Prof. Dr. Gerhard Reinelt
Dr. Marcus Oswald
Dipl.-Math. Hanna Seitz
Institut für Informatik
Universität Heidelberg

http://www.informatik.uni-heidelberg.de/groups/comopt/

Richtlinien und Empfehlungen für das Softwarepraktikum

18. Februar 2008

Dieses Dokument enthält unverbindliche Ratschläge und Empfehlungen, die bei der Einarbeitung in eine Programmiersprache bzw. der Auswahl von Entwicklungswerkzeugen helfen sollen.

Ein wesentlicher Bestandteil des Dokumentes umfasst Richtlinien und Konventionen, deren Einhaltung bei der Programmierung im Rahmen des Softwarepraktikums *obligatorisch* ist. Dies betrifft insbesondere die Abschnitte 3.3 bis einschliesslich 3.7, Kapitel 4 und Kapitel 5.

Hinweise auf Fehler in diesem Dokument, Ergänzungen oder sonstige Anmerkungen sind jederzeit willkommen und zu richten an `Marcus.Oswald@informatik.uni-heidelberg.de`.

Das vorliegende Dokument ist auch elektronisch auf unserer AG Homepage unter teaching/praktikum verfügbar.

Inhaltsverzeichnis

1 Programmiersprachen und Standardliteratur

Im Rahmen des Softwarepraktikums sollte in einer der zwei Programmiersprachen **C** oder **C++** programmiert werden. Die im folgenden aufgeführten Bücher helfen bei der Einarbeitung in die Programmiersprachen und/oder als Nachschlagewerke.

1.1 C

- Der Klassiker "Programmieren in C" von Kernighan und Ritchie [KR90] ist nach wie vor das Standardwerk für C.

1.2 C++

- Als Standardwerk für C++ gilt das von Entwickler Bjarne Stroustrup verfasste Buch "Die C++ Programmiersprache" [Str98].
- Ein sehr empfehlenswertes Buch für den schon etwas geübten C++ Programmierer ist "Effektiv C++ programmieren" von Scott Meyers [Mey99a]. Hier findet man jede Menge guter Tips, die einem helfen, typische Programmierfehler in C++ zu vermeiden. Die Fortsetzung zu diesem Buch heisst "Mehr Effektiv C++ programmieren" [Mey99b].

2 Planung

2.1 Analyse und Entwurf

Bevor man sich an einen Rechner setzt und zu programmieren anfängt, sollte man möglichst genau wissen, was das fertige Programm bezüglich Funktionalität und Bedienung leisten soll (**Analysephase**) und aufbauend darauf entscheiden, welche Datenstrukturen und Algorithmen dazu benötigt werden (**Entwurfsphase**).

Mittels einer sorgfältig durchgeführten Analyse und einem gut durchdachten Entwurf erhöht sich die Wahrscheinlichkeit, dass während der Implementierungsphase keine größeren Änderungen mehr im Entwurf notwendig sind. Der anfängliche Aufwand lohnt sich also letztendlich. Das soll aber nicht bedeuten, dass man von dem ersten Entwurf nicht mehr abweichen darf oder soll. Mittlerweile ist man der Auffassung, dass die Entwicklung von Software natürlicherweise **iterativ** und **inkrementell** abläuft. Das heisst, dass die Implementierung in mehreren verfeinernden Schritten stattfindet und regelmässig der Entwurf hinterfragt und gegebenenfalls modifiziert wird.

Für kleinere Projekte läßt sich die Analyse und der Entwurf relativ schnell bewerkstelligen. Man benötigt dazu in der Regel keine vorgeschriebene Vorgehensweise oder spezielle Notationen. Bei sehr großen Softwareentwicklungsprojekten ist es jedoch ratsam, sich vorher mit bereits erprobten Methoden und gängigen Notationen vertraut zu machen. Einen guten Überblick bietet das "Lehrbuch der Software-Technik" von Balzert [Bal98].

