Stilhandbuch für C++ Quellcode

Regeln zur Formatierung und zu anderen optischen Aspekten von C++ Quellcode

Martin Aupperle, 20.01.2003

Zusammenfassung:

Dieses Papier beschreibt die optischen Aspekte, die bei der Entwicklung von Software in C++ zu beachten sind. Es werden Regeln vorgestellt, die sich in der Praxis als sinnvoll erwiesen haben.

DAS PROBLEM

Normalerweise sieht jeder Programmierer, jeder Projektleiter, allgemein jeder am Projekt Beteiligte die Notwendigkeit von Programmiervorgaben als sinnvoll an. Das Problem besteht nur leider darin, dass jeder Programmierer eigene Vorstellungen darüber hat, wie diese Vorgaben aussehen sollen. In großen Projekten kann man normalerweise einem Codeabschnitt sofort ansehen, wer ihn geschrieben hat: zu unterschiedlich sind die Stile der einzelnen Mitarbeiter.

Selbstverständlich hält jeder einzelne „seinen“ Stil für den Besten. Schließlich praktiziert er ihn schon seit Jahren in vielen Projekten und ist bisher immer gut damit gefahren. Warum etwas ändern? Der Abgabetermin drängt, und für Stil wird schließlich niemand bezahlt.

Diese Denkweise lässt eine wichtige Erkenntnis außer acht: Quellcode dient nicht nur als „Futter“ für den Compiler, sondern vor allem zur Kommunikation zwischen Individuen. Quellcode wird geschrieben und oft verändert, aber weit aus häufiger gelesen. Wenn man „wirklich“ verstehen will, was eine Funktion macht, sieht man sich den Quellcode an, und nicht die Dokumentation. Arbeiten wie Fehlerbehebung, Performancetuning oder Weiterentwicklung setzen voraus, dass man sich mit dem Quellcode gründlich auseinandergesetzt hat. In letzter Instanz ist es der Code selber, der die ultimative Dokumentation einer Funktion, einer Klasse oder irgendeines anderen Programmkonstrukts darstellt.

Der letzte Abschnitt wirft einige Fragen auf. Wenn Code wesentlich öfter gelesen als geschrieben bzw. geändert wird, warum achten wir dann beim Programmieren hauptsächlich darauf „dass es läuft“, d.h. dass die funktionalen Anforderungen an die Software erfüllt werden? Wieso legen wir nicht mehr Wert auf Verständlichkeit des Codes für den menschlichen Leser? Lesbarer und aus sich selbst heraus verständlicher Code kann auf Dokumentation weitestgehend verzichten. Welchen Stellenwert soll also Dokumentation insgesamt haben? Was soll dokumentiert werden, und noch wichtiger, was braucht nicht dokumentiert werden? Ist es besser, ein komplexes Statement zu dokumentieren oder so um zu formulieren, dass es keiner weiteren oder zumindest weniger Erklärung bedarf?

DIE GRÜNDE

Die Antworten dieser Fragen liegen tief in den Strukturen und Zwängen der kommerziellen Softwareentwicklung, andererseits in urmenschlichen Verhaltensweisen verborgen. Um dies zu verstehen, muss man sich das Spannungsfeld vor Augen führen, in dem Entwicklung normalerweise abläuft.

Wird Software kommerziell entwickelt, gibt es normalerweise ein Lastenheft, in dem die Anforderungen an das fertige Produkt nieder gelegt sind. Wohl gemerkt: die Anforderungen *an das Produkt*, nicht *an den Quellcode.* Zeit- und finanzielle Rahmenbedingungen des Projekts sind in der Regel so, dass für Überlegungen, die nicht direkt zielführend sind, kein Raum ist. Schließlich soll die Entwicklung ja *effizient* sein, d.h. das Lastenheft soll mit möglichst geringem Aufwand implementiert werden. Konkurrenzsituationen, in denen mehrere Parteien für ein Projekt bieten, verstärken diesen Effekt noch. Der Gewinner (in der Regel der mit dem niedrigsten Preis) hat nun zwar ein Projekt, aber oft nicht mehr ausreichend Mittel, um Qualität zu produzieren. Die große Zahl von abgebrochenen Projekten, schlecht funktionierenden Systemen etc. ist mit durch diesen Effekt bedingt.

Menschliche Verhaltensweisen spielen ebenfalls eine wichtige Rolle. Wenn man intensiv an einem Problem arbeitet, hat man viele Details des Problems selber, der Lösung, des Umfeldes, Randbedingungen etc. im Kopf. Der schließlich entstehende Programmcode spiegelt all dies wieder – aber nur implizit. Warum der Programmierer eine lineare Liste anstatt eines

Vektors verwendet hat, kann man dem Code später nicht mehr ansehen, sondern nur noch die Tatsache an sich feststellen.

Während der Entwicklung entsteht kein Bedarf, solche Entscheidungen, Erkenntnisse etc. zu dokumentieren – man selber weiß ja warum und hat alles im Kopf. Dokumentationsarbeiten in diesem Stadium sind außerdem kontraproduktiv: sie halten auf, brauchen Zeit, und verhindern so eine vollständige Konzentration auf das Problem.

Leider geht diese zusätzliche, während der intensiven Entwicklungsarbeit noch vorhandene Information verloren. Denn während der Entwicklung will niemand zusätzlich Dokumentationsaufgaben leisten, und nach Abschluss der Entwicklung warten neue Aufgaben.

DER ANSATZ

Die Auseinandersetzung mit den aufgeworfenen Fragen der letzten Abschnitte führt zu der Überlegung, ob es nicht möglich ist, grundsätzliche Techniken, Strukturen, Vorgehensweisen oder Stile zu finden, die Software leichter lesbar (d.h. verständlicher) und damit auch wartungsfreundlicher machen.

Dieses Ziel versuchen die zahlreichen Programmierrichtlinien, Stilhandbücher etc. zu erreichen. Die meisten Entwicklungsabteilungen besitzen solche Vorgaben, und die meisten Programmierer wenden sie nicht an. Der Grund liegt häufig in der Unpraktikabilität der dort gemachten Vorschriften. Wenn eine Stilvorgabe z.B. Zeit und Überlegung benötigt, um umgesetzt zu werden, werden Manager und Programmierer aus den oben genannten Zwängen immer versuchen, zunächst einmal ohne aus zu kommen („*Dafür* haben wir *jetzt* keine Zeit“, „Nicht zielführend“, „do the obvious first“).

Ein Killerkriterium für die Anwendung von solchen Vorschriften ist immer, dass die Programmierer dahinter stehen. Vorschriften, die nicht angenommen werden, lassen sich in der Praxis auch nicht durchsetzen. Dies bedeutet, dass Vorschriften nicht wesentlich behindern dürfen, und dass der konkrete Nutzen sofort erkennbar wird. Die Kunst besteht darin, ein Regelwerk zu schaffen, das einerseits Antworten zu den oben gestellten Fragen liefert, andererseits aber die Arbeit nicht zu weit behindert.

Die in der Praxis eingesetzten Regeln sind in den meisten Fällen zu rigide. Eine Vorschrift, dass jede Klasse einen virtuellen Destruktor besitzen muss, macht in der Praxis einfach keinen Sinn, auch wenn sie für viele Klassen zutrifft. Genau so verhält es sich mit der immer wieder an zu treffenden Vorschrift, dass Mitgliedsvariablen einer Klasse nicht öffentlich sein dürfen. Oder dass globale Daten grundsätzlich verboten sind – die Aufzählung könnte beliebig fort gesetzt werden.

Die gerade genannten Regeln haben die Eigenschaft, dass damit etwas grundsätzlich Erstrebenswertes ausgedrückt werden soll, dieses aber für *alle* Situationen als korrekt behauptet wird. Beispiel: Es ist erstrebenswert, den Gültigkeitsbereich eines Namens (z.B. einer Variable) möglichst klein zu halten. Daraus folgt neben vielen anderen Dingen z.B. auch, dass Variablen nicht unnötig global deklariert werden sollen, sondern eben möglichst lokal. Wer daraus die generelle Regel „Globale Variablen sind verboten“ macht, hat etwas Wesentliches nicht verstanden.

IN DIESEM PAPIER

In diesem Papier konzentrieren wir uns optische Fragen. Wir befassen uns damit, wie man gute Namen für Klassen, Variablen und Funktionen (allgemein: für Programmobjekte) findet, und was überhaupt einen „guten“ Namen ausmacht. Wir werden sehen, wie man Code optisch so strukturieren kann, dass er leicht zu lesen ist. Wir werden sehen, wie man Code verständlich formuliert. Wir werden sehen, welche Rolle Inline-Dokumentation spielt.

Kurz: Wir werden Software nicht mehr unter dem rein funktionalen Aspekt betrachten, sondern aus der Sicht eines Mitarbeiters, der sich (wieder) mit den Abläufen vertraut machen muss.

NICHT IN DIESEM PAPIER

Wir befassen uns in diesem Dokument ausschließlich mit optischen Fragen, also der Anordnung von Code auf Papier (bzw. analog auf dem Bildschirm). Ein vollständiges Programmierhandbuch kann und sollte jedoch durchaus weitere Dinge regeln. In welchen Fällen ist z.B. das Überladen von Operatoren sinnvoll[1]? Wie ist das Verhältnis zwischen *malloc/free* und *new/delete*? Wann sollte eine Klasse einen virtuellen Destruktor erhalten? Wann ist Komposition der Ableitung vorzuziehen? Was ist bei wirklich großen Systemen zu beachten? Wie stehen wir zu *Design by Contract*? Welchen Stellenwert sollte *Design for Debugging* einnehmen?

Jedes Mitglied eines Programmierteams (auch der Manager!) sollte fundierte Antworten zu diesen Fragen für sein konkretes Projekt geben können. Optimal wäre es, wenn die Antworten auch noch identisch wären. Fordern Sie als Programmierer klare Antworten zu diesen Themen von Ihrer Leitung!

NAMEN: MEHR ALS SCHALL UND RAUCH.....

Das Finden von guten Namen gehört zu den schwierigeren Aufgaben bei der Planung und Implementierung von Software. Der Compiler hat noch die wenigsten Probleme damit: Hauptsache, alle Namen sind unterschiedlich, dann ist eine Zuordnung zum damit bezeichneten Programmobjekt sicher gestellt.

Als Leser von Quellcode müssen wir jedoch weiter gehende Anforderungen stellen. Die folgenden beiden Punkte sind essentiell:

- Aus einem Namen muss auf seine (fachliche) Bedeutung geschlossen werden können. Liest man z.B. *account_no*, wird damit die Nummer eines Kontos gemeint sein.

  Diese Forderung ist nur auf den ersten Blick trivial. Was bedeuten z.B. *ACTNBR* oder *nract*? Ein ungeübter Leser wird nur mit Mühe die Assoziation zu *Account Number* herstellen können. Wir werden gleich noch sehen, welche Namensregeln dazu führen, dass die Assoziation leicht fällt.

[1] Das Überladen von Operatoren ist ein Sprachmittel, das in der Praxis viel zu häufig eingesetzt wird. Der Grund ist wahrscheinlich, dass man nach der Teilnahme an einer C++-Schulung die ganzen neuen Dinge sofort im eigenen Projekt ausprobieren will und die – zugegebenermaßen faszinierenden – Sprachmittel der Sprache C++ falsch einsetzt.

- Umgekehrt muss aus der fachlichen Bedeutung eines Programmobjekts auf seinen Namen geschlossen werden können. Sucht man also nach dem Variablennamen für die Kontonummer, wird man zu etwas wie *account_no*, *account_nbr*, *accountNbr*, *nbrAccount* etc. kommen.

  Wichtig ist hierbei, dass man jedes Mal, wenn man den Variablennamen sucht, ohne viel Überlegen zum gleichen Ergebnis kommt. Dies soll vor allem auch dann gelten, wenn später jemand anderes den Code ändert. Es ist einfach lästig, ständig die Schreibweise von existierenden Variablen nachsehen zu müssen, um sie korrekt verwenden zu können.

  Wie man leicht erkennen kann, ist die zweite Forderung ungleich schwerer zu erfüllen als die erste. Aber auch hier gibt es jedoch einige Regeln, die bei konsequenter Anwendung die Sache erheblich erleichtern.

Alle weiteren Überlegungen zur Vergabe von Namen orientieren sich an diesen beiden zentralen Forderungen.

BESTANDTEILE EINES NAMENS

Namen sollen in erster Linie die fachliche Verwendung des bezeichneten Programmobjekts wieder spiegeln. Begriffe wie *Check*, *Balance*, *Account*, *Order*, *Process*, *Temperature, Amount, Vehicle, Calculate, Increase* etc. kommen deshalb häufig als Namensbestandteile vor.

Zusätzlich kann es sinnvoll sein, bestimmte technische Sachverhalte in einem Namen zu codieren. So erhalten z.B. Datenmitglieder einer Klasse oft ein vorangestelltes *m*, um sie in einer Mitgliedsfunktion von Parametern und lokalen Variablen unterscheiden zu können.

ALLGEMEINE REGELN FÜR DEN FACHLICHEN NAMENSTEIL

LÄNGE VON NAMEN

Untersuchungen[2] haben heraus gefunden, dass Namen mit einer Länge von 10 bis 16 Zeichen für die Lesbarkeit optimal sind. Längere Namen sind schwer zu erfassen, kürzere Namen beinhalten zu wenig Information, um ihre Bedeutung zu beschreiben.

Zu langer Name	***Besser***
AnzahlFahrzeugeImFuhrpark	*AnzFhrzgFhrprk* oder *AnzFhrzg*
prüfeBankleitzahlAufKorrektheit()	*prfBlz() oder checkBlzSyntx()*

Die langen Namen wurden durch geeignete Abkürzung von Namensteilen (s.u.) sowie durch Weglassen von unwichtigeren Namensbestandteilen in kürzere, leichter zu handhabende Namen überführt.

