Produktes verbunden.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung der Diplomarbeit

In dieser Diplomarbeit wurde der während einer mehrjährigen Projektarbeit entstandene Prototyp zur Zeitreihenprognose mit Hilfe objektorientierter Ansätze neu implementiert.

Im Vergleich zum Prototypen sind einige Neuerungen entstanden: Die wichtigste ist die Vorverarbeitung der Zeitreihen *on the fly*. Dadurch sind Tests, zum Beispiel zum Einfluß einiger Parameter oder Vorverarbeitungen auf die Prognoseergebnisse, flexibler und handhabbarer geworden.

Durch den objektorientierten Ansatz wurden die verschiedenen Arbeitsschritte (Vorverarbeitungen, Prognose, Zeitrechnung, Speicherung, Projektanbindung) klar voneinander getrennt implementiert. Durch diese Trennung ist eine Erweiterung einzelner Arbeitsschritte einfacher geworden. Durch die Abbildung auf Objekte muß einfach eine bestehende Klasse abgeleitet und die gewünschte Funktionalität hinzugefügt werden.

Durch die deutlich höhere Verarbeitungsgeschwindigkeit (dank einer compilierbaren Programmiersprache) ist auch eine Vorverarbeitung zur Laufzeit möglich geworden. Diese zusätzliche Funktionalität erleichtert den Test verschiedener Vorverarbeitungen (oder Parameter einer Vorverarbeitung). Hier ist das enstandene System also deutlich flexibler als der Prototyp geworden.

Das System kann in den Arbeitsablauf des Prototypen eingebettet werden, anstatt eines Aufrufes von `training.pl` muß nur das enstandene Trainingsprogramm der Diplomarbeit verwendet werden. Alle restlichen Perl-Skripte des Prototyps funktionieren weiterhin im Zusammenspiel mit dem neuen Programm.

Alle Quellen (insgesamt über 15.000 Codezeilen) sind auf der beiliegenden CD-ROM verfügbar, ebenso die Quellen der Diplomarbeit in LaTeX, HTML und PostScript.

8.2 Ausblick

Obwohl bereits ein Großteil der Funktionalität des Prototyps in C++ nachgebildet wurde, sind doch noch Teile des Systems unimplementiert geblieben. Dabei handelt es sich allerdings nur um Methoden, die im derzeitigen Prognoseablauf nicht benötigt wurden (z.B. Umrechnungen zwischen bestimmten Zeiteinheiten). Weiterhin wird teilweise noch auf Skripte des Prototyps zurückgegriffen, d.h., auch hier müssen (z.B. bei neuen Projekten) einige wenige Perl-Dateien gepflegt werden. Im folgenden sind deshalb die wichtigsten Erweiterungen aufgeführt, die im Anschluß an diese Arbeit durchgeführt werden können.

- Bei der Behandlung der Neuronalen Netze (mittels BP und SNNS) und der statistischen Tools (Gauss) könnten die Perl-Skripte `fe2bp.pl` und `create_prognose.pl` durch neu implementierte Prognoseklassen ersetzt werden. Allerdings ist diese Aufgabe in Perl bereits gut gelöst, da hier viel Stringbearbeitung und reguläre Ausdrücke auftreten, was in C++ nicht im Sprachumfang enthalten ist und von Hand programmiert werden müßte.
- Die Aggregation ist nur exemplarisch bei einigen Zeiteinheiten implementiert worden (Tage, Monate, Quartale), bei Bedarf ist hier ebenfalls noch Programmierarbeit zu leisten. Da die nötigen Schnittstellen bereits vorhanden sind, ist dies lediglich eine Umsetzung einiger Algorithmen zur Zeitrechnung.
- Es wurden nur einige Programme zur Anwendung der Funktionen der Klassenbibliothek erstellt, zusätzlich ein größeres Programm, das einen kompletten Prognoselauf mit Hilfe einer Steuerdatei startet (und somit das bestehende Perl-Skript `training.pl` ersetzt). Alle anderen Skripte, z.B. zur Auswertung der Prognoseergebnisse, sind nicht in C++ implementiert worden. Meist spricht auch (wie bei den Skripten zum Prognoseablauf, siehe oben) der hohe Anteil an Stringbearbeitung gegen eine Neuimplementation in C++.