2.2 Algorithmen und Datenstrukturen

Für eine sehr gute und umfassende Einführung in grundlegende Datenstrukturen und Algorithmen empfehlen wir das Buch "Introduction to Algorithms" von Cormen, Leiserson und Rivest [CLR94]. Weitere gute Bücher zu dem Thema sind "Algorithms" von Robert Sedgewick [Sed95] und "Algorithmen und Datenstrukturen" von Ottmann und Widmayer [OW96].

2.3 Entwurfsmuster

Bei der objektorientierten Programmierung treten häufig verschiedene wiederkehrende Problemstellungen auf, für die sich bereits "kluge" Leute Lösungen überlegt haben. Diese abstrakten Lösungen werden auch als **Entwurfsmuster** (**Design Patterns**) bezeichnet. Der Klassiker zu diesem Thema, "Design Patterns: Elements of Reusable Object-Oriented Software", stammt von Gamma et al. [GHJV94]. Wer also elegant und professionell programmieren möchte, der sei auf die Lektüre dieses Buches verwiesen. Für den Einstieg in die objektorientierte Programmierung ist es jedoch nicht geeignet.

3 Implementierung

3.1 Entwicklungserkzeuge

3.1.1 Editor

Die Wahl des Editors ist natürlich Geschmackssache. Man sollte sich das Leben jedoch so einfach wie möglich gestalten und nicht auf Features wie Syntax Highlighting (d.h. syntaktischen Elemente der Programmiersprache werden in verschieden Farben dargestellt) etc. verzichten. Ferner sollte man darauf achten, dass der Editor auf verschiedenen Plattformen verfügbar ist. Aus diesen Gründen empfehlen wir den **GNU Emacs**, einen frei verfügbaren mächtigen Editor, den es nicht nur unter Unix und Linux, sondern auch für Windows gibt. Unter http://www.gnu.org/software/emacs/ gibt es Informationen und Möglichkeiten zum Herunterladen.

3.1.2 Compiler und Debugger

Für C und C++ gibt es von GNU frei verfügbare Compiler (`gcc` und `g++`) unter
http://www.gnu.org/software/gcc/gcc.html Ebenfalls von GNU gibt es auch den passenden Debugger `gdb` unter
http://www.gnu.org/software/gdb/gdb.html Innerhalb von Emacs ist es möglich sowohl einen Übersetzungsvorgang zu starten und bei Fehlern direkt an die betroffene Stelle in der Quelldatei zu gelangen als auch ein Programm zu debuggen (durch Aufruf von `gdb`).

3.1.3 Make

Das Werkzeug **Make** hilft dabei, den Übersetzungsvorgang zu vereinfachen. Dazu ist ein sogenanntes **Makefile** erforderlich, in dem die notwendigen Informationen spezifiziert werden. Zum Beispiel gibt man hier an, welche Quelldateien übersetzt werden sollen und welche Bibliotheken zum Programm gebunden werden müssen. Durch einen Aufruf `make` wird dann das Makefile gelesen und der vollständige Übersetzungsvorgang durchgeführt.

Der Unterschied zwischen make und einem einfachen Skript, in dem die Befehle zur Übersetzung aufgelistet sind, besteht darin, dass in einem Makefile Abhängigkeiten zwischen Dateien spezifiziert werden, sowie Aktionen, die auszuführen sind, um aus einer Datei eine andere zu "machen" (daher der Name make). Beispielsweise gibt man an, wie aus einer Datei `xxx.c` die Datei `xxx.o` gemacht wird. Bei einer Übersetzung wird `xxx.o` nur dann neu erzeugt, wenn das Änderungsdatum von `xxx.c` jünger als das von `xxx.o` ist. So wird das unnötige Übersetzen von Programmteilen, die nicht geändert worden sind, vermieden.

Informationen zu make gibt es unter http://www.gnu.org/software/make/make.html Im Abschnitt 6.1 sind exemplarische Makefiles angegeben, die man für das eigene Projekt benutzen und entsprechend anpassen sollte.