[2] Gorla, Benander et al 1990.

Zu kurzer Name	*Besser*
tArv	*tmeToArrv* oder *timeToArrival*
spd	*speed*

tArv ist zu kurz, um die damit verbundene fachliche Größe (nämlich die Zeit bis zur Ankunft) erschließen zu können. *speed* ist kurz genug und braucht nicht zu *spd* abgekürzt zu werden.

Insgesamt gilt also: verwenden Sie die gesamte Bezeichnung des Fachobjekts, der fachlichen Klasse, der Tätigkeit oder des Zustandes um einen Namen zu bilden. Vollständige Bezeichnungen können z.B. *amount*, *account*, *Konto*, *Währung*, *vehicle*, *calculate*, *display*, *drucken* sein. Nur wenn die daraus entstehenden Namen zu lang sind, sollen Abkürzungen (s.u.) verwendet werden.

AUSNAHME ZUR 10-16-ZEICHEN-REGEL

Es gibt eine Ausnahme der 10-16 Zeichen-Regel: Laufvariablen in Schleifen werden traditionell mit Einzelbuchstaben (*i*,*k*,*l*,*x* etc.) bezeichnet. Dies ist vertretbar, denn

- die Lebenszeit der Variablen ist auf die Schleife beschränkt, also sehr kurz. Die Variable hat nur sehr lokale Bedeutung
- die Variable wird regelmäßig zum Durchlaufen von Datenstrukturen (Felder, lineare Listen etc.) verwendet.

Es ist durchaus in Ordnung, etwas wie

```
for ( int i = 0; i < accounts.getSize(); i++ )
{
  cout << accounts[ i ] << endl;
}
```

zu schreiben. Der Gültigkeitsbereich der Variablen *i* beschränkt sich auf die Schleife, die mit einem Blick übersehen werden kann. Die Verwendung von *i* ist also implizit klar: ein aussagekräftiger Name ist nicht unbedingt erforderlich.

Interessant ist, dass ein aussagekräftiger Name für die Schleifenvariable den Code unter Umständen sogar schlechter lesbar macht. Vergleichen Sie dazu den folgenden Codeabschnitt:

```
for ( int accountIndex = 0; accountIndex < accounts.getSize();
accountIndex++ )
{
  cout << accounts[ accountIndex ] << endl;
}
```

Um den Code verstehen zu können, muss ein Leser die vier Vorkommen der Zeichenkette *accountIndex* als identisch klassifizieren. Dazu muss er genauer hinsehen als in der ersten Version[3].

[3] Der Effekt ist hier gering, da das Beispiel nur zur Demonstration dient. Wenn man viel Code lesen muss, macht er sich jedoch bemerkbar.

GÜLTIGKEITSBEREICH UND LÄNGE VON NAMEN

Die Verwendung von kurzen Namen ist vertretbar, wenn die Lebenszeit der Variablen sehr gering ist. Dies gilt häufig für Schleifenvariablen, aber auch z.B. für Parameter in kurzen (inline-) Funktionen.

In diesem Beispiel ist die Wahl der kurzen Namen *v1* und *v2* in Ordnung:

```
double calcMean( double v1, double v2 )
{
  return ( v1 + v2 ) / 2.0;
}
```

Lesbarkeit und Verständlichkeit würden durch Verwendung von „korrekten“ 10-16-Zeichen-Namen nicht erhöht, sondern wahrscheinlich verschlechtert:

```
double calcMean( double firstOperand, double secondOperand )
{
  return ( firstOperand + secondOperand ) / 2.0;
}
```

ABKÜRZUNGEN

Abkürzungen sollen nur verwendet werden, wenn mindestens eine der folgenden Bedingungen zutrifft:

- der vollständige Name wäre zu lang
- der vollständige Begriff wird regelmäßig und überall sonst auch abgekürzt (z.B. km anstelle Kilometer)
- der vollständige Begriff wird sehr häufig verwendet, bedarf keiner Erklärung und ist jedem geläufig (z.B. *brng* und *hdng* für *bearing* und *heading* in eines GPS-Navigationssoftware).

Für bestimmte Begriffe sollen grundsätzlich Abkürzungen verwendet werden. Dazu gehören Begriffe mit fester technischer oder fachlicher Bedeutung:

Vollständiger Begriff	***Abkürzung***
Claculate	*Calc*
Display	*Disp*
Append	*app*
Delete	*del*
Number	*Nbr, Num oder No*
Coordinate	*Coord oder crd*

Hier bringt das vollständige Ausschreiben nicht mehr an Information.

- Bei der Bildung von Abkürzungen sollen zuerst geeignete Vokale aus dem Begriff entfernt werden[4]. Erst wenn der Name immer noch zu lang ist, sollte man sich Gedanken über weitere Auslassungen machen. Beispiele:

Vollständiger Begriff	***Abkürzung***
account	*Acnt*
amount	*Amnt*
Fahrzeug	*Fhrzg*
Transporter	*Trnsprtr*

- Eine weitere Möglichkeit besteht im Weglassen der hinteren Teile eines Begriffs.

Vollständiger Begriff	***Abkürzung***
Account	*Acc*
Amount	*Amo*
Fahrzeug	*Fahrz*
Transporter	*Transp*
Referenz	*Ref*
Maximum	*Max*

  Diese Möglichkeit zur Bildung von Abkürzungen wird häufig bei Zusammensetzungen (s.u.) verwendet, und hat eigentlich nur dort Vorteile:

Vollständiger Begriff	***Abkürzung***
MaximumLevel	*MaxLvl*
calculateAverage	*calcAverage*

- Schließlich gibt es die Möglichkeit, bei längeren zusammen gesetzten Begriffen nur die jeweiligen Anfangsbuchstaben oder Anfangssilben zu verwenden.

Vollständiger Begriff	***Abkürzung***
Global Positioning System	*GPS*

Werden Abkürzungen verwendet, müssen sie konsistent und überall gleich angewendet werden. Optimal ist die Führung einer Liste mit Fachbegriffen und zugehörigen Abkürzungen. Insgesamt sehen Abkürzungen, die nach einer der drei obigen Regeln gebildet wurden, manchmal etwas gewöhnungsbedürftig aus. Die Regeln haben jedoch die positive Eigenschaft, dass die zu einem Begriff gehörige Abkürzung zweifelsfrei und mit wenig Aufwand (d.h. ohne Auswendiglernen) gebildet werden kann. In der Praxis gewöhnt man sich sehr schnell daran.

[4] Der Grund ist, dass es nur vier Vokale gibt, ein Vokal daher wesentlich weniger Information trägt als ein Konsonant. Wenn man also schon auf etwas verzichten muss, dann auf diejenigen Teile mit dem wenigsten Inhalt.

GUTE UNTERSCHEIDBARKEIT VON NAMEN

Damit Namen auf den ersten Blick korrekt auf genommen werden können, müssen alle Namen gut von einander unterscheidbar sein. Dies erreicht man in erster Linie durch die Beachtung der folgenden beiden Regeln:

- Namen oder Namensteile, die sich nur durch die Groß- / Kleinschreibung unterscheiden, sollen vermieden werden.

Name 1	***Name 2***
biDirectional	*bidirectional*
KontoStand	*Kontostand*

  Ein weiterer Nachteil ist, dass solche Namen identisch ausgesprochen werden – die Kommunikation mit anderen über das Programm wird unmöglich.

- Namen oder Namensteile sollen optisch sofort unterscheidbar sein. Ähnlich aussehende Zeichen sind zu vermeiden.

Name 1	***Name 2***
name_length	*name_length*
BetragsSumme	*Betrags5umme*

  Die folgenden Zeichen sind im Schriftbild ähnlich und somit prädestiniert für Verwechslungen:

Buchstabe	***ausgeschrieben***
l I 1	kleines l, großes i, Zahl 1
O 0	großes o, Zahl 0
Z 2	großes z, Zahl 2
S 5	großes s, Zahl 5
G 6	großes G Zahl 6

GUTE AUSSPRECHBARKEIT VON NAMEN

Unterhält man sich mit anderen über Details eines Programms, muss man oft Namen von Programmobjekten aussprechen. Verwendet man vollständige Begriffe wie oben empfohlen, bildet die Aussprechbarkeit normalerweise kein Problem. Abkürzungen dagegen können schwierig auszusprechen sein. Worte wie *Amnt*, *Fhrzg* oder *KZBV* lassen sich nicht wirklich verständlich aussprechen. In solchen Fällen verwendet man dann (für die Kommunikation, nicht im Programm!) die vollständigen Namen (*Amount*, *Fahrzeug*) oder buchstabiert (*Ka-Zet-Be-Vau*).

ENGLISCH ODER DEUTSCH?

Die Frage nach der Sprache, in der Namen abgefasst werden sollen, ist nicht eindeutig zu beantworten. In Frage kommen praktisch Englisch und Deutsch. Die meisten Bibliotheken verwenden angloamerikanische Begriffe. Benutzt man für eigene Namen die Deutsche Sprache, erhält man ein Mischung zwischen beiden Welten[5]:

```
void FlacherKnopf::PreSubclassWindow()
{
  CButton::PreSubclassWindow();

  //-- In der Dialogvorlage muss sichergestellt sein, dass
  //   static edge eingestellt ist
  //
  DWORD erweiterterStil  = GetExStyle();
  ASSERT((erweiterterStil & WS_EX_STATICEDGE) != 0);
}
```

Dies liest sich nicht flüssig. Der *extended window style* ist ein Begriff, der sich in der Original- Windows-Dokumentation findet und sich auch nicht gut ins Deutsche übersetzen lässt. Wozu auch? Die wesentliche Literatur über Windows ist nun einmal in Englisch, genau so wie die meisten Beispielprogramme. Die Funktion, die den zugehörigen Wert liefert, verwendet *ExStyle* als Namensteil für den extended style. Unsere Variable dagegen verwendet für den gleichen fachlichen Begriff *erweiterterStil*.

Gleiches gilt für den Klassennamen. Der Klassenname für Standard-Knöpfe in den MFC ist *CButton*. Wir dagegen verwenden für eine Ableitung (mit erweiterter Funktionalität) *FlacherKnopf*.

Insgesamt hat man es also für einen fachlichen Begriff (hier also z.B. einen bestimmten Knopf in einem Dialog) mit zwei technischen Begriffen (*CButton* und *FlacherKnopf*) zu tun, die phonetisch nichts mit einander zu tun haben.

Ein weiterer, kleinerer Punkt, der gegen deutsche Namen spricht, ist der erlaubte Zeichensatz für Namen. C++ lässt z.B. die deutschen Umlaute nicht zu. Der Begriff Füllhöhe müsste daher in den Namen FuellHoehe umgesetzt werden.

Verwendet man auch für eigene Namen die englische Sprache, vermeidet man diese Probleme:

```
void FlatButton::PreSubclassWindow()
{
  CButton::PreSubclassWindow();

  //-- In der Dialogvorlage muss sichergestellt sein, dass
  //   static edge eingestellt ist
  //
  DWORD exStyle  = GetExStyle();
  ASSERT((exStyle & WS_EX_STATICEDGE) != 0);
}
```

Beachten Sie bitte, dass der Kommentar weiterhin in deutsch bleibt.

Auf der anderen Seite ist die Entwurfsdokumentation häufig in Deutsch abgefasst. Der Kunde spricht von Kontonummer und Fahrzeuggewicht, und nicht von *account number* oder *vehicle weight*. Möchte man die englischen Begriffe verwenden, hat man den Bruch dann zwischen (Design-)dokumentation und Programm.

---

[5] Hier am Beispiel der weit verbreiteten *Microsoft Foundation Classes*.

Beide Fälle führen in der Praxis zu unschönen Situationen. Man sollte sich also für einen Weg entscheiden und diesen auch durchhalten. Die Problematik zeigt außerdem, wie wichtig ein korrekt geführtes Begriffsbuch ist.

ZUSAMMEN GESETZTE NAMEN

Begriffe bestehen oft aus mehreren Komponenten: *account number*, *Fahrzeuggewicht*. Bei der Bildung der zugehörigen Namen sollen die Komponenten klar als solche erkennbar sein. In der Praxis werden die beiden folgenden Methoden verwendet:

Methode	***Ergebnis***
Anfangsbuchstaben der Komponenten werden innerhalb des Namens groß geschrieben, der Rest klein. Komponenten werden direkt aneinander gehängt.	*AccountNbr*
Komponenten werden durch einen Unterstrich getrennt. Komponentennamen werden in der Regel vollständig klein geschrieben.	*account_nbr*

Die Trennung durch Unterstrich findet man häufig in älteren Programmen. Sie hat den großen Nachteil, dass man in längeren Anweisungen oder Codeblöcken nicht ohne genaues Hinsehen die vielen Einzelkomponenten zu Namen verbinden kann:

```
void update_customer( Customer& the_customer )
{
  if ( !aquire_database_lock() )
    throw database_error_no_lock;
  the_customer.set_update_flag();
  the_customer.update_score(the_customer.get_score() + standard_increase);
  the_customer.write_to_database();
  the_customer.release_update_flag();
  release_database_lock();
}
```

So etwas ist extrem schwer zu lesen. Vergleichen Sie dies mit folgender Formulierung der Funktion:

```
void updateCustomer( Customer& customer )
{
  if ( !aquireDatabaseLock() )
    throw databaseErrorNoLock;
  customer.setUpdateFlag();
  customer.updateScore( customer.getScore() + standardIncrease );
  customer.writeToDatabase();
  customer.releaseUpdateFlag();
  releaseDatabaseLock();
}
```

Der Unterstrich ist – genau so wie Punkt und Komma – ein *semi-whitespace*, d.h. ein Zeichen, das fast wie ein Leerzeichen wirkt. Entsprechend ist der optische Unterschied z.B. zwischen *database_error* und *database error* nur gering. Es ist nicht sofort klar, ob es sich um ein oder um zwei Programmobjekte handelt.