Anhang A

Beispiel-Dateien des bestehenden Systems

A.1 Kopf einer Zeitreihe

```
# ErzeugtAm:            24.04.98
# Name:                 Wochen.ABV
# Faktor:               1
# Start:                2593 (36/1994)
# Ende:                 2778 (13/1998)
# Binaer:               0
# Komponenten:          3
# Aggregierung:         1
# Intervallbildung:     0
# Prognose:             0
# Maximum:              194
# Minimum:              32
# Mittelwert:           61.90323
# Varianz:              379.91537
# StdAbweichung:        19.49142
# D-Beschreibung:       010 KOELLNFL.BLUE.ZART500
# D-Zeitreihentyp:      Abverkauf
# D-Komponente1:        Tegut (Projekt)
# D-Komponente2:        AAA (Filiale)
# D-Komponente3:        229104 (Artikelnr.)
# D-ZeitEinheit:        Wochen
# D-Einheit:            Stck
# E-Beschreibung:       010 KOELLNFL.BLUE.ZART500
# E-Zeitreihentyp:      Sale
# E-Komponente1:        Tegut (Project)
# E-Komponente2:        AAA (branch)
# E-Komponente3:        229104 (Articleno.)
# E-ZeitEinheit:        Weeks
# E-Einheit:            pieces
```

Datei $PROJEKT_HOME/Tegut/AAA/229104/ZR/Head/Wochen.ABV

A.2 Rumpf einer Zeitreihe

```
60
64
74
64
45
66
61
59
182
103
70
66
71
57
84
57
41
60
52
44
...
```

Datei `$PROJEKT_HOME/Tegut/AAA/229104/ZR/Body/Wochen.ABV`

Im obigen Beispiel wurden aus Platzgründen nur die ersten 20 Zeilen der Datei angegeben.

A.3 Beispiele für INDEX-Dateien

```
Projekte
1 Tegut       | tegut... Projekt
1 CO2         | CO2 Projekt
1 Sunspots    | Sunspots
5 Global      | globale Zeitreihen
```

Datei $PROJEKT_HOME/INDEX

```
Filialen
1 AAA | Filiale AAA
```

Datei $PROJEKT_HOME/Tegut/INDEX

```
Artikel AAA/229104
2 Wochen.ABV | wochenweiser Abverkauf
2 Wochen.PRS | wochenweise Preise
2 Wochen.AKT | wochenweise Aktionen
2 Wochen.AUM | wochenweiser Umsatz
2 Tage.AKT | tageweise Aktionen
2 Wochen.Tegut_0 | Tegut Prognose+0
2 Wochen.BP_001.TRAIN | BP-Training [tst-tst]
2 Wochen.BP_001.TEST | BP-Testmenge [tst-tst]
2 Wochen.BP_001 | BP-Prognose [tst-tst]
```