3.1.4 Source Code Versionsverwaltung

Für umfangreichere Projekte (mit mehreren Entwicklern) kann es sinnvoll sein, ältere Versionen der Software aufzubewahren. Auch hierfür gibt es ein unterstützendes frei verfügbares Werkzeug, **GNU CVS**, unter http://www.gnu.org/software/cvs/cvs.html

3.1.5 Entwicklungsumgebungen

Am komfortabelsten läßt sich in der Regel mit sog. **Entwicklungsumgebungen** (**Integrated Development Environment**, **IDE**) arbeiten, die meistens alle obengenannten Entwicklungswerkzeuge in einer einzigen Anwendung vereinigen.

3.1.6 Automatisches Auffinden von Fehlern

Eine Quelle vieler bösartiger Fehler in C und C++ Programmen betrifft das Speichermanagement (Zugriff auf nicht reservierten Speicher, keine Freigabe von Speicher etc.). Ein leistungsfähiges (kommerzielles) Programm, welches derartige Fehler aufspürt, ist **purify** von Rational. Dieses haben wir bei uns installiert, so dass ein regelmässiger "Purify-Prozess" auf unseren Rechnern durchgeführt werden kann.

Für C-Programme gibt es ein Werkzeug namens **lint**, welches streng auf Typenkorrektheit prüft, nicht portable und verdächtige Stellen aufspürt, etc.. Dies wird allerdings nur anhand der Syntax des C-Programmes geprüft. Laufzeitfehler lassen sich somit nicht aufspüren.

3.2 Bibliotheken

Bei der Softwareentwicklung sollte man so oft wie möglich auf bereits vorhandene Softwarebausteine, sog. **Bibliotheken** (**Libraries**), zurückgreifen. Eine Ausnahme ist natürlich, wenn aus Lernzwecken explizit die Implementierung eines bestimmten Algorithmus oder einer bestimmten Datenstruktur gefordert wird.

3.2.1 C++

Für C++ gibt es die **Standard Template Library** (**STL**), die seit 1994 zum Standard von C++ gehört. Diese stellt grundlegende Datenstrukturen und Algorithmen zur Verfügung. Informationen gibt es unter
http://www.cs.rpi.edu/projects/STL/stl-new-sav/stl-new.html Eine Online Dokumentation ist unter
http://www.sgi.com/tech/stl/ verfügbar.

Speziell für Problemstellungen der kombinatorischen Optimierung wurde die "Library of Efficient Data structures and Algorithms", (**LEDA**), in C++ entwickelt. Informationen gibt es unter
http://www.mpi-sb.mpg.de/LEDA/leda.html

Für die plattformunabhängige Programmierung von grafischen Benutzeroberflächen ist die Verwendung der **Qt** Bibliothek zu empfehlen. Informationen dazu gibt es unter
http://www.trolltech.com/products/qt/ Die Bibliothek ist für Unix und Linux frei erhältlich. Soll auf Windows entwickelt oder portiert werden, sind Lizenzgebühren zu entrichten. Mittlerweile gibt es auch für Windows eine nicht-kommerzielle Version von Qt, die jedoch nur für das Microsoft Visual Studion Version 6 kompiliert ist[1].

[1] Stand: 18. Februar 2008

3.3 Code Konventionen

In diesem Abschnitt werden eine ganze Reihe von Konventionen eingeführt, auf deren Einhaltung wir großen Wert legen. Die Vorgabe formeller Konventionen wird durch folgende Gründe motiviert:

- Der größte Teil des Arbeitsaufwandes für Software liegt in der Wartung (ca. 80 %); natürlich nur unter der Voraussetzung, dass sie auch benutzt und weiterentwickelt wird.
- Software wird selten nur von einer Person gepflegt. Zur Lebenszeit einer Software wird diese in der Regel von verschiedenen Entwicklern gepflegt und erweitert. Im Falle des Programmierpraktikums werden *wir* mit der von Ihnen erstellten Software weiterarbeiten.
- Code Konventionen tragen erheblich zur Lesbarkeit des Codes bei und vereinfachen somit die Einarbeitung in fremden Code (oder auch den eigenen Code, den man ein paar Wochen oder Monate nicht mehr angerührt hat).