In neu erstellten Programmen sollte daher immer die erste Regel angewendet werden. Zusammen mit den Regeln zur Verwendung von Leerzeichen (s.u.) wird erreicht, dass sich zusammengesetzte Namen optisch schnell erkennen und von anderen Programmelementen separieren lassen.

Eine Sonderstellung nehmen Makros ein. Sie werden traditionell vollständig groß geschrieben. Eine Trennung der Namensteile kann dann nur durch Unterstriche erfolgen:

```
TRACE_WITH_PARAMETERS( ... )
```

Weitere Regeln:

- Wird eine Komponente bis auf die Anfangsbuchstaben abgekürzt, sind diese in zusammen gesetzten Namen ebenfalls (bis auf den ersten Buchstaben) klein zu schreiben. An Stelle von *connectSQLDatabase* schreibt man besser *connectSqlDatabase*. Durch diese Schreibung wird die Trennung der einzelnen Komponenten deutlicher.
- Zusammen gesetzte Begriffe bestehen häufig aus einem spezielleren und einem allgemeineren Teil. Im Begriff *Fahrzeuggewicht* ist *Fahrzeug* die spezielle und *Gewicht* die allgemeinere Komponente. Weitere Beispiele:

Vollständiger Begriff	***Spezielle Komponente***	***Allgemeine Komponente***
account number	*account*	*number*
x-Koordinate	*Koordinate*	*x*
Schiffslänge, Breite, Höhe	*Schiff*	*Länge, Breite, Höhe*

Bei der Bildung des Namens sollte man sich grundsätzlich entscheiden, ob die spezielle Komponente vorne oder hinten stehen soll:

Allgemeine Komponente vorne	***Spezielle Komponente vorne***
nbrAccount	*accountNbr*
xCoord	*coordX*
schiffLaenge	*laengeSchiff*

Für die Wahl „Spezielle Komponente zuerst“ gibt es zwei Argumente:

1. spezielle Komponenten enthalten mehr Information als allgemeine. Sie sind wichtiger und sollten daher zuerst kommen.
2. In der Praxis folgen oft mehrere Statements aufeinander, in denen Variablen mit der gleichen Hauptkomponenten verwendet werden:

```
accountNbr            = ...
accountOwner          = ...
accountAccessMode     = ...
```

Dies liest sich auf Grund der günstigeren vertikalen Anordnung (s.u.) besser als

```
nbrAccount            = ...
ownerAccount          = ...
accessModeAccount     = ...
```

Hinzu kommt, dass die erste Version in den meisten Fällen dem natürlichen Sprachgebrauch entspricht: es heißt *account number* und nicht *number account*, *Schiffslänge* und nicht *Länge Schiff*.

- In zusammen gesetzten Begriffen, deren Komponenten Verben und Substantive sind, sollten die Verben voran gestellt werden. Ausschließlich Funktionsnamen sollten Verben als Komponenten enthalten:

```
setUpdateFlag();
increaseAmount();
calculateVelocity();
```

  Der Name soll insgesamt so gewählt werden, dass sich das Verb (also die Tätigkeit, die die Funktion ausführt) auf das nachfolgende Substantiv bezieht:

Name	***Aktion***
setUpdateFlag()	das updateFlag wird gesetzt
increaseAmount()	der Betrag wird erhöht
calculateVelocity()	die Geschwindigkeit wird berechnet

- Im Falle von Mitgliedsfunktionen bezieht sich die Tätigkeit häufig auf das eigene Objekt. In diesem Fall soll die Komponente, die das eigene Objekt bezeichnet, weggelassen werden. Für eine Klasse *Account* deklariert man eine Mitgliedsfunktion zur Lieferung des Eigentümers deshalb als

```
class Account
{
public:

  string getOwner() const;

  /* .. weitere Mitglieder von Account */

};
```

  und nicht etwa als

```
string getAccountOwner() const;
```

- Analoges gilt für die Mitgliedsvariablen einer Klasse. Man schreibt also

```
class Account
{
  /* .. weitere Mitglieder von Account */
private:
  string mOwner;
};
```

anstelle von

```
  string mAccountOwner;
```

Auf das Prefix *m* für Datenmitglieder einer Klasse kommen wir weiter unten zurück.

SPEZIELLE REGELN FÜR DEN FACHLICHEN NAMENSTEIL

Neben den eher allgemeinen Regeln aus dem letzten Abschnitt gibt es einige spezielle Namenskonventionen, die hier vorgestellt werden.

NAMEN FÜR BESTIMMTE WERTE

Die Tatsache, dass eine Variable das Minimum aus einer Reihe von Werten speichert, kann im Variablennamen angegeben werden. Eine Variable für den niedrigsten Wasserstand in der letzten Woche könnte z.B. *MinWaterLevel*, *WaterMinLevel* oder *WaterLevelMin* heißen. Der Namensteil „Min“ ist sehr allgemein und sollte deshalb am Ende des Namens stehen. Die Variable sollte also *WaterLevelMin* heißen.

Die folgenden Werte kommen in Programmen immer wieder vor:

Wert	***Namensteil***	***Beispiel***
Minimum, Maximum aus einer Reihe von Werten	*Min, Max*	*waterLevelMin, waterLevelMax*
Durchschnitt aus einer Reihe von Werten	*Avg*	*waterLevelAvg*
Summe von Werten	*Sum*	*WeightedErrorSum*
Anzahl von Einträgen in Containern (Felder etc.)	*Size*	*getSize()*
Index in einen Container	*Ofs, Index, Idx*	*elementOfs, elementIdx*
Aktueller Wert aus einer Auswahl von Werten	*act*	*personAct*
Erster/Letzter Wert in einer Reihe von Werten	*First, Last*	*bankFirst, bankLast*
Anzahl von Werten	*Anz, Count*	*particleCount*

Die Namensteile *Min*, *Max*, *Avg* etc. sind sehr allgemein und sollten deshalb nach der Regel aus dem letzten Abschnitt zur Ordnung von Namensteilen am Ende des Namens stehen, wie in obiger Tabelle angegeben. Viele Programmierer bevorzugen jedoch genau die umgekehrte Notation. Sie schreiben lieber *actElement*, *avgWaterLevel* und *firstEntry*. Dies ist in Ordnung, so lange das Schema im Projekt konsistent durch gehalten wird.

Eine Sonderstellung nimmt der Namenteil *Num* (analog *Nbr* oder *Nr*) ein. Er kann sowohl am Anfang eines Namens als auch am Ende auf treten mit jeweils anderer Bedeutung.

Wert	***Namensteil***	***Beispiel***
Anzahl von Werten, Summe	*Num* am Anfang	*numPersons*
laufende Nummer, Index etc.	*Num* am Ende	*personNum*

Diese Doppelbedeutung ist natürlich nicht optimal. Meist kann man auf *Num* verzichten und statt dessen für die erste Bedeutung Anzahl bzw. *Count* und für die zweite *Index* oder *Offset* verwendet.

KOMPLEMENTÄRE NAMEN

Wenn es in einem Programm eine Routine zum Sperren von Datensätzen gibt, wird in der Regel auch eine zum Freigeben benötigt. Die Namensgebung dieser beiden Routinen sollte diesen inneren Zusammenhang aus drücken. Folgende Tabelle gibt eine Reihe von solchen komplementären Namen:

Name und Komplement	***Name und Komplement***	***Name und Komplement***
Min / Max	add / remove	source / target
Old / New	suspend / resume	source / destination
Low / High	lock / unlock	create / destroy
First / Last	show / hide	suspend / continue
Begin / End	start / stop	acuire / release
Up / Down	get / set	allocate / dispose
Next / Prev	get / put	open / close
	peek / poke	
	store / retrieve	

GETTER UND SETTER

Getter und *Setter.* Ein *Getter* ist eine Mitgliedsfunktion, die den Wert einer oder mehrerer Werte[6] aus einem Objekt liefert. Getter erhalten als Verb grundsätzlich den Namensteil *get:*

```
class Account
{
public:

  string getOwner() const;

  /* .. weitere Mitglieder von Account */

};
```

Ein Getter führt keinerlei Berechnungen aus[7] und ändert den Zustand des Objekts nicht. Getter sind grundsätzlich konstant deklariert.

[6] Werte aus dem problem domain. Ein solcher Wert kann, muss aber nicht direkt einer Mitgliedsvariablen entsprechen.

Ein *Setter* ist eine Mitgliedsfunktion, die eine oder mehrere Mitgliedsvariable(n) mit Werten versorgt. Setter erhalten als Verb grundsätzlich den Namensteil *set*.

```
class Account
{
public:

  void setOwner( const string& );

  /* .. weitere Mitglieder von Account */

};
```

Ein Setter führt keinerlei Berechnungen aus[8] und ändert ausschließlich die direkt betroffenen Mitgliedsvariablen des Objekts.

Liefert ein Getter einen Wahrheitswert zurück, können auch die Verben *is* bzw. *has* an Stelle von *get* sinnvoll sein:

Standard Name	***Name mit is, has***
getValidityStatus()	*isValid()*
getOwnerFlag()	*hasOwner()*

STATUSVARIABLEN

Statusvariablen werden verwendet, um einen Programmzustand zu speichern. Statusvariablen werden nicht in Rechenvorgängen, sondern nahezu ausschließlich in Bedingungen (*if*, *while*, *for* etc.) verwendet.

Die Tatsache, dass eine Variable keinen Wert, sondern einen Zustand repräsentiert, sollte im Namen ausgedrückt werden. Gebräuchliche Namensteile für Zustandsvariable zeigt folgende Tabelle:

Namensteil	***Beispiel***
State	*printerState*
Flags	*formatFlags*

Eine Zustandsvariable kann meist mehrere Zustände annehmen. Oft sieht man dann Abfragen der Art

```
if ( printerState & PRINTER_READY ) ...
```

Obwohl dies gut lesbar ist, sollte man davon Abstand nehmen, da die Abfrage zu sehr auf die Implementierung des Status (nämlich auf einzelne Bits) abgestimmt ist. Genau dies interessiert den Leser meist nicht – er möchte wissen, ob der Drucker bereit ist, und nicht, wie das implementiert ist.

---

7 Hier sind fachliche Berechnungen gemeint. Zulässig ist z.B. eine Prüfung, ob die Mitgliedsvariable, deren Wert geliefert werden soll, überhaupt gültig ist. Ein ungültiger Wert kann z.B. bestehen, wenn noch niemals ein Wert gesetzt wurde.

8 Auch hier sind z.B. Prüfungen, ob der übergebene Wert überhaupt zulässig ist, erlaubt.

Man schreibt also besser

```
if ( isPrinterReady( printerState ) ) ...
```

und implementiert die Funktion *isPrinterReady* entsprechend.

BOOL'SCHE STATUSVARIABLEN

Kann eine Zustandsvariable nur zwei Zustände annehmen, sollte man dies ebenfalls im Namen codieren. Gebräuchlich sind die folgenden Namensteile:

Namensteil	***Beispiel***
is	*printerIsReady*
has	*clientHasId*

Beachten Sie bitte, dass der Namensteil is bzw. has nicht am Anfang des Namens steht. Die Variable heißt also *printerIsReady* und nicht *isPrinterReady*.

- Die Form *printerIsReady* impliziert eine Aussage. „Der Drucker ist bereit" ist ein Status, den man festgestellt hat, und den man nun in einer Variablen speichert.
- Die Form *isPrinterReady* impliziert eine Frage. „Ist der Drucker bereit?" ist eine Frage, die einen Funktionsaufruf meint, der eben diesen Status feststellt.

Eine typische Anweisung könnte dann so aussehen:

```
bool printerIsReady = isPrinterReady( printerState );
```

bzw. wenn der Druckerstatus als eigene Klasse ausgeführt ist:

```
bool printerIsReady = printerState.isPrinterReady();
```

Statusvariable sollen positiv formuliert werden, d.h. sie sollen kein *Not* im Namensteil haben. Etwas wie *printerIsNotReady* führt spätestens dann zu Problemen, wenn eine Negation erforderlich ist:

```
if ( !printerIsNotReady )   // Schlecht!
{
  ... Drucken ...
}
```

STEUERVARIABLE

Häufig vorkommende Statuswerte sind z.B. „Fertig“, „Ende erreicht“, „Fehler aufgetreten“, „Weiterer Durchlauf erforderlich“ oder „Muss noch gemacht (berechnet...) werden“ bzw. „Ist bereits gemacht (berechnet...)“. Diese Werte werden häufig in Schleifenkonstruktionen (z.B. als Endekriterium) verwendet. Für diese immer wiederkehrenden Statustypen sind die folgenden Namensteile angemessen:

Namensteil	***Bedeutung***
Done	Meist in Schleifen mit mehreren Iterationen, wenn das (zusätzliche) Endekriterium innerhalb der Schleife berechnet wird.
Error	Signalisiert, dass ein Fehler aufgetreten ist. Wird meist in *if*-Anweisungen verwendet, um Fehlerbehandlungscode von „normalem“ Code zu trennen.
Found	Meist in Funktionen oder Schleifen verwendet, die etwas absuchen. Dignalisiert, ob das gewünschte Element gefunden wurde, oder nicht
Success	Signalisiert, ob ein Vorgang das gewünschte oder erwartete Ergebnis bringt. Hat nichts mit Zustand *Error* zu tun: ist *Sucess* nicht gesetzt (d.h. war die Berechnung ohne Erfolg) ist das kein Fehler.
Fit	Signalisiert, dass etwas passt oder ausreichend groß ist. Häufig mit Resourcen wie z.B. Speicherplatz verwendet
Equal, Same	Signalisiert Gleichheit.
Busy	Signalisiert, dass derzeit keine Aufträge angenommen werden können. Wird hauptsächlich bei externen Geräten (Drucker), bei mehreren Prozessen oder Threads verwendet.