Datei $PROJEKT_HOME/Tegut/AAA/229104/INDEX

A.4 Beispiel einer Prognoseliste

```
#FORMAT BP:Prognoseverfahren;Kommentar;Netzkonfiguration;Advance;
    min. Anzahl Epochen;max. Anzahl Epochen;Fehler alle Epochen;
    max. Fehler;Startgewichte aus;Lernverfahren;
    zufllige Trainingsmenge;Split;Eta;Alpha;Lernfaktor;
    Impulsfaktor;Testpaare;Validierungspaare

1:BP;BP-tst1;2::15:5(5):1;2;200;1000;40;0.01;[-0.5,0.5];ONLINE;
    33;;0.3;0.2;1;1;10;12;
    Tegut|AAA|229104|Wochen.ABV;
    CROP 2593 2660|UNSIGNED|INTEGER|LINEAR 0.1 0.9 41 194;
    UNKNOWN, Tiefe 5;
    Tegut|AAA|229104|Wochen.ABV;
    CROP 2593 2662|DIFF 1 1 ONESTEP Wochen.DIFF-TMP24
        |UNSIGNED|INTEGER|LINEAR 0.1 0.9 -79 147;
    UNKNOWN, Tiefe 5;
    Tegut|AAA|229104|Wochen.ABV;
    CROP 2593 2662|UNSIGNED|INTEGER|LINEAR 0.1 0.9 39 194;
    USE, Tiefe 1;
    Tegut|AAA|229104|Tage.AKT;
    AGGREGATE Wochen 1|CROP 2593 2660|BINARY 0.1 0.9 0.5 TSBinary;
    USE, Tiefe 1;
    Tegut|AAA|229104|Wochen.PRS;
    CROP 2593 2660|TREND -1 0 1;
    UNKNOWN, Tiefe 3;
    Tegut|AAA|Regal|Wochen.RUS;
    CROP 2593 2662|UNSIGNED|LINEAR 0.1 0.9 671.25 1691.2
```

Datei `$PROJEKT_HOME/Tegut/AAA/229104/PROGNOSE.LST`

Im obigen Beispiel wurden die zwei Zeilen der Datei `PROGNOSE.LST` auf mehrere Zeilen aufgeteilt, da die Zeilen zu lang für einen Ausdruck sind.

A.5 Beispiel für eine MODELL-Datei

```
# Abverkauf-Skalierungen
ABVERKAUF    LIN         UNSIGNED | INTEGER | LINEAR 0.1 0.9
ABVERKAUF    SIG4        UNSIGNED | INTEGER | MY_SIGMA 4
ABVERKAUF    DIFF1_2-LIN UNSIGNED | INTEGER | DIFF 1 2 ONESTEP | LINEAR 0.1 0.9
#
# Preis-Skalierungen
PREIS        TRD         TREND 0.1 0.5 0.9
#
# Aktions-Skalierungen (tage-/wochenweise)
AKTION_TAGE  BIN         BINARY 0.1 0.9 0.5
AKTION_TAGE  E_1T        EXPBINARY 1 1
```

Datei $PROJEKT_HOME/Tegut/MODELL

Anhang B

Speicherschutz und Speicherlöcher

Hier werden die Ideen vom Observer dargestellt. Dazu gehören die Makros, die C++-Methoden und das auswertende Perl-Skript (in Verbindung mit dem GNU-Debugger gdb).

Die dynamische Verwaltung von Speicher ist in C++ nicht so einfach zu handhaben, wie z.B. in Java, da hier kein automatisches Aufräumen stattfindet. Bei der Programmierung muß man daher darauf achten, daß jedes angeforderte Stück Speicher auch wieder freigegeben wird. Dies wird in größeren Quellen sehr schnell unübersichtlich.

Ebenfalls ist in C++ kein Mechanismus eingebaut, der bei Überschreiten von Arraygrenzen einen Fehler meldet. Ein falscher Zugriff führt entweder zu einem sofortigen Abbruch des Programms (segmentation fault) oder löst im weiteren Verlauf Fehler aus, die nicht nachvollziehbar sind.

Zur Lösung der beiden Probleme ist eine Klasse entstanden, die es möglich macht, Fehler in der Speicherverwaltung aufzudecken und damit einen großen Anteil an der Stabilität des Codes zu gewährleisten. Das Verfahren ist einfach und kann auch in fremde Programme aufgenommen werden.

In C++ sind nur zwei Methoden möglich, um dynamisch Speicher zu belegen und freizugeben. Für skalare Werte ist dies das Methodenpaar `new / delete`, bei Arrays kommen die eckigen Array-Klammern `[]` dazu. Diese vier Aufrufe sind durch vier Makros ersetzt worden, die statt `new` und `delete` in den Code gebaut werden:

```
NEW( ptr, class, init )
DELETE( ptr )
NEW_ARRAY( ptr, class, size, init )
DELETE_ARRAY( ptr, class, size )
```

Wobei `ptr` ein Zeiger auf das Objekt ist, `class` der Typ oder der Klassenname des Objekts, `init` eine Argumentliste für den Konstruktor (in runden Klammern) und `size` die Anzahl der Elemente im Array.