3.3.1 Strukturierung des Codes in Dateien, Suffixe

C : In C legt man den Code in **Definitionsdateien** (**Headerfiles**) mit Suffix `.h` und **Quelldateien** (**Sourcefiles**) mit Suffix `.c` ab. Jede Quelldatei sollte in der Regel eine Funktion enthalten, die eine größere Aufgabe erledigt sowie eventuell weitere Hilfsfunktionen. Die Bennenung der Quelldatei sollte die bereitgestellte Funktionalität widerspiegeln. In der dazugehörigen gleichlautenden Definitionsdatei werden die Funktionsköpfe und Datenstrukturen aufgeführt.

C++ : In C++ legt man den Code in Definitionsdateien mit Suffix `.hh` und Quelldateien mit Suffix `.cpp` ab. Jede Definitionsdatei sollte die Deklaration einer einzelnen Klasse bzw. eines einzelnen Templates enthalten und den Klassen- bzw. Templatenamen auch als Dateinamen tragen. Im Falle der Deklaration einer Klasse enthält die dazugehörige Quelldatei die Implementierung der Klassenfunktionen. Für Templates wird nur eine `.hh` Datei angelegt, d.h. auch die Implementierung der Klassenfunktionen wird in dieser Datei untergebracht.

3.3.2 Aufbau der Dateien

Allgemein : Jede Datei sollte mit einem einheitlichen **Kopf** beginnen, der die Informationen

- Dateiname
- Kurzbeschreibung
- Autor und Emailadresse
- Copyright
- Erstellungsdatum
- Letzte Änderung

enthält. Ferner sollte im Anschluß an diesen Block ein weiterer Kommentarblock folgen, in dem Änderungen (Wann, Wer, Was) eingetragen werden.

Für eine C++ Quelldatei sieht der Kopf z.B. wie folgt aus:

```
////////////////////////////////////// cpp /////////////////////////////////////
//
//  Module           : ExampleClass.cpp
//  Description      : Example class
//  Author           :
//  Email            :
//  Copyright        :
//  Created on       : Thu Apr 19 17:08:22 2001
//  Last modified by : ahr
//  Last modified on : Fri Feb 28 12:32:35 2003
//  Update count     : 2
//
////////////////////////////////////////////////////////////////////////////////
//
//  Date        Name            Changes/Extensions
//  ----        ----            ------------------
//
////////////////////////////////////////////////////////////////////////////////
```

Im folgenden wird der spezifische Aufbau der Dateien für jede Programmiersprache erläutert. Im Abschnitt 6 befinden sich Beispiele. Diese sollte man sich am besten auf seinem Rechner ablegen und bei Anlegen einer neuen Datei als Vorlage nehmen.

C : Jede Definitionsdatei, z.B. `definition.h` sollte von den **Präprozessoranweisungen**

```
#ifndef DEFINITION_H
#define DEFINITION_H

...

#endif
```

eingerahmt sein, damit mehrfache Einbindungen derselben Definitionsdatei vermieden werden.

Exemplarische Definitions- und Quelldateien für C befinden sich im Abschnitt 6.2.

C++ : Wie bei Definitionsdateien für C sollten bei C++ ebenfalls die entsprechenden Präprozessoranweisungen gemacht werden. Danach folgen eventuell `include` Anweisungen. Die Bestandteile der Klassendeklaration sollten wie folgt geordnet sein:

- Variablen
- öffentliche Methoden
- überschriebene Methoden
- geschützte Methoden
- private Methoden

Jeder Abschnitt wird mit einem entsprechende Kommentarblock gekennzeichnet.

In der Definitionsdatei sollte jede Variable und jede Methode dokumentiert werden. Dies geschieht in einem speziellen Format, so dass aus dieser Dokumentation automatisch HTML-Seiten oder andere Formate generiert werden können. Der Aspekt der Dokumentation von Quellcode wird in Abschnitt 3.4 behandelt.