Die Namen können alleine stehen (d.h. direkt als Variablennamen verwendet werden):

```
bool found = strchr( command, '$' );

bool done = false;
while ( !done )
{
  ...
}
```

In komplexerem Code kann sinnvoll sein, den Namen mit weiteren qualifizierenden Namensteilen zu versehen werden. In diesem Fall steht der Statusteil hinten:

```
bool commandDelimiterFound = strchr( command, '$' );

bool commandEvaluationDone = false;
while ( !commandEvaluationDone )
{
  ...
}
```

Steuervariable sollen positiv formuliert werden, also kein *Not* im Namen führen, da sonst Negationen unlogisch werden:

```
if ( !commandEvaluationNotDone )   // schlecht!
{
  ... Kommando ist ausgewertet ...
}
```

BESONDERHEITEN BEI AUFZÄHLUNGEN

Aufzählungen bilden keinen eigenen Namensbereich. Die dort definierten Konstanten sind auch im umliegenden Gültigkeitsbereich sichtbar. Es empfiehlt sich daher, die Aufzählungskonstanten zu qualifizieren. Dazu haben sich zwei oder drei vorangestellte Buchstaben, die sich aus dem Namen der Aufzählung ableiten lassen, bewährt.

Beispiele:

```
enum PrinterState
{
  psReady
, psBusy
, psWarmingUp
, psCoverOpen
, psNoPower
};

enum WrkPrcsState
{
  wpReady
, wpBusy
, wpDied
};
```

UNGEEIGNETE NAMENSTEILE

Nicht alle Silben oder Begriffe eignen sich gleich gut zur Bildung von Namen. Wenig geeignet sind normalerweise

- ***Zahlen***. Variablen, die Zahlen enthalten, zeigen meist nur, dass der Entwickler nicht ausreichend nachgedacht hat. Code wie z.B.

  ```
  void doIt( const Person1& p1 )
  {
    Person2 ps2 = p1;
    ...
  }
  ```

  zeugt von solchen Nachlässigkeiten. Ausnahmen bestätigen diese Regel:

  ```
  int sum( int value1, inr value2 )
  {
    return value1 + value2;
  }
  ```

Oft falsch geschriebene Begriffe	
absense	prefered
acummulate	reciept
acsend	defferred
calender	

Insbesondere wenn deutsche Programmierer englische Namen verwenden sollen, werden diese Begriffe häufig falsch geschrieben. Englische Wörterbücher oder Lehrbücher enthalten oft Listen mit solchen Problembegriffen.

REGELN FÜR DEN TECHNISCHEN NAMENSTEIL

Technische Aspekte eines Namens beziehen sich auf die Implementierung. Man kann z.B. im Namen codieren, ob es sich bei dem Programmobjekt um einen Typ (z.B. eine Klasse), ein Interface, eine globale Variable, eine lokale Variable, eine Konstante oder ein Makro handelt. Diese Information hat nichts mit der fachlichen Bedeutung des Programmobjekts zu tun!

GENERELLE KENNZEICHNUNG

Die technischen Kennzeichen eines Namens können voran- oder nach gestellt werden. Folgende Kennungen sind für voran gestellte Kennzeichen sinnvoll:

Kennung	Bedeutung	Beispiel
C	Klasse	*class CAccount;*
I	Interface	*class IGlobalRegister;*
T	Template-Parameter	*template<tapename TTarget> class CProxy;*
N	Namensbereiche	*using namespace NFastHash;*
M oder L	Implementierungsklasse	*class MGridGeometry*
m	Mitgliedsvariable einer Klasse	*class CPerson* *{* * string mName;* *};*
s	statische Variable, auch bei Klassenmitgliedern	*class CPerson* *{* * static int sDebugLevel;* *};*
a	Parameter (Argument) einer Funktion	*void f(const string& aName);*
c	Konstante	*const int cMaxEntries = 10;*
g	Globale Variable	*OperatingMode gOperatingMode;*

Da die meisten C++-Klassen direkt Klassen im problem domain abbilden, würden die meisten C++-Klassen ein voran gestelltes C erhalten. Dies wird von vielen Entwicklern als überflüssig an gesehen, so dass sie direkt z.B.

```
class Person;
```

schreiben.

Zur Unterscheidung wird dann ein Interface, dass in C++ ja ebenfalls durch eine Klasse abgebildet wird, durch ein nachgestelltes *If* gekennzeichnet:

```
class GlobalRegisterIf;
```

Ebenso könnte man auf das Kennzeichen *N* bei Namensbereichen verzichten, da Namensbereich-Bezeicher nur in *namespace*- bzw. *using*- Anweisungen vor kommen können, und der Kontext damit eindeutig klar ist. Die Praxis zeigt jedoch, dass es häufig zu Namenskonflikten zwischen Namensbereichen und Klassen kommt. Da Klassen wesentlich häufiger vorkommen als Namensbereiche, werden Namensbereiche zusätzlich gekennzeichnet. Analog zu Interfaces kann dies auch durch ein nachgestelltes *Ns* erfolgen:

```
namespace FastHashNs { ...
```

Implementierungsklassen werden verwendet, um „normale“ Klassen zu strukturieren, z.B. dann, wenn eine Klasse zu groß werden würde. Meist handelt es sich um lokale Klassen, die nicht nach außen sichtbar sind. Es sind reine Hilfskonstruktionen, um die Komplexität zu reduzieren. Als Kennzeichen kommt *I* nicht in Frage, statt dessen hat sich *M* (manchmal auch *L*) bewährt. Auch hier ist eine Nachstellung des Kennzeichens (dann *Imp*) häufig:

```
class GridGeometryImp;
```

Folgende Tabelle zeigt gängige Kennungen bei nach gestelltem Kennzeichen:

Kennung	***Bedeutung***	***Beispiel***
	Klasse (kein Kennzeichen)	*class Account;*
If	Interface	*class GlobalRegisterIf;*
Tpl	Template-Parameter	*template<typename TargetTpl> class Proxy;*
Ns	Namensbereich	*using namespace FastHashNs;*
Imp	Implementierungsklasse	*class GridGeometryImp;*

KENNZEICHNUNG DES TYPS

Es kann helfen, einige Eigenschaften eines Typs, die nicht direkt aus dem Namen hervor gehen, durch Kennzeichen zu verdeutlichen. Folgende Tabelle gibt eine Übersicht sinnvoller Kennzeichen:

Kennung	***Bedeutung***	***Beispiel***
p	Ein Zeiger auf ein Programmobjekt	*pPerson*
n	Eine Anzahl	*nPersons*
i	Ein Index (meist in ein Feld)	*pPerson = persons[iPerson];*
h	Ein Handle	*hWnd*

Manchmal findet man auch die folgenden Kennzeichen:

Kennung	***Bedeutung***	***Beispiel***
c, ch	Einzelne Zeichen	*chCommandDelimiter;*
s	Zeichenkette (string)	*sCommand;*

Diese sind jedoch weniger sinnvoll, da der Datentyp normalerweise aus der Bedeutung des Namens hervorgeht.

SINGULAR / PLURAL

Bezeichnet ein Name genau ein Objekt, sollte er im Singular stehen. Sind mehrere Objekte gemeint (z.B. bei einem Feld), oder handelt es sich um eine Anzahl, ist der Plural zu verwenden:

```
int nPersons = persons.getSize();   // Eine Anzahl ist gemeint
Person* person = persons[ i ];      // Ein Feld von Zeigern
```

Für Aufzählungstypen ist der Singular zu verwenden:

```
enum Color
{
  clrRed
, clrBlue
};
```

Dies sieht gewöhnungsbedürftig aus, wird jedoch klarer, wenn man Variable bildet:

```
Color color;  // repräsentiert einen Wert
```

Im Gegensatz zu einem richtigen Container speichert *color* genau einen Wert, und muss deshalb im Singular stehen.

GROSS / KLEINSCHREIBUNG

Die Groß- Kleinschreibung des Anfangsbuchstabens bzw. des ganzen Wortes codiert folgende Information:

- ***Anfangsbuchstabe groß***: es handelt sich um einen selbstdefinierten Typ (meist eine Klasse, eine Aufzählung, ein Feld etc).

```
class Person;
enum Color { ... };
```

- ***Anfangsbuchstabe klein***: es handelt sich um eine Variable oder eine Konstante.

```
Person person;
Color color;
```

- ***Alles Groß***: Es handelt sich um ein Makro.

```
MIN
MAX
TRACE( ... )
```

WHITESPACE

Mit *Whitespace* wird allgemein jeglicher Leerraum im Quelltext bezeichnet. Dazu gehören natürlich Leerzeichen und Tabulatoren (*horizontal whitespace*) , aber auch Leerzeilen (*vertical whitespace*).

Weiterhin gibt es noch den Begriff *Semi-Whitespace*. Er bezeichnet alle Zeichen, die bei einem Ausdruck wenig Druckerschwärze benötigen, und deshalb manchmal kaum von richtigem Whitespace zu unterscheiden sind. Zu den Semi-Whitespace-Zeichen gehören vor allem Punkt, Komma und Unterstrich[9].

AUFGABE VON WHITESPACE

Whitespace soll eingesetzt werden, um Quelltext lesbarer zu gestalten. Horizontaler Whitespace dient dazu, Namen optisch voneinander zu trennen und so die Namen leichter erfassbar zu machen. Verwendet man innerhalb von Namen keinen Semi-Whitespace, bilden Namen optische Einheiten, die die visuelle Erfassung weiter unterstützen. Whitespace am Anfang von Codezeilen bewirkt eine Einrückung, mit der man gut die Programmstruktur visualisieren kann.

Vertical Whitespace soll verwendet werden, um logisch zusammen gehörige Anweisungen zu optischen Blöcken zu gruppieren. Zusammen mit Kommentaren vor einem Block erhält man optimale Unterstützung der Programmstruktur durch die Optik.

[9] Bei älteren Druckern mit schlechtem Farbband kam es früher gerne vor, dass Unterstriche in Programmlistings überhaupt nicht sichtbar waren. Dies war ein weiterer Grund, Unterstriche nicht zu verwenden. Diese Aversion hat sich bei vielen Programmierern bis heute gehalten.

HORIZONTAL WHITESPACE ZUR TRENNUNG VON SYMBOLEN

Um Funktionen, Variable, Rechenzeichen etc. von der Optik als einzelne Einheiten erscheinen zu lassen, werden die Namen durch whitespace voneinander getrennt.

- Grundsätzlich steht ein Leerzeichen vor und nach jedem Namen, Operator etc.
- vor einem Komma steht kein Leerzeichen, d.h. das Komma in Parameterlisten folgt sofort nach dem Parameter.
- vor der öffnenden Klammer einer Parameterliste steht kein Leerzeichen, d.h. die öffnende Klammer folgt sofort dem Funktionsnamen.
- Increment- und Decrement-Operator stehen direkt bei ihrer Variablen (ohne Leerzeichen)

Beispiele:

Formatierung	***Kommentar***
int calcProduct(int, int);	Parameter werden durch Leerzeichen separiert. Kommata in Parameterlisten folgen direkt ihrem Parameter.
for(int i = 0; i < size; i++)	Increment-Operator folgt direkt der Variablen
contents[ofs + 1]	Leerzeichen steht nach öffnender und vor schließender Klammer

HORIZINTAL WHITESPACE ZUR EINRÜCKUNG VON PROGRAMMZEILEN

Die Einrückung von Programmzeilen wird verwendet, um die Programmstruktur zu visualisieren. In der Praxis sind die folgenden beiden Stile (bzw. Variationen davon) verbreitet:

- ***K&R-Stil***. Dieser Stil ist nach den Autoren Kernighan und Ritchie benannt, die das berühmte Buch *Die C-Programmiersprache* (1978) geschrieben haben.

  ```
  if ( Bedingung ) {
    statement;
    ...
    statement;
  }
  ```

  Alle Beispiele in diesem Buch sind so formatiert. Der Stil hat lange Zeit die Softwarenetwicklung in C geprägt.

  Die wesentliche Eigenschaft des K&R-Stils ist die öffnende Klammer in der gleichen Zeile wie die Bedingung. Dadurch wird gegenüber den anderen Stile eine Zeile eingespart. Dies war in Zeiten, in denen Bildschirme 24 Zeilen darstellen konnten, wichtig. Dieses Argument hat heute jedoch an Bedeutung verloren. Dem gegenüber steht der Nachteil, dass öffnende und schließende Klammer nicht in der gleichen Spalte stehen. In größeren Verschachtelungen kann dies die Lesbarkeit beeinträchtigen.

  Für neu entwickelte Software wird der K&R-Stil praktisch nicht mehr verwendet.

- ***Allman-Stil***. Dieser Stil ist nach Eric Allman benannt, der große Teile der UNIX-Utilies geschrieben hat.

  ```
  if ( Bedingung )
  {
    statement;
    ...
    statement;
  }
  ```

  Dieser Stil braucht eine Zeile mehr als der K&R-Stil, hat aber dafür den Vorteil, dass öffnende und schließende Klammer in der gleichen Spalte stehen.

Praktisch alle heute verwendeten Einrückungsstile sind Variationen des Allman-Stils, da der korrekten vertikalen Ausrichtung der Klammern ein großer Wert bei gemessen wird. Variationsmöglichkeiten gibt es z.B. bei der Einrückungstiefe, sowie bei der Einrückung der Klammern sowie der Anweisungen im Block selber.