Weiterhin wird ein Makro `debug_printf()` zur Verfügung gestellt, das die gleichen Argumente besitzt wie der `printf`-Befehl, allerdings mit doppelter Klammerung.

Standard C++:	Observer-Makros:
x = new Class(y);	x = NEW(x,Class,(y));
x = new Class[N];	x = NEW_ARRAY(x,Class,N,());
x = new Class[N](5,6);	x = NEW_ARRAY(x,Class,N,(5,6));
return new Class();	return NEW(as_you_like,Class,());
delete x;	DELETE(x);
delete [] x;	DELETE_ARRAY(x,Class,N);
printf("Hello, world!");	debug_printf(("Hello, world!"));

Beispiel B.1:
Der folgende Programmcode macht deutlich, wie die Änderungen beim Einsatz des Observers ausfallen müssen. Das Programm erzeugt ein Integer-Array und löscht dieses gleich wieder:

```
// Standard C++
void main() {
  int *arr = new int[100];
  ...
  delete [] arr;
}
```

```
// mit Observer-Makros
void main() {
  int *arr = NEW_ARRAY(arr, int, 100, ());
  ...
  DELETE_ARRAY(arr, int, 100);
}
```

Ist beim Compilieren das Makro `OBSERVEMEM` definiert (über das Compiler-Argument `-DOBSERVEMEM`), so werden die erweiterten Makros aufgenommen. Wird das Makro nicht gesetzt, so werden die normalen `new` und `delete` Aufrufe eincompiliert. Dies ist wichtig, da man so bei einer endgültigen Programmversion ohne Änderung des Programmcodes sämtliche Debug-Informationen entfernen kann.

Das `debug_printf`-Makro wird ohne das Makro `OBSERVEMEM` einfach auskommentiert, mit gesetztem Makro wird eine eigene Ausgabemethode gestartet. Diese garantiert bei einer Ausgabe in eine Datei (s.u.), daß bei einem Abbruch mit `core`-File auch die letzten Meldungen noch ausgegeben werden. Bei einer Standard-Ausgabe mit `printf` kann hier durch die Pufferung von `stdout` etwas verloren gehen (insbesondere bei Benutzung von Pipes).

Zur Laufzeit des Programms kann der Benutzer entscheiden, wie er den Speicherschutz anwenden will, gesteuert durch eine Environmentvariable `$OBSERVEMEM`:

OBSERVEMEM	Funktion
nicht gesetzt	Speicherprüfung und Debug-Ausgabe inaktiv
-	aktiviert, Ausgabe auf stdout
`XXX`	aktiviert, Ausgabe in `XXX.mem` und `XXX.debug`

Hat der Benutzer die Ausgaben in eine Datei umgelenkt, so steht in der Datei XXX.mem der komplette Speicherverlauf des Programms, und in XXX.debug die Ausgaben von `debug_printf`. Jetzt kann der Ablauf der Speicherverwaltung auf Korrektheit geprüft werden. Zu diesem Zweck existiert ein Perl-Programm `observemem.pl`.

Das Skript bekommt den Namen der Speicherdatei übergeben, also:

```
observemem.pl XXX.mem
```

Jetzt wird die Datei parsiert und mögliche Fehler werden ausgegeben. Dazu zählen nicht freigegebene Bereiche, doppelt freigegebene Bereiche, `new/delete`-Paare die Skalare anfordern und Arrays freigeben (und umgekehrt), sowie Schreiben über die Arraygrenzen hinaus.

Mit dem Parameter `-m` kann eine Speicherkarte ausgegeben werden, d.h., die Variablen werden so sortiert ausgegeben, wie sie im Speicher liegen. Der Parameter `-c prgfile corefile` bezieht bei einem Programmabsturz das erzeugte `core`-File mit in die Betrachtung ein. Dazu muß allerdings der GNU-Debugger `gdb` installiert sein.