In einer C++ Quelldatei sollten nach dem Kopf und `include`-Anweisung(en) die implementierten Klassenfunktionen in derselben Reihenfolge wie in der dazugehörigen Definitionsdatei angeordnet werden. Ferner sollten auch die trennenden Kommentarblöcke aus der Definitionsdatei benutzt werden.

Exemplarische Definitions- und Quelldateien für C++ finden sich im Abschnitt 6.3.

3.3.3 Einrücken, Zeilenlänge, Zeilentrennung und ternäre Ausdrücke

Einrücken : Für das Einrücken von Programmzeilen sollten **4** Leerzeichen als Maß benutzt werden. Ein guter Editor läßt sich entsprechend einstellen.

Zeilenlänge : Eine Zeile sollte nicht mehr als **80** Zeichen enthalten, da längere Zeilen in der Darstellung unhandlich sind.

Zeilentrennung : Falls es aufgrund obiger Vorgaben nicht möglich ist, einen Ausdruck in einer Zeile unterzubringen, sollten folgende **Trennregeln** berücksichtigt werden:

1. Trenne nach einem Komma.
2. Trenne vor einem Operator.
3. Bevorzuge Trennungen zwischen gleichwertigen Ausdrücken.
4. Richte den Code der nächsten Zeile an dem Anfang des unterbrochenen Ausdrucks aus.
5. Führen diese Regeln zu einer unschönen Darstellung oder zu Zeilen, die am rechten Rand kleben, dann benutze einfach eine Einrückung von **8** Leerzeichen.

Die folgenden Beispiel veranschaulichen die Regeln:

```
// Regel 1. und 4.
eineMethode(langerParameter1, langerParameter2, langerParameter3,
            langerParameter4, langerParameter5);
```

Im folgenden Beispiel werden zwei Möglichkeiten demonstriert, einen arithmetischen Ausdruck zu trennen:

```
// Regel 2. und 4.
langerName1 = langerName2 * (langerName3 + langerName4
                             - langerName5) + 4 * langerName6; // vermeiden

// Regel 2., 3. und 4.
langerName1 = langerName2 * (langerName3 + langerName4 - langerName5)
              + 4 * langerName6; // vorzuziehen
```

Das folgende Beispiel veranschaulicht die Bedeutung der letzten Regel anhand einer Methoden Deklaration

```
// Normale Trennung (Regel 1. und 4.)
eineMethode(int parameter1, Object parameter2, String parameter3,
            Object parameter4){
    ...
}

// Normale Trennung fuer langen Funktionskopf (Regel 1. und 4.)
private static synchronized langerMethodenName(int parameter1,
                                                Object parameter2,
                                                String parameter3,
                                                Object parameter4){
    ...
}

// Einrueckung um 8 Leerzeichen (Regel 1. und 5.)
private static synchronized langerMethodenName(int parameter1,
        Object parameter2, String parameter3, Object parameter4){
    ...
}
```

Bei `if` Anweisungen sollte grundsätzlich die 5. Regel angewandt werden, da sich dann der Rumpf der Anweisung optisch besser von der Bedingung absetzt.

```
// Diese Einrueckung NICHT verwenden!
if ((bedingung1 && bedingung2)
    || (bedingung3 && bedingung4)
    || !(bedingung5 && bedingung6)){
    anweisung1;
    ...
    anweisungn;
}

// Diese Einrueckung anstattdessen verwenden
if (    (bedingung1 && bedingung2)
        || (bedingung3 && bedingung4)
        || !(bedingung5 && bedingung6)){
    anweisung1;
    ...
    anweisungn;
}
```

Ternäre Ausdrücke : Folgende Formatierungen für **ternäre Ausdrücke** sind akzeptabel:

```
alpha = (einLangerBoolescherAusdruck) ? beta : gamma;

alpha = (einLangerBoolescherAusdruck) ? beta
                                      : gamma;
```

```
alpha = (einLangerBoolescherAusdruck)
        ? beta
        : gamma;
```

3.3.4 Deklarationen

Anordnung : Es sollte nur eine Deklaration pro Zeile gemacht werden. Dies ist übersichtlicher und ermöglicht es, einen Kommentar zur Variable an das Zeilenende zu setzen.

```
// So nicht...
int level, size;

// ... sondern so
int level; // indentation level
int size;  // size of table
```

Initialisierung : Wenn möglich sollte eine Variable, die deklariert wird, sofort initialisiert werden.