Der Original-Allman-Stil verwendet eine Einrückungstiefe von 0 für die Klammern (d.h. die Klammern stehen direkt unter der Bedingung) sowie von 4 für den Block. Ich persönlich verwende die Einrückungen 0/2, die mir auch tiefere Blöcke erlauben, ohne dass ich den Bildschirm horizontal rollen muss.

Ein Spezialfall des Allman-Stils ist der GNU-Stil, der nahezu ausschließlich in GNU-Projekten verwendet wird. Er verwendet Einrückungen vom Typ 4/8 (manchmal auch 2/4):

```
if ( Bedingung )
  {
    statement;
    ...
    statement;
  }
```

Der Allman-Stil wird analog für Klassendefinitionen etc. verwendet:

```
class Point
{

};
```

Eine Ausnahme bilden Namensbereiche. Da die Definition eines Namensbereiches nur global möglich ist, stehen die entsprechenden namespace-Anweisungen in der Datei ganz oben und schließen den gesamten Quellcode einer Datei ein.

Da eine Einrückung in diesem Fall keinen optischen Vorteil bringt, kann man ausnahmsweise

```
namespace NXyz {

...

}
```

schreiben.

TABS VS. SPACES

Zur Einrückung sollen ausschließlich Leerzeichen verwendet werden. Tabulatoren können in jedem Editor und in jedem Druckertreiber anders eingestellt sein. Code, der mit einer anderen Tabeinstellung als bei der Erstellung angezeigt oder gedruckt wird, sieht unbrauchbar aus. Grundsätzlich sollte die visuelle Darstellung des Quellcodes nicht von dem Werkzeug abhängen, mit dem der Code betrachtet oder bearbeitet wird. Die visuellen Aspekte sollen ausschließlich vom Programmierer bestimmt werden!

HORIZONTAL WHITESPACE: VERSCHIEDENES

In diesem Abschnitt betrachten wir einige Einzelfälle, die nicht unter die vorherigen Kapitel passen.

- ***Bedingungen mit Einzelanweisung***. Geschweifte (Block-)klammern sind syntaktisch nur dann erforderlich, wenn mehrere Anweisungen in if- oder else-Teil stehen. Aus Gründen der Optik kann man diese Klammern auch bei Einzelanweisungen schreiben. Der Quellcode liest sich dann konsistenter, weil z.B. nach einer Bedingung *immer* ein (expliziter) Block mit Klammern steht.

```
if ( Bedingung )
{
  statement;
}
```

- ***else-if-Kaskaden***. Kaskaden aus aufeinander folgenden else-if-Blöcken werden nicht gesondert eingerückt. Man schreibt:

```
if ( Bedingung1 )
{
   ...
}

else if ( Bedingung2 )
{
   ...
}

else if ( Bedingung3 )
{
   ...
}
```

Wendet man die normalen Einrückungsregeln an, müsste man dagegen etwas wie

```
if ( Bedingung1 )
{
   ...
}

else
  if ( Bedingung2 )
  {
    ...
  }

  else
    if ( Bedingung3 )
    {
      ...
    }
```

schreiben.

- ***Artifizielle Blöcke***. Man kann Blöcke gut verwenden, um zusammengehörige Anweisungen zu gruppieren. Wäre von der Syntax her eigentlich kein Block erforderlich, spricht man von *artifiziellen Blöcken.*

```
{
   statement;
   ...
   statement;
}

{
   statement;
   ...
   statement;
}
```

Durch die Verwendung der zusätzlichen Blöcke entsteht kein zusätzlicher Overhead, der Compiler optimiert sie einfach weg[10].

[10] Natürlich ist wie immer zu beachten, dass bei Beendigung eines Blocks die Destruktoren lokaler Variable aufgerufen werden. Die schließende Klammer kann also durchaus mit der Ausführung von Anweisungen verbunden sein. Werden artifizielle Blöcke verwendet, erfolgt die Ausführung dieses Codes zu anderen Zeitpunkten als ohne artifizielle Blöcke. Deswegen wird aber nicht mehr Code ausgeführt.

Artifizielle Blöcke sind insbesondere deswegen sinnvoll, weil sie zur Verkürzung der Lebensdauer von Variablen bei tragen. Schreibt man mit einem beliebigen Typ *T*

```
{
   T t;
   ...
   statement;
}
```

kann man sicher sein, dass die lokale Variable t nach der schließenden Klammer nicht mehr existiert[11].

VERTICAL WHITESPACE

Vertikale Zwischenräume werden verwendet, um logisch zusammengehörige Codeteile auch optisch als Block zu präsentieren und die einzelnen Blöcke voneinander zu trennen.

Die Platzierung von Leerzeilen hängt somit wesentlich von der fachlichen Struktur der Aufgabenstellung ab, so dass generelle Regeln nicht angegeben werden können. Grundsätzlich sind natürlich in Klammern eingeschlossene Anweisungsblöcke „logisch zusammengehörend“, woraus die Forderung nach vertical whitespace nach einer schließenden Blockklammer resultiert:

```
if ( Bedingung )
{
  statement;
}

statement;
```

Die Einteilung einer Funktion in logisch voneinander unabhängige Codeteile hat außerdem Auswirkungen auf Dokumentationsfragen (s.u.)

Analoges gilt selbstverständlich für die Definition von Klassen, für die Aufteilung des vertikalen Raumes innerhalb der Klassendefinition, etc. Überall gilt: man sollte versuchen, zusammen gehörige Teile zu identifizieren und diese durch Abtrennung mit vertical whitespace auch als solche kenntlich zu machen.

DOKUMENTATION

Über die Dokumentation von Programmen wird viel geschrieben – kaum ein Stilhandbuch kommt ohne detaillierte Vorschriften aus, wie Funktionen, Klassen, Dateien etc. zu dokumentieren sind. Leider sind viele Vorschriften zu detailliert, die Information wird nicht gebraucht, und damit bei Änderungen oft nicht auf dem aktuellen Stand gehalten. Schlimmer als keine Dokumentation ist jedoch eine falsche oder nicht (mehr) zum Code passende Dokumentation.

[11] Diese Technik ist eine Weiterführung der allgemein anerkannten Regel, Variablen nicht gesammelt am Funktionsanfang zu definieren, sondern erst bei ihrer Verwendung. Dadurch wird der Zusammenhang zwischen Code und Variable deutlicher, die Lebenszeit der Variable beginnt erst, wenn sie wirklich gebraucht wird. Platziert man die Variable noch ein einen artifiziellen Block, begrenzt man die Lebenszeit zusätzlich auch noch nach „hinten“: ihre Lebenszeit wird beendet, sobald sie nicht mehr benötigt wird. Insgesamt wird dadurch die Lebenszeit minimiert.

Insgesamt wird viel zu viel geschrieben. Das folgende Codesegment z.B. wurde von einem Metrikprogramm[12] als „gut kommentiert“ klassifiziert:

```
double calcSqrt( double value )
{
  //wenn der Wert kleiner 0 ist, exception werfen!
  if ( value < 0.0 )
    throw "Argument kleiner 0!";

  double result;           // Deklaration der Ergebnisvariablen
  result = sqrt( value );  // Aufruf der Funktion sqrt
  return result;           // Rückgabe von result
}
```

Aber ist es das wirklich? Die Kommentare sagen nichts anderes, als bereits im Code steht. Der Informationsgewinn ist gleich Null.

Wie sollte man also die Funktion dokumentieren? Die Antwort ist: *gar nicht*. Hier haben wir den Glücksfall, dass der Code so gut verständlich ist, dass es keiner weiteren Erläuterung bedarf. Die Funktion sollte also einfach als

```
double calcSqrt( double value )
{
   if ( value < 0.0 )
    throw "Argument kleiner 0!";

  double result;
  result = sqrt( value );
  return result;
}
```

geschrieben werden. Nur leider erfüllt sie dann die Forderung des Stilhandbuches nicht, das Metriktool deckt dies auf, und der Entwickler wird zu einem Gespräch zum Manager gebeten, der ihn über die Notwendigkeit von Richtlinien und deren Einhaltung belehrt. Dass die Funktion (bzw. der Programmierer) ein ganz anderes, viel wichtigeres Problem hat (nämlich die unnötige Verwendung der Variablen *result*) fällt niemandem auf.

Welche Anforderungen sind also an eine gute Dokumentation zu stellen? Die folgenden Abschnitte geben einige Hinweise.

AUFGABE VON DOKUMENTATION

Inline-Dokumentation soll zum Verständnis eines Codesegments benötigte Information bereit stellen, die nicht direkt aus dem Code hervorgeht.

In dieser grundlegenden Anforderung ist von zwei Arten von Information die Rede:

- Welche Information geht bereits aus dem Code *an sich* hervor?
- Welche Information benötigt ein Leser, um den Code zu verstehen?

Code und Inline-Dokumentation sind zwei grundsätzlich verschiedene Informationstypen. Der Code definiert das tatsächliche Programmverhalten, die in ihm enthaltene Information ist daher immer aktuell und korrekt. Code wird durch den Compiler auf syntaktische Korrektheit geprüft. Inline-Dokumentation dagegen unterliegt keinen Beschränkungen. Sie kann an beliebiger Stelle stehen, beliebigen Inhalt haben und kann nicht formal geprüft werden. Es sollte also klar sein, dass der Code selber die beste Informationsquelle darstellt. Erst in zweiter Linie steht die explizite Dokumentation.

[12] Ein Programm, das Quellcode analysiert und die Erfüllung von formalen Kriterien berechnet. Als Ergebnis stehen Maßzahlen, die den Erfüllungsgrad der Kriterien wieder spiegeln sollen.

Das Ziel besteht also darin, möglichst viel Information im Code selber unter zu bringen, und weniger explizite Inline-Dokumentation zu schreiben. Die Zeit, die man zum Schreiben von Dokumentation verwendet, ist besser in die lesbare Formulierung von Programmcode investiert. Erst wenn dieses Mittel ausgeschöpft ist, sollte man an zusätzliche Dokumentation denken.

Diese Ansicht wiederspricht den traditionellen Vorstellungen über Softwareentwicklung. Eine bekannte Aussage dort ist z.B.: *Eine Funktion, die es nicht wert ist, dokumentiert zu werden, ist es auch nicht wert, ausgeführt zu werden“*. In diesem Handbuch wird genau der umgekehrte Ansatz propagiert: „*Eine Funktion, die dokumentiert werden muss, ist es nicht wert, in einem guten System ausgeführt zu werden“.*

Beide Ansätze bezeichnen natürlich Extrempositionen und dienen insofern nur dazu, das zu Grunde liegende Problem auf zu zeigen. In der Praxis wird man sich irgendwo dazwischen auf der Skala platzieren.

CODE ALS DOKUMENTATION

Ein Qualitätsmerkmal von Code ist die *Lesbarkeit*, d.h. die Eigenschaft, die in ihm vorhandene Information preis zu geben. Je mehr man beim einfachen Lesen von Quellcode versteht, um so besser - um so mehr kann auf explizite Dokumentation verzichtet werden.

WAHL GEEIGNETER NAMEN

Aus dem Namen einer Klasse, einer Funktion oder einer Variablen sollte ihr Verwendungszweck unmittelbar hervorgehen. Der zusätzliche Überlegungsaufwand bzw. Schreibaufwand für korrekte Namen wird durch die dadurch erreichte Erhöhung der Lesbarkeit bei weitem ausgeglichen.

Geeignete Namen stellen eine Verbindung zwischen dem Geschäftsmodell (*problem domain*) und der Implementierung (*implementation domain*) her. Schreibt man z.B.

```
bool patientIsValid = patientName.length() > 0;
```

hat man damit dokumentiert, dass Patienten gültig oder nicht gültig sein können (Information aus dem problem domain), und dass diese Information über einen leeren bzw. nicht leeren Patientennamen codiert wird[13] (implementation domain).

Eine Anweisung wie z.B.

```
if ( patientIsValid )
{
  assignRoom();
}
```

lässt an Klarheit nichts zu wünschen übrig. Dagegen ist

```
if ( patientName.length() > 0 )
{
  assignRoom();
}
```

kommentierungsbedürftig. Es ist überhaupt nicht klar, warum der Patientenname gesetzt sein muss, damit eine Zimmerzuweisung erfolgen kann. Wieso könnte man den Namen nicht später setzen?

Das Thema „Bildung von Namen“ haben wir bereits weiter oben ausführlich behandelt.

[13] Dies ist natürlich keine gute Idee für die Praxis, das Beispiel dient nur zur Erläuterung.

AUFBRECHEN VON ANWEISUNGEN

Komplexe Anweisungen enthalten meist mehrere, oft voneinander unabhängige Teile, die auf Grund der Fähigkeit des Programmierers zur Formulierung in einer einzigen Anweisung auch als solche geschrieben werden. Beispiel:

```
if ( ( r->mStartPage == 0 ) ||
     ( ( r->mStartPage != 0 ) &&
       ( r->mStartPage <= aPrintInfo->m_nCurPage ) &&
       ( ( r->mEndPage >= aPrintInfo->m_nCurPage ) ||
         ( r->mEndPage == 0 ) ) ) )
{
  ...
}
```

Der Programmierer hat zwar versucht, die logische Struktur seiner Abfrage durch Einrückungen deutliche zu machen, trotzdem liest sich die Anweisung spröde. Besser wäre z.B.

```
bool pageIsInRange = r->mStartPage <= aPrintInfo->m_nCurPage
                  && r->mEndPage   >= aPrintInfo->m_nCurPage;

bool printIsOk = r->mStartPage != 0 && ( pageIsInRange || r->mEndPage == 0 );

if ( printIsOk )
{
  ...
}
```

bzw. noch besser

```
bool startPageIsSet   = r->mStartPage != 0;
bool endPageIsSet     = r->mEndPage   != 0;

bool pageIsInRange = r->mStartPage <= aPrintInfo->m_nCurPage
                  && r->mEndPage   >= aPrintInfo->m_nCurPage;


bool printIsOk = startPageIsSet && ( pageIsInRange || !endPageIsSet );
```

Die drei zusätzlichen Variablen belasten das Programm nicht wesentlich. Beachten Sie bitte, wie durch die korrekte Namenswahl der Variablen eine Verbindung zwischen *problem domain* und *implementation domain* her gestellt wird: aus der Definition von *startPageIsSet* kann man z.B. erkennen, dass die Anfangsseite des Reports gesetzt oder nicht gesetzt sein kann (Information aus dem Geschäftsmodell, *problem domain*), und wie diese Information im Programm codiert ist (Implementierung, *implementation domain*).