Mit Hilfe dieses Verfahrens kann man sicherstellen, daß keine Speicherlecks im Programm vorhanden sind, weiterhin ist ein korrekter Zugriff auf die Arrays sichergestellt. Nach der Entwicklungsphase muß das Programm nur einmal komplett ohne das `OBSERVEMEM`-Makro compiliert werden, Umstellungen im Code sind nicht mehr nötig.

Anhang C

Verzeichnisaufbau der Quellen und Objektdateien

`.`	Basisverzeichnis der Quellen (Makefiles, Hauptprogramme)
`./admin`	administrative Quellen (`Class`, `Administration`, `System`, `Message`, `Observer`, `Templates.cpp`).
`./modifier`	Vorverarbeitungen (`MDBase` und Subklassen).
`./persist`	persistente Speicherung (`PSBase` und Subklassen).
`./progno`	Prognoseverfahren (`PRBase` und Subklassen).
`./project`	Projekte (`PJBase` und Subklassen).
`./series`	Zeitreihenklassen (`TSHead` und Subklassen)
`./std`	sonst. Hilfsklassen (`String`, `List`, `Language`, ...)
`./time`	Zeiteinheiten (`TMBase` und Subklassen)
`./programs`	alle Programme, die die Klassenbibliothek benutzen (siehe Kapitel 5).
`./SYSTEM/$ARCH`	Objekte und Programme für Architektur `$ARCH`. Dabei sind die Objekte ebenfalls in Unterverzeichnissen verteilt, analog zu den zugehörigen C++-Quellen (z.B. `./SYSTEM/$ARCH/series/TSBase.o`).

Anhang D

Compilierung der Klassenbibliothek und der Programme

Die Klassenbibliothek dieser Diplomarbeit wurde mit Hilfe des GNU-C++-Compilers in der Version 2.7.2.1 und 2.7.3 erstellt. Frühere Versionen werden nicht unterstützt, da hier der Mechanismus der Registrierung nicht richtig umgesetzt wird (die Initialisierung von Klassenvariablen mit Hilfe von Methoden wurde noch nicht vollständig unterstützt). Die Klassenbibliothek wird als shared-library gebunden, die Bibliothek wird also vom Linker nicht an das Hauptprogramm gebunden, sondern zur Laufzeit vom System montiert.

Die Programme wurden unter den Unix-Systemen SunOS, Solaris und Linux erfolgreich übersetzt und getestet. Auf diesen Systemen ist das System der shared-libraries korrekt implementiert. Das Betriebssystem HP-UX wurde getestet, allerdings traten Kompatibilitätsprobleme zwischen Betriebssystem und Compiler bei der Benutzung der shared-libraries auf.

Ein Versuch, die Bibliothek auch als statische Bibliothek anzubieten, die dann durch den Linker an das Programm gebunden wird, machte folgende Probleme: In der Bibliothek waren noch alle Klassen enthalten, der Linker entscheidet aber selbständig, welche Klassen im Hauptprogramm referenziert werden. Dadurch werden nur ein Teil der Klassen zum Programm gebunden. Dadurch ist aber der Mechanismus der Klassensuche zur Laufzeit unbrauchbar, da unreferenzierte Klassen nicht zur Verfügung stehen.

Ein ähnliches Problem kann entstehen, falls die Objektdateien (`*.o`) direkt mit dem Hauptprogramm zusammengebunden werden. Ein Test auf Linux brachte den gewünschten Erfolg, alle Klassen wurden eingebunden. Der Linker von HP-UX benutzte allerdings ebenfalls nur die referenzierten Klassen, so daß auf diesem System kein lauffähiges Programm erstellt werden konnte.

Zum Compilieren der Quellen ist ein ausführliches Makefile mitgeliefert, bei einem Aufruf von `make` werden sowohl Bibliothek, als auch alle Programme im Verzeichnis `SYSTEM` (siehe Anhang C) erzeugt. Die shared-library muß danach in ein Verzeichnis kopiert werden, welches in der Environmentvariablen `$LD_LIBRARY_PATH` enthalten ist.