Strukturen, Klassen und Interfaces : Für Strukturen (C und C++) und Klassen (C++) gelten die folgenden Formatierungsregeln:

- Die öffnende Klammer "{" wird in dieselbe Zeile wie die Deklarations Anweisung gesetzt.
- Die schließende Klammer "}" steht in einer eigenen Zeile und wird mit dem Anfang der Deklarationsanweisung ausgerichtet. Sie wird gefolgt von einem Kommentar, der Typ und Namen des Objektes enthält. Letzteres ist hilfreich, wenn man lange Deklarationen hat, durch die man navigieren muss.

```
struct Sample {
    int var1;
    int var2;
}; // struct Sample

class Sample extends Object {
    int m_var1;
    int m_var2;

    Sample(int i, int j) {
        m_var1 = i;
        m_var2 = j;
    }

} // class Sample
```

Methoden innerhalb einer Klassen- oder Interfacedeklaration werden durch eine einzelne Leerzeile separiert (siehe auch 3.3.6).

3.3.5 Anweisungen

Einfache Anweisungen : Jede Zeile sollte höchstens eine **einfache Anweisung** enthalten, z.B.

```
firstCounter++;  // Korrekt
secondCounter++; // Korrekt
firstCounter++; secondCounter++; // Vermeiden
```

Zusammengesetzte Anweisungen : Unter einer **zusammengesetzten Anweisung** versteht man eine Liste von einfachen Anweisungen, die in geschweiften Klammern ("{" und "}") eingeschlossen sind. Für diese gelten die folgenden Regeln:

- Die enthalten Anweisungen sollten eine Stufe weiter eingerückt sein als die zusammengesetzte Anweisung.
- Die öffnende Klammer sollte am Ende der Zeile stehen, in der die zusammengesetzte Anweisung beginnt, die schließende Klammer sollte mit dem Anfang der zusammengesetzten Anweisung auf einer Höhe sein.
- Bei Kontrollstrukturen, wie `if-else` oder `for`, sollte selbst eine einzelne Anweisung in Klammern eingeschlossen werden. Dies erleichtert das Einfügen weiterer Anweisungen, da man nicht mehr auf die Klammerung achten muß.

`if`, `if-else`, `if else-if else` Anweisungen : Die Klasse der `if-else` Anweisungen sollte wie folgt formatiert werden:

```
if (condition) {
    statements;
}


if (condition) {
    statements;
}
else {
    statements;
}


if (condition) {
    statements;
}
else {
    if (condition) {
        statements;
    }
    else {
        statements;
    }
}
```

Bei `if-else` sollten bei einzelnen Anweisungen *immer* geschweifte Klammern benutzt werden. Dies hilft Fehler zu vermeiden, die durch nachträgliches Hinzufügen von Anweisungen ohne die entsprechenden Klammern entstehen können.

`for` Anweisungen : Eine `for` Anweisung sollte die folgende Form haben:

```
for (initialization; condition; update) {
    statements;
}
```

Eine leere `for` Anweisung hat die folgende Form:

```
for (initialization; condition; update);
```

Bei Benutzung des Komma Operators im Initialisierungs- oder Aktualisierungsteil sollten nicht mehr als *zwei* Variablen benutzt werden.

`while` Anweisungen : Eine `while` Anweisung sollte die folgende Form haben:

```
while (condition) {
    statements;
}
```

Eine leere `while` Anweisung hat die folgende Form:

```
while (condition);
```

`do-while` Anweisungen : Eine `do-while` Anweisung sollte die folgende Form haben:

```
do {
    statements;
} while (condition);
```

`switch` Anweisungen : Eine `switch` Anweisung sollte die folgende Form haben:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Bei denjenigen `case` Anweisungen, die keine abschließende `break` Anweisung haben, sollte dies explizit mit einem Kommentar kenntlich gemacht werden (hier "falls through").