Die Einführung der Variablen *printIsOk* erfüllt noch einen weiteren Zweck. Oft ist man beim Lesen von Programmcode nicht so sehr an den Details der Bedingung interessiert, sondern nur an der Tatsache der Bedingung selber. Die Anweisung

```
if ( printIsOK )
{
  .... was immer dann gemacht werden soll
}
```

lässt den Schluss zu, dass der Code ausgeführt wird, wenn das Drucken möglich ist. *Wie* berechnet wurde, dass das Drucken erlaubt ist, spielt hierbei keine Rolle.

NATÜRLICHE ANORDNUNG VON ANWEISUNGSTEILEN

In Anweisungsteilen, in denen es auf die Reihenfolge der Teile syntaktisch nicht ankommt, sollte man die „variablen“ Teile vor den „festen“ Teilen platzieren.

Dies trifft vor allem Vergleiche: Anstelle von

```
bool pageIsInRange = r->mStartPage <= aPrintInfo->m_nCurPage
                  && r->mEndPage   >= aPrintInfo->m_nCurPage;
```

schreibt man besser

```
bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
                  && aPrintInfo->m_nCurPage <= r->mEndPage;
```

da die aktuelle Seite (*m_nCurPage*) der variable Teil ist, der mit den unveränderlichen Teilen *mStartPage* bzw. *mEndPage* verglichen wird.

Gleiches gilt für den Vergleich mit „richtigen“ Konstanten:

```
const int cMaxIndex = 100;
...
if ( index == cMaxIndex ) ...
```

Der in manchen Richtlinien zu findende Tipp, die Konstante zuerst zu schreiben, um den Schreibfehler

```
if ( cMaxIndex = index ) ...   // Syntaxfehler!
```

finden zu können, steht der Lesbarkeit entgegen. Moderne Compiler belegen die Abfrage

```
if ( index = cMaxIndex ) ...   // sollte Warnung des Compilers ergeben!
```

mit einer Warnung, so dass man auf das Problem hingewiesen wird.

Letztendlich hat diese Regel etwas damit zu tun, wie wir Programmlistings lesen – nämlich von *links oben* nach *links unten*. Wir orientieren uns also am linken Rand der Programmzeilen. Was dort steht bestimmt, ob wir uns den Rest der Anweisung rechts ansehen, oder direkt weiter nach unten springen. Einige Folgerungen aus dieser Beobachtung sind:

- Wichtige Teile einer Anweisung sollen möglichst links stehen. „Wichtig“ ist hier im Sinne von „zum Verständnis der Anweisung wichtig“ zu sehen. Bei Vergleichen von Variablen mit Konstanten ist dies z.B. die Variable, nicht die Konstante.
- Hat eine Anweisung mehrere wichtige Teile, kann ein Umbruch sinnvoll sein, um beide wichtige Teile links zu platzieren.
- Die Einrückung von Codezeilen zur Visualisierung der Struktur darf nicht mit zu vielen Leerzeichen erfolgen, sonst sind beim Lesen zu viele horizontale Sprünge des Auges erforderlich. Die manchmal geforderte Einrückung um 8 Positionen ist definitiv zu viel.

AUFBRECHEN VON FUNKTIONEN

Aus Sicht der Lesbarkeit von Code sollten Funktionen nicht zu lang sein. Manche Stilhandbücher fordern ein Maximum von einer Druckseite (70 Zeilen), andere setzen das Limit bei 100 Zeilen.

Solche Angaben sind zwecklos. Es gibt immer Möglichkeiten, Code enger oder weiter zu schreiben. Ein Programmierer hat Möglichkeiten, durch Zusammenfassungen Code „effizienter“ zu notieren, und so die benötigte Menge Code in seiner Funktion zu platzieren, ohne den

Styleguide zu verletzen. Außerdem gibt es Funktionen, die vom logischen Standpunkt nicht kürzer zu formulieren sind. Eine hocheffiziente Sortierroutine kann z.B. mehr als 300 Zeilen Code haben.

Lange Funktionen bestehen oft aus (relativ) unabhängigen Teilen, die nacheinander oder evtl. auch geschachtelt ausgeführt werden. Die Lesbarkeit kann erhöht werden, wenn man solche Teile

- als separate Funktionen formuliert
- oder zumindest optisch (durch vertikalen whitespace) voneinander separiert.

Welche der beiden Möglichkeiten sinnvoller ist, hängt vom Einzelfall ab.

Zunächst sollte man anstreben, Funktionen in wohl definierte Blöcke auf zu teilen, die sequentiell bzw. innerhalb von Bedingungen ab laufen können. Jeder dieser *chunks* sollte eine klar definierte Aufgabe durch führen. In Ausnahmefällen lässt sich die geforderte klare Definition nicht finden, dann sollte man auch keinen chunk definieren.

Innerhalb von Kontrollstrukturen sind Blöcke häufig an zu treffen, da die Syntax dies fordert. Diese Blöcke sollten so gestaltet werden, dass sie auch chunks bilden, d.h. dass sie eine einzelne, wohldefinierte Aufgabe durchführen.

```
if ( stateIsValid )
{
   ... chunk 1
}

if ( somethingElseIsTheCase )
{
   ... chunk 2
}
else
{
   ... chunk 3
}
```

Weniger bekannt, aber ebenso nützlich, ist die Verwendung von Blöcken, um chunks zu gruppieren:

```
{
   ... chunk 5
}

{
   ... chunk 6
}

{
   ... chunk 7
}
```

Von der Syntax her wären die Blockklammern nicht erforderlich, sie dienen hier der Zusammenfassung von Codeabschnitten zu eigenständigen chunks.

Beachten Sie bitte, dass

- Variablen, die nur innerhalb eines chunks benötigt werden, auch innerhalb der Klammern definiert werden sollen (Begrenzung der Lebenszeit)
- Die chunks optisch durch vertical whitespace separiert werden.

Die Aufteilung einer längeren Funktion in kleinere Funktionen macht nicht immer Sinn. Man könnte z.B. die drei chunks 5, 6, und 7 aus dem letzten Beispiel als eigenständige Funktionen formulieren und in der Rahmenfunktion dann lediglich

```
chunk5();
chunk6();
chunk7();
```

- evtl. mit Parametern - schreiben. Dadurch wird die Lesbarkeit nur dann erhöht, wenn es gelingt, für die drei Funktionen aussagekräftige Namen zu finden. Dies ist jedoch gerade bei chunks häufig nicht der Fall. In solchen Fällen ist die Auslagerung in Hilfsfunktionen nicht angeraten.

Lassen sich jedoch gute Namen finden, ist sogar die Formulierung von einzelnen Anweisungen als Funktion sinnvoll. So berechnet die Anweisung

```
bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
                  && aPrintInfo->m_nCurPage <= r->mEndPage;
```

ob ein Wert zwischen zwei anderen liegt. Dies wird noch klarer, wenn man die Funktionalität „liegt zwischen zwei Werten“ als eigene Funktion formuliert und dann schreibt:

```
bool pageIsInRange =
  isInRange( aPrintInfo->m_nCurPage, r->mStartPage, r->mEndPage );
```

Die Funktion *isInRange* wird sicher häufiger gebraucht, bereits allein deshalb ist eine Formulierung als separate Funktion angezeigt. Der Hauptvorteil ist jedoch, dass der Namen *isInRange* mehr aussagt als die explizite Formulierung mit zwei Vergleichen.

Grundsätzlich kann man also sagen, dass Code immer dann als eigene Funktion formuliert werden soll, wenn sich ein guter Name (möglichst aus dem problem domain) dafür finden lässt. Dies gilt auch dann, wenn die Funktion nur an einer einzigen Stelle auf gerufen wird.

Man sollte die Entscheidung, eine solche Hilfsfunktion zu verwenden, im Namen der Funktion codieren. Beispiele:

Namensteil	***Beispiel***	***Allgemeine Komponente***
calc	*calcIndexFromString*	Berechnung eines Wertes, oft *bool* oder *int*. Im Falle einer Mitgliedsfunktion oft *const*
do	*doInternalBufferResize*	Veränderung von Daten
help	*helpBufferResize*	unspezifizierte Hilfsfunktion

Sind diese Hilfsfunktionen Mitglieder einer Klasse, werden sie privat deklariert.

EXPLIZITE DOKUMENTATION

Trotz Bemühung um inhärent lesbaren Code gibt es natürlich in bestimmten Situationen immer noch Bedarf für explizite Dokumentation. Wir beschränken uns hier auf die Inline-Dokumentation, d.h. Information, die in Form von Kommentarzeilen im Programm selber steht.

PLATZIERUNG VON KOMMENTAREN

Inline-Dokumentation sollte immer möglichst nahe bei den betroffenen Anweisungen stehen. Zu Dokumentationszwecken verwenden wir ausschließlich C++ - Zeilenkommentare, die entweder in einer eigenen Zeile vor den zu kommentierenden Programmteilen stehen oder als Endekommentare in der gleichen Zeile wie die Anweisung. Kommentare am Anfang bzw. Ende einer Datei sollen nicht verwendet werden, um Programmkonstrukte zu dokumentieren.

Beispiel für einen Zeilenkommentar:

```
//-- Absolute Position und Abstand, jeweils in px
//
int xPos,   yPos;
int xDelta, yDelta;
```

Beispiel für einen Zeilenendekommentar:

```
int xPos,   yPos;        // Absolute Position in px
int xDelta, yDelta;      // Abstand in px
```

ZUSAMMENFASSUNGEN

Funktionen, Klassen, aber auch chunks können von einer kurzen Zusammenfassung ihrer Aufgabe bzw. Funktionalität profitieren. Diese soll jedoch nicht die Implementierung wiederholen (implementation domain), sondern soll die Aufgabe mit Worten aus dem problem domain beschreiben.

Nicht besonders sinnvoll ist der Kommentar in diesem Codesegment:

```
//-- wir berechnen, of m_nCurPage größer oder gleich als mStartPage
//   sowie kleiner oder gleich mEndPage ist. Wir berücksichtigen,
//   dass mStartPage als auch mEndPage 0 sein können.
//   Im Erfogsfall wird doPreview ausgeführt.
//
bool startPageIsSet   = r->mStartPage != 0;
bool endPageIsSet     = r->mEndPage   != 0;

bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
                  && aPrintInfo->m_nCurPage <= r->mEndPage;

bool printIsOk = startPageIsSet && ( pageIsInRange || !endPageIsSet );

if ( printIsOk )
{
  doPreview();
}
```

Der zusätzliche Kommentar sagt nicht mehr aus, als man bereits aus dem Codes entnehmen kann. Besser wäre z.B.

```
//-- die preview der aktuelle Seite wird nur angezeigt,
//   wenn die Seitennummer passt!
//
bool startPageIsSet   = r->mStartPage != 0;
bool endPageIsSet     = r->mEndPage   != 0;

bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
                  && aPrintInfo->m_nCurPage <= r->mEndPage;

bool printIsOk = startPageIsSet && ( pageIsInRange || !endPageIsSet );

if ( printIsOk )
{
  doPreview();
}
```

Auch hier sagt der Kommentar eigentlich nichts, was nicht auch im Code steht. Aber er sagt es auf eine Weise, die nicht *wiederholt*, sondern *zusammen fasst*. Ein Leser sieht im Kommentar die Phrasen *„preview“*, *„aktuelle Seite“* und *„wenn Seitenummer passt“*. Er kann daraus erkennen, mit welchen Dingen sich der nachfolgende Abschnitt befassen wird – und damit, ob er ihn genauer lesen muss, oder gleich zum nächsten springt.

Insbesondere im Zusammenhang mit chunks ist die zusammen fassende Dokumentation sinnvoll:

```
//-- Einlesen der Parameter (XML-Datei)
//
{
  ...
}

//-- Gültigkeitsprüfung aller Parameter
//
{
  ...
}

//-- Broadcast-message an alle Programmteile senden
//   note: DOM ist globale Variable und kann von allen
//   direkt verwendet werden
//
{
  ...
}
```

Die Platzierung der Kommentarzeilen außerhalb der Klammern signalisiert, dass sich der Kommentar auf den gesamten chunk bezieht.

Analoge Überlegungen gelten natürlich auch für Klassen. Etwas wie

```
//--- Klasse für Listenelemente
//
struct ListElem
{
  ...
};
```

zu schreiben ist sinnlos. Besser wäre z.B.

```
//--- Basisklasse für Objekte, die in linearen Listen
//    (template <> class List) verwaltet werden sollen
//    Klassen müssen von ListeElem ableiten.
//
struct ListElem
{
  ...
};
```

STATUSANGABEN

An bestimmten Punkten innerhalb einer Funktion sind oft bestimmte Bedingungen erfüllt. Meist war es der Zweck vorangehender chunks, solche Bedingungen sicher zu stellen, da sie zur korrekten Funktion nachfolgender chunks erforderlich sind.