Anhang E

Quellen/Arbeit auf CD-ROM

Die Quellen und der Text der Diplomarbeit sind auf der beiliegenden CD-ROM enthalten. Die Diplomarbeit kann entweder als LaTeX-Quelle oder als Online-Dokument in HTML benutzt werden.

Die HTML-Version dieser Arbeit wurde mit Hilfe des latex2html-Paketes erstellt [Dra97], alle nötigen Konfigurationsdateien sind ebenfalls auf der CD-ROM enthalten. Verwendet wurde das Skript in der Version 96.1-h.

Zum Betrachten der Online-Dokumentation benötigen Sie einen WWW-Browser (Netscape, MS Internet Explorer, ...).

Aufbau der CD-ROM:

`/Html` hier befindet sich der komplette HTML-Teil, als Startseite muß `www.html` geladen werden. Zum Betrachten benötigt man einen HTML-Browser, der Frames und JavaScript unterstützt (z.B. die aktuellen Versionen von Netscape Communicator oder Microsoft Explorer).

`/LaTeX` hier befindet sich die komplette LaTeX-Quelle, inklusive aller Makefiles und Bilder. Zum Erzeugen der DVI- oder PostScript-Dateien wurde ein Unix-System (Solaris und Linux) benutzt.

`/PS` hier liegen die fertigen PostScript-Dateien zum Ausdruck bereit.

`/Sources` hier liegen alle Quellen der Diplomarbeit, der Aufbau innerhalb dieses Verzeichnisses ist in Anhang C erläutert.

`/Software` hier sind aktuelle Versionen der WWW-Browser von Netscape und Microsoft (für verschiedene Bestriebssysteme).

Literaturverzeichnis

[Apt96a] APTEC SYSTEMS, INC.: *GAUSS Applications: Time Series*, Januar 1996.

[Apt96b] APTEC SYSTEMS, INC.: *GAUSS Command Reference Manual*, September 1996.

[Apt96c] APTEC SYSTEMS, INC.: *GAUSS for UNIX*, Oktober 1996.

[Apt96d] APTEC SYSTEMS, INC.: *GAUSS System and Graphics Manual*, September 1996.

[Deu75] DEUTSCHER NORMENAUSSCHUSS: *DIN 1355, Blatt 1*, März 1975.

[Dra97] DRAKOS, NICOS: *The LaTeX2HTML Translator*, April 1997. WWW: http://www-dsed.llnl.gov/files/programs/unix/latex2html/manual/.

[Kop94] KOPKA, HELMUT: *LaTeX Band 1: Einführung.* Addison-Wesley, 1994.

[Kop95] KOPKA, HELMUT: *LaTeX Band 2: Ergänzungen.* Addison-Wesley, 1995.

[Kra] KRAUS, OLIVER: *Immer wieder Sonntag.* c't Magazin für Computertechnik, 15/1997:312–315.

[Kro97] KRONE, STEPHAN: *Eine interaktive Oberfläche zur grafischen Analyse von Zeitreihen.* Diplomarbeit, Universität Osnabrück, Juli 1997.

[Mid95] MIDDELBERG, ULRICH: *Parallele Backpropagation.* Diplomarbeit, Universität Osnabrück, September 1995.

[Mor98] MORANZ, KARSTEN: *Analyse und Auswertung von Zeitreihenprognosen mit dem Stuttgarter Neuronale Netze Simulator (SNNS).* Diplomarbeit, Universität Osnabrück, Mai 1998.

[Thi98] THIESING, FRANK M.: *Analyse und Prognose von Zeitreihen mit Neuronalen Netzen.* Dissertation, Universität Osnabrück, Mai 1998.

[WS91] WALL, LARRY und SCHWARTZ, RANDAL L.: *Programming perl.* O'Reilly & Associates, Inc., 1991.

[Zel95] ZELL, ANDREAS: *SNNS Stuttgart Neural Network Simulator User Manual, Version 4.1.* University of Stuttgart, Germany, 1995. Report No. 6/95.