Jede `switch` Anweisung sollte einen `default` Fall haben. Dieser mag meistens redundant sein, jedoch bietet sich hier die Gelegenheit eine Fehlermeldung auszugeben, wenn keiner der regulären Fälle eingetreten ist.

`try-catch` Anweisungen : Eine `try-catch` Anweisung sollte die folgende Form haben:

```
try {
    statements;
}
catch (ExceptionClass e) {
    statements;
}
```

3.3.6 Leerzeilen und Leerzeichen

In diesem Abschnitt wird beschrieben, wann Leerzeichen und Leerzeilen zur Erhöhung der Lesbarkeit eingesetzt werden sollten.

Leerzeilen : Unter folgenden Umständen sollten *zwei* Leerzeilen verwendet werden:

- In einer *C++ Definitionsdatei* jeweils zwischen Kopf, Präprozessor-Anweisungsblock, `#include`-Anweisung(en), Anfang der Klassendeklaration, den Abschnitten der Klassendeklaration (siehe Abschnitt 3.3.2) und der schließenden Präprozessor-Anweisung. Das Gerüst für eine C++ Quelldatei befindet sich im Abschnitt 6.3.
- In einer *C++ Quelldatei* jeweils zwischen Kopf, `include`-Anweisungsblock und den trennenden Kommentarblöcken.

Unter folgenden Umständen sollte *eine* Leerzeile verwendet werden:

- Zwischen Methodendeklarationen
- Zwischen den lokalen Variablen einer Methode und der ersten Anweisung
- Vor Kommentarzeilen oder -blöcken
- Zwischen logischen Abschnitten innerhalb einer Methode

Leerzeichen : Unter folgenden Umständen sollten Leerzeichen benutzt werden:

- Zwischen einem Schlüsselwort und einer Klammer, wie z.B. in

  ```
  while (true) {
      ...
  }
  ```

  Es sollte jedoch *kein* Leerzeichen zwischen einem Methodennamen und seiner öffnenden Klammer stehen. Mit dieser Regelung läßt sich leichter zwischen Schlüsselwörtern und Methodenaufrufen unterscheiden.

- Hinter jedem Komma einer Liste von Argumenten.
- Zur Trennung von binären Operatoren und deren Operanden, jedoch *nicht* bei unären Operatoren. Beispiel:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
```

- Hinter jedem Semikolon innerhalb eines `for`-Ausdrucks, wie z.B.

```
for (expr1; expr2; expr3)
```

3.3.7 Namenskonvention

Es sind grundsätzlich *englische* Bezeichnungen zu verwenden. Es gibt mehrere Gründe für diese Vorgabe:

- Die Mischung von deutschen Namen oder Funktionen mit englischen Namen oder Funktionen.
- Englische Begriffe sind oft kürzer und prägnanter.
- Der Code wird für "jedermann" lesbar.

Generell sollten aussagekräftige **Bezeichner**[2] benutzt werden, die ruhig etwas länger sein dürfen. Auf keinen Fall sollten Abkürzungen verwendet werden, es sei denn, sie sind bekannter als ihre eigentliche Bedeutung, wie z.B. HTML.

Der Eingabeaufwand ist mit dieser Vorgabe vielleicht ein wenig höher, jedoch gibt es ein paar Möglichkeiten die Eingabe dennoch effizient zu gestalten:

- Am einfachsten kann man durch "Kopieren und Einfügen" (Copy & Paste) von Bezeichnern Zeit sparen und insbesondere auch Tipfehler vermeiden.
- Wem das trotzdem zu lästig ist, der kann beim Eingeben kurze Bezeichner verwenden und diese anschließend mit einem "Ersetze"-Befehl des Editors in lange Namen umwandeln.
- Moderne Entwicklungswerkzeuge besitzen Mechanismen, die in der Lage sind, gültige Methodennamen für eine Klasse bei der Eingabe in einem Menu anzubieten (wie z.B. CodeInsight im JBuilder oder IntelliSense in MS Visual Studio).

Klassen-