Diese Bedingungen sollen dokumentiert werden:

```
//-- Einlesen der Parameter (XML-Datei)
//
{
  ...
}

//-- Hier haben wir ein komplettes DOM
//   Gültigkeitsprüfung aller Parameter
//
{
  ...
}

//-- alle Parameter sind validiert und können verwendet werden
//   Broadcast-message an alle Programmteile senden
//   note: DOM ist globale Variable und kann von allen
//   direkt verwendet werden
//
{
  ...
}
```

ZUSÄTZLICHE INFORMATIONEN

Während der Implementierung eines Systems entsteht eine Vielzahl von Informationen, die sich nicht direkt im Quellcode wieder spiegeln. Obwohl der Quellcode vielleicht leicht lesbar ist, wird nicht deutlich, *warum* ein bestimmter Ansatz gewählt wurde. Die Dokumentation zusätzlicher Informationen ist immer dann sinnvoll, wenn nicht „das Offensichtliche“ passiert. Liest man z.B. etwas wie

```
double calcCutoffValue( double aHighValue )
{
   double limit = mySqrt( aHighValue ); // eigene Funktion für Quadratwurzel

   ...
}
```

weiß man zwar sofort, was hier passiert, nicht jedoch, warum der Programmierer nicht die übliche Funktion *sqrt* aus der C++ Standardbibliothek zur Bildung der Quadratwurzel verwendet hat.

Besser wäre z.B.

```
double calcCutoffValue( double aHighValue )
{
   //-- die Standardfunktion sqrt ist zu langsam. Wir verwenden
   //   daher eine schnelle, näherungsweise Implementierung mit
   //   ausreichender Genauigkeit
   //
   double limit = mySqrt( aHighValue );

   ...
}
```

Etwas nicht offensichtliches passiert auch in dieser Schleife:

```
for ( int i = 1; i <= array.getSize(); i++ )
{
  ...
}
```

Hier ist nicht auf den ersten Blick zu erkennen, warum die Schleifenvariablen nicht wie üblich bei 0 beginnt und bis zur Größe des Containers läuft. Wenn es sich nicht um einen Fehler handelt, sollte man das auch hinschreiben – der nächste Leser braucht diese Überlegung dann nicht mehr an zu stellen.

```
for ( int i = 1; i <= array.getSize(); i++ ) // note: Index von 1 bis size!
{
  ...
}
```

Obwohl keine Begründung angegeben ist, ist der Kommentar sinnvoll: er signalisiert, dass es sich bei der ungewöhnlichen Schleife nicht um einen Fehler handelt.

INTERFACES

Die letzten Abschnitte haben sich mit Gestalt und Dokumentation von Implementierungen befasst. Etwas anders liegt der Fall bei Schnittstellen (*interfaces*).

Eine Schnittstelle bietet Funktionalität nach außen an, meist ohne näher darauf ein zu gehen, wie diese erbracht wird. Im Gegensatz zur Implementierung kann man bei der Dokumentation einer Schnittstelle nicht voraus setzen, dass ein Benutzer die Funktion des Moduls, der Klasse etc. kennt. Die Dokumentation wird daher umfangreicher sein.

Eine Schnittstellendokumentation sollte zumindest die folgenden Punkte berücksichtigen:

- angebotener Service (Funktionalität, Zweck der Klasse etc)
- Vorbedingungen (was muss erfüllt sein, damit der Service in Anspruch genommen werden kann?
- Nachbedingungen (wie ist die Lage nach Serviceerbringung?)
- Fehlersituationen (Was passiert bei Fehlern? Returncodes, Ausnahmen?)
- Sonstige Informationen, die zum Ablauf erforderlich sind

Dies bedeutet nicht, dass alle diese Punkte für jede Funktion, jede Klasse etc. notiert werden müssen. Die Dokumentation sollte jedoch *insgesamt* die genannten Punkte berücksichtigen.

Beispiel (aus einer Headerdatei):

```
//======================== export ================================

//-- PubSub dient zur Kommunikation zwischen zwei Klassen,
//   die sich nicht kennen mit Hilfe eines Interface I.
//   P (Publisher): speichert I*
//   S (Subscriber): leitet von I ab
//   P benachrichtigt S durch Aufruf der Funktion über
//     template callSubscribers
//
//   geht subscriber out of scope, bemerkt das der Publisher nicht.
//     Dann darf dieser Aufruf nicht erfolgen.
//   Subscriber werden daher in globaler Liste (gIfList)gehalten,
//     so dass Existenzprüfung möglich ist.
//
//   insgesamt keine Vor/Nachbedingungen, kein Fehlersituationen

namespace QLib {
namespace PubSub {

/*****************************************************************/
/*                                                               */
/*  struct InterfaceBase                                         */
/*                                                               */
/*****************************************************************/

//-- dies ist das I: Subscriber leitet ab,
//   Publisher definiert Container mit I*
//
struct InterfaceBase
{
  InterfaceBase();  // registriert bei gIfList
  ~InterfaceBase(); // deregistriert

}; // InterfaceBase
```

```
/*****************************************************************/
/*                                                               */
/*  globale Funktionen                                           */
/*                                                               */
/*****************************************************************/

//-- ruft alle Targets im Container auf und übergibt 0-3 Argumente
//    (per Referenz)
//   sowohl Interface als auch die Argumente können beliebig sein
//   Note: Da es Überladen aufGrund von Template-Parametern nicht gibt,
//    brauchen wir unterschiedliche Funktionen für 0,1, 2 und 3-
//    Argument-Templates

template<typename TIf>
void callSubscribers0
(
  const VectorI<TIf>&      // Liste mit Subscribern
, void (TIf::*)()          // auf zu rufende Funktion (0-Argument-Version)
);

template<typename TIf, typename TArg>
void callSubscribers1
(
  const VectorI<TIf>&      // Liste mit Subscribern
, void (TIf::*)( TArg& )   // auf zu rufende Funktion (1-Argument-Version)
,  TArg&                   // das Argument
);

template<typename TIf, typename TArg1, typename TArg2>
void callSubscribers2
(
  const VectorI<TIf>&
, void (TIf::*)( TArg1&, TArg2& )
, TArg1&
, TArg2&
);

... etc ...

}} // namespaces PubSub, QLib
```

Für den Aufruf der Funktionen gibt es keine Vor- oder Nachbedingungen etc. Diese Tatsache wird notiert, so man weiß, das dies nicht vergessen wurde.

JAVA DOC STYLE

In der Java-Welt werden Formatkonventionen verwendet, die sich maschinell auswerten und automatisch zu einer zusammenhängenden Dokumentation verarbeiten lassen. Für C++ und viele andere Sprachen sind ähnliche Tools auf dem Markt, die das gleiche leisten. Eines der bekanntesten ist wohl Doxygen (www.doxygen.org), das zudem für private Nutzung kostenlos ist.

Folgendes Codesegment zeigt die Dokumentation einer Funktion *multiply* im Javadoc-Style:

```
   /**
        * berechnet das Produkt zweier Zahlen
        * @param aValue1 der erste Multiplikand
        * @param aValue2 der zweite Multiplikand
        * @see divide()
        * @return das Produkt
        */
        double multiply( double aValue1, double aValue2 );
```

Der Vorteil ist die Möglichkeit zur maschinellen Generierung von Dokumentationsdateien (z.B. in HTML), der Nachteil die mangelnde Übersichtlichkeit beim direkten Lesen des Quellcodes.

Eine besser lesbare Version wäre z.B.

```
//-- berechnet das Produkt zweier Zahlen
//
double multiply
(
  double    // erste Multiplikand
, double    // zweite Multiplikand
);
```

Hier steht ist die Dokumentation der Parameter direkt bei den Parametern selber. Eine Wiederholung der Parameter im Kommentarblock ist nicht erforderlich. Beachten Sie bitte, dass hier auf die Parameternamen verzichtet wurde, da ihr Name unwichtig ist.

KOMMENTARE ZU ANDEREN ZWECKEN

Manchmal sind Kommentare zu rein optischen Zwecken sinnvoll. Schreibt man in der Implementierungsdatei vor jede Funktion einen Kommentarkopf, kann man im Quellcode schnell jede Funktion finden.

Beispiel eines Kommentarkopfes für Funktionen:

```
//------------------------------------------------------------------------
//        du2Twips[14]
//
int du2TwipsX( CDC& aDC, int aValDU )
{
  int devUnitsPerInchX   = aDC.GetDeviceCaps(LOGPIXELSX);
  return MulDiv( aValDU, 1440, devUnitsPerInchX );
}

int du2TwipsY( CDC& aDC, int aValDU )
{
  int devUnitsPerInchY   = aDC.GetDeviceCaps(LOGPIXELSY);
  return MulDiv( aValDU, 1440, devUnitsPerInchY );
}
```

Da beide Funktionen zusammen gehören, wurde nur ein Kommentarkopf geschrieben.

[14] *du* steht hier für *device units*, *twip* für *twentieth of a point.*

SCHLÜSSELWÖRTER IN KOMMENTAREN

Es ist sinnvoll, bestimmte Erkenntnisse oder Situationen mit zusätzlichen Schlüsselwörtern zu dekorieren. Dadurch kann ein Leser zuordnen, um was für einen Kommentar es sich handelt. Vor allem aber ist eine Suche nach bestimmten Schlüsselwörtern in allen Dateien des Projekts möglich.

Folgende Schlüsselwörter sind sinnvoll:

Schlüsselwort	***Bedeutung***
TODO	sicherlich das am häufigsten verwendete Schlüsselwort. Es zeigt an, dass der Entwickler an der betreffenden Stelle noch etwas tun muss. Meist fehlt „unwichtige" Funktionalität wie z.B. Fehlerabfragen, Behandlung von Sonderfällen etc.
KLUDGE	nicht ganz sauberer Code. Sollte man besser machen, aber kann auf später verschoben werden.
COMPILER	Spezieller Code, der auf die Befindlichkeiten (Unzulänglichkeiten) des verwendeten Compilers Rücksicht nimmt. Bei Umstellung auf nächste Version des Compilers noch mal ansehen, ob der Code verbessert werden kann.
PERF	(Potentielles) Performance-Problem. Der Programmierer hat hier zunächst eine (einfachere) Lösung gewählt, die funktioniert. Bei späteren Performanceproblemen sollte man zunächst hier nachsehen.

Weitere Schlüsselwörter können nach Bedarf definiert werden.

In der Praxis gibt es häufig das Problem, dass man sich an die genaue Schreibweise eines Schlüsselwortes nicht mehr erinnert. Es hat sich daher bewährt, einem Schlüsselwort grundsätzlich ein Kennzeichen vorzustellen. Wir verwenden hier die Zeichenfolge .:.

Da es sich um Semi-whitespace handelt, stört das Kennzeichen nicht weiter. Beispiel:

```
void calcMaintenanceRequirements( Vehicle& aVehicle )
{
  // .:.TODO vehicle auf Plausibilität prüfen

  ...
}
```

KOMMENTIERUNG VON DATEIEN

Header- und Implementierungsdateien können Kommentare erhalten, die die Dateien (nicht die einzelnen Routinen, Definitionen oder Klassen) als Ganzes beschreiben. Hierzu zählt im wesentlichen Versions- und Historieninformation. Versionsdaten und Änderungshistorie werden normalerweise vom Versionskontrollsystem automatisch eingefügt.

Führt man die Versionshistorie in der Datei selber, sollte diese am Ende der Datei angeordnet werden, und zwar aus zwei Gründen:

- Die Versionshistorie ist normalerweise uninteressant
- Man kann sich schlecht auf Zeilennummern beziehen, wenn bei jeder fachlichen Änderung der Datei zusätzliche Zeilen mit Versionsinformation vorne angefügt werden.

Kommentarzeilen werden außerdem verwendet, um die einzelnen Sektionen der Datei optisch zu trennen.

Folgendes Listing zeigt eine Headerdatei mit Dateikommentaren:

```
#ifndef __QLIB_Q7WorkWnd_H
#define __QLIB_Q7WorkWnd_H

//========================== header ================================
/*
 * $Header: /qLib/q7WorkWnd.h 8     31.01.01 18:22 Aupperle $
 *
 * namespace WorkWnd
 * class WorkWnd::Wnd
 *
 */

//========================== interface dependencies ================

#include "q3Vector.h"
#include "q3Callback.h"
#include "q3EnhancedStyles.h"
#include "q3NamedObject.h"
#include "q6StatusBar.h"


//========================== export ================================

... hier stehen die Deklarationen, Definitionen etc,
    die diese Headerdatei bereit stellt ...

//========================== Misc ==================================

... hier stehen Kommentare, die die ganze Datei betreffen,
    Hyperlinks zu anderen Dokuemnten etc.

//========================== log ===================================
/*
 * $Log:: /qLib/q7WorkWnd.h                                       $
 *
 * 8     31.01.01 18:22 Aupperle
 * ShowHide Interface dazu
 *
 * 7     23.01.01 13:31 Beyer
 * Handling of ID_EDIT_CUT, ID_EDIT_COPY, ID_EDIT_PASTE
 *
 * ... etc. bis Version 1
 */


#endif
```

Die Datei besteht aus den folgenden Sektionen:

- ***Sektion mit Defines***. Diese sind erforderlich, um eine Mehrfach-Einbindung der Headerdatei aus zu schließen.
- ***Header-Sektion***. Sie enthält vor allem Datum und Version der Datei (hier 31.01.01 18:22, 8) sowie den Namen des Bearbeiters, der die letzte Änderung vorgenommen hat (hier Aupperle). Diese Zeile wird vom Konfigurations-Kontrollsystem aktuell gehalten.

  Die nachfolgenden Zeilen geben kurz an, was diese Datei bereit stellt (hier den Namensbereich *WorkWnd* so wie darin die Klasse *Wnd.*
- ***Sektion Interface-Dependencies***. Hier stehen alle Deklarationen und Definitionen, die notwendig sind, um die Headerdatei („das Interface“) übersetzten zu können. Im

Beispiel kommen diese Angaben aus den mit #include ein gebundenen Headerdateien[15].

- ***Sektion Export***. Hier stehen die Deklarationen und Definitionen, die diese Datei exportiert, d.h. für andere bereit stellt. (aus Platzgründen nicht gezeigt).

  Implementierungen von Inline-Funktionen sowie Schablonendefinitionen stehen am Ende dieser Sektion. Es wird meist sinnvoll sein, diese in eine eigene Include-Datei aus zu lagern. Wir verwenden dazu die Endungen *.inl* (für inline-Funktionen) bzw. *.tpl* (für Schablonendefinitionen).

- ***Section Misc***. Die Sektion „Verschiedenes“ (*Miscellaneous*) nimmt das auf, was nicht in die anderen Sektionen passt. Das können z.B. weitere (unspezifische) Kommentare, TODO-Merker etc. sein. Wir verwenden häufig Hyperlinks zu anderen Dokumenten (z.B. zur Designdokumentation).

- ***Sektion Log***. Diese Sektion wird vom Konfigurations-Kontrollsystem geführt. Hier stehen die Kommentare, die beim Einchecken der Datei ins Konfigurations-Kontrollsystem angegeben werden.

DAS BUCH-PARADIGMA

Studien haben ergeben, dass Programme dann besonders lesbar sind, wenn die Dateien nach dem „Buch-Paradigma“ aufgebaut und formatiert werden. Ein Buch besteht aus Titel, Vorwort, Inhaltsverzeichnis, den einzelnen Kapiteln sowie dem Stichwortverzeichnis. Ein Kapitel kann sich weiter in Unterkapitel auf teilen.

Programmcode und Kommentare werden also in eine Form gebracht, die der Aufteilung in einem Buch ähneln.

- Im Vorwort befinden sich allgemeine Kommentare, die das gesamte Programm betreffen. Diese Information wird am Besten nicht in den Programmdateien, sondern in eigenen Textdateien unter gebracht.

- Das Inhaltsverzeichnis listet die Bestandteile des Programms auf. Bis zu welcher Detaillierungstiefe dies gehen soll, muss von Projekt zu Projekt entschieden werden. Auf jeden Fall sollten alle Quellcodedateien auf gelistet sein.

- Die Kapitel entsprechen den Übersetzungseinheiten. Eine Übersetzungseinheit wird vom Compiler als ganzes übersetzt und später vom Linker mit anderen Übersetzungseinheiten zum fertigen Programm gebunden. Normalerweise entspricht eine Übersetzungseinheit einer Quellcodedatei. Die mit *#include* eingebundenen Dateien zählen logisch ebenfalls dazu.

- Innerhalb einer Übersetzungseinheit erfolgt eine weitere Aufteilung z.B. in einzelne Klassen, Routinen etc. Diese Aufteilung soll durch die Optik (d.h. durch Kommentare) unterstützt werden.

- Innerhalb einer Klasse, einer Routine etc. können noch feinere Teilungen vor genommen werden. Klassen sollen in *Interface* und *Implementation*-Abschnitte auf geteilt werden, Routinen in *chunks* (s.o.). Wiederum ist es Aufgabe der Kommentierung, diese feinere Aufteilung optisch geeignet zu visualisieren.

[15] Im Dateinamen dieser Include-Dateien steht q für QLib (der Bibliotheksname) gefolgt von der Hierarchieebene. Die QLib ist streng hiearchisch auf gebaut – höhere Ebenen verwenden Services darunter liegender Ebenen.

- Dem Stichwortverzeichnis entsprechen im Programm Querverweise auf andere Programmobjekte. Eine Angabe von Datei und Zeilennummer wäre ideal, jedoch gibt es derzeit noch keine Entwicklungsumgebung, die z.B. die Zeilennummern der Querverweise automatisch aktualisiert, wenn Quellcode verändert wird.

FORMATIERUNG VON PROGRAMMELEMENTEN

Klassendefinitionen, Funktionsdeklarationen etc. erstrecken sich häufig über mehrere Zeilen. Dies kann in der Länge bzw. Komplexität der Anweisung liegen, oder aber die Aufteilung auf mehrere Zeilen wird bewusst durchgeführt, um einen optischen Effekt zu erreichen.

ANWEISUNGEN MIT ÄHNLICHEN TEILAUSDRÜCKEN

Sind in einer längeren Anweisung mehrere Teilausdrücke gleich oder zumindest ähnlich, sollen diese bündig untereinander geschrieben werden. Anstelle von

```
bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
   && aPrintInfo->m_nCurPage <= r->mEndPage;
```

schreibt man besser

```
bool pageIsInRange = aPrintInfo->m_nCurPage >= r->mStartPage
                  && aPrintInfo->m_nCurPage <= r->mEndPage;
```

Dies erhöht die Lesbarkeit auch dann, wenn die Anweisung prinzipiell in einer Zeile Platz hätte.

VERTIKALE AUSRICHTUNG

Grundsätzlich sollen vertikale Fluchtlinien verwendet werden, um die Erkennung des Wesentlichen zu erleichtern. Anstelle von

```
typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;
typedef const char cchar;
```

schreibt man besser

```
typedef unsigned char        uchar;
typedef unsigned int         uint;
typedef unsigned long        ulong;
typedef const char           cchar;
```

Besser lesbar als

```
struct TypeInfo
{
  cchar* name; // Name der Client-Klasse
  int version; // Version der Client-Klasse
  int id;// Nummerierung der TypeInfo-Objekte
  int size; // Größe der Client-Klasse
  void* (*buildIt)(); // Factory-Funktion
  static TypeInfo* first; // Lineare Liste der TypeInfo-Objekte
  TypeInfo* prev;

}; // TypeInfo
```

ist sicherlich

```
struct TypeInfo
{
  cchar* name;                // Name der Client-Klasse
  int    version;             // Version der Client-Klasse
  int    id;                  // Nummerierung der TypeInfo-Objekte
  int    size;                // Größe der Client-Klasse
  void*  (*buildIt)();        // Factory-Funktion

  static TypeInfo* first;     // Lineare Liste der TypeInfo-Objekte
  TypeInfo* prev;

}; // TypeInfo
```

AUFZÄHLUNGEN

Enumerationen können häufig nicht selbsterklärend gestaltet werden. Sollen die einzelnen Elemente der Aufzählung kommentiert werden, hat sich folgende Anordnung bewährt:

```
enum Accessability
{
  acbAll,           // alles (default)
, acbReadOnly       // nur lesen, nicht ändern
, acbNoRead         // auch nicht lesen
, acbInvisible      // Control ist abgeblendet
};
```

Durch das führende Komma können neue Zeilen hinzu gefügt oder weggenommen werden, ohne die darüber liegende Zeile ebenfalls „anfassen“ zu müssen.

PARAMETERLISTEN

Für Parameterlisten in Funktionsdeklarationen gilt das gleiche wie für Aufzählungen. Hier kommt es auf die Dokumentation der Parameter an:

```
//-- setzt einen Pfad aus seinen Einzelteilen zusammen.
//   Zwischenzeichen (:\. etc) werden korrekt eingefügt,
//    wenn noch nicht vorhanden
//
string makePath
(
  cchar* aDriveName     // mit oder ohne :, kann leer oder 0 sein
, cchar* aDirName       // mit oder ohne führende/anhängende / oder \,
                        // kann 0 sein
, cchar* aFileName      // kann 0 sein
, cchar* aExtension     // mit oder ohne ., kann 0 sein
);
```

FUNKTIONS- UND SCHABLONENKÖPFE

Funktionsköpfe bestehen aus den folgenden drei Teilen: Rückgabewert, Funktionsname sowie Parameterliste. Bei Schablonen kommen noch die Typparameter am Anfang hinzu. Es kann die Lesbarkeit deutlich erhöhen, wenn einzelne oder alle Teile in unterschiedlichen Zeilen geschrieben werden. Grundsätzlich benötigen kurze Elemente (z.B. einfache Rückgabetypen wie *void*, *int*, etc., kurze bzw. leere Parameterlisten) keine neue Zeile, längere jedoch schon. Was „kürzer“ bzw. „länger“ ist, muss man selber entscheiden. Das Schlüsselwort *template* gefolgt von der Liste der Schablonenparameter sollte dagegen immer auf einer eigenen Zeile stehen.

Beispiele:

```
//-- Rückgabetyp auf eingener Zeile, da komplexer Typ
//
Persons::Iterator
Persons::getIterator() const;

//-- template-Anteil auf eigener Zeile
//   Parameter mit Namen, die selbsterklärend sind
//   => keine weitere Doku erforderlich
//
template<typename TIf>
void callSubscribers0( const VectorI<TIf>& aSuscribers, void (TIf::*aFunction)() );
```

```
//-- template-Anteil auf eigener Zeile
//   Parameter auf eigener Zeile
//   Parameter ohne Namen, da explizit dokumentiert
//
template<typename TIf>
void callSubscribers0
(
  const VectorI<TIf>&   // Liste mit Subscribern
, void (TIf::*)()       // auf zu rufende Funktion (0-Argument-Version)
);
```

KLASSEN

Klassen dienen häufig dazu, einen Service nach außen anzubieten. Eine Klasse Directory wird z.B. eine Funktion zum Aufzählen der Dateien in einem Verzeichnis bieten. Die Klasse wird außerdem Hilfsfunktionen zum internen Gebrauch sowie evtl. Datenelemente benötigen.

Diese Klassenmitglieder haben aus Sicht des Designs (problem domain) ganz unterschiedliche Bedeutungen. Hier kommt es hauptsächlich auf die Leistungen an, die die Klasse nach außen anbietet. Wie diese dann implementiert werden, ist eine andere Frage, die später beantwortet wird.

Leider bietet das C++ Klassenmodell keine Möglichkeit, diese Überlegungen aus dem problem domain zu formulieren. Die C++ Klasse ist ein Konstrukt aus dem implementation domain, d.h. hier geht es um Variablen, Speicherplatz und Mitgliedsfunktionen. Eine Klasse sieht technisch nicht anders aus, wenn Mitglieder anstelle von public nun private deklariert sind.

Es ist daher Aufgabe der Formatierung sowie evtl. Dokumentation, die Strukturen aus dem Design auf die Implementierungsebene zu übertragen. Aus Sicht des Designs ist eigentlich ausschließlich das Interface der Klasse interessant, d.h. diejenigen Klassenmitglieder, die nach außen hin sichtbar und damit zugreifbar sind. Diese Klassenmitglieder sind öffentlich deklariert.

Daraus folgt, dass die Klassendefinition in zwei Teile zerfällt: den öffentlichen Teil, der die Schnittstelle nach außen darstellt, und den nicht-öffentlichen Teil, der die Implementierungsdetails enthält. Man kann diese Aufteilung durch Kommentarzeilen verstärken:

```
class Account
{

//-------- Interface ----------------------------------------------------
public:

  bool isValid() const;

  string getNbr()       const;    // ohne führende Nullen
  string getNbrPadded() const;    // mit führenden Nullen
  void   setNbr( const string& ); // mit oder ohne führende Nullen

  ...

//-------- Implementation -----------------------------------------------
protected:

  bool checkNbr() const; // syntaktische Korrektheit

  string mNbr;

  ...

}; // Account
```

Wie die Mitglieder innerhalb der Gruppen zu ordnen sind, muss man fallweise entscheiden. Man kann z.B. alle Getter zusammen fassen, und danach alle Setter notieren. Wir schlagen hier den *datenzentrischen* Ansatz vor, d.h. die zu einem Datenelement (aus dem problem domain!) gehörenden Mitglieder werden zusammen gruppiert.

Normalerweise wird empfohlen, alle Datenelemente zusammen zu fassen und am Ende der Klassendefinition zu notieren:

```
class Account
{

//-------- Implementation -----------------------------------------------
protected:

  //-- Syntaktische Korrektheit
  //
  bool checkNbr()  const;
  bool checkName() const;


private:

  //-- Daten
  string mNbr;   // immer ohne führende Null
  string mName;

  ...

}; // Account
```

Der datenzentrische Ansatz erhöht jedoch auch im Implementierungsteil der Klasse die Lesbarkeit:

```
class Account
{

//-------- Implementation ----------------------------------------------
protected:

  //-- Kontonummer. Checkroutine erforderlich wg. Syntaxvorschriften
  //   immer ohne führende Null
  //
  string mNbr;
  bool checkNbr()  const;
  ... weitere Routinen, die mit Kontonummern zu tun haben


  //-- Kontoname. Checkroutine erforderlich wg. Syntaxvorschriften
  //
  string mName;
  bool checkName() const;
  ... weitere Routinen, die mit Kontonamen zu tun haben


}; // Account
```

Die Gruppierung von Daten und zugehörigen Funktionen zu optischen Einheiten ermöglicht eine einfache Dokumentation des abgebildeten Konzepts aus dem problem domain (hier also z.B. einer Kontonummer mit den zugehörigen Syntaxvorschriften).

Noch deutlicher wird dieser Effekt im folgenden Beispiel:

```
class Vehicle
{

//-------- Implementation ----------------------------------------------
protected:

  //-- Bewegungsvektor (Richtung und Geschwindigkeit)
  //
  //
  double dirX, dirY;  // direction
  double velX, velY;  // velocity
  bool checkNbr
  ... weitere Routinen, die mit Kontonummern zu tun haben


  //-- Kontoname. Checkroutine erforderlich wg. Syntaxvorschriften
  //
  string mName;
  bool checkName() const;
  ... weitere Routinen, die mit Kontonamen zu tun haben


}; // Account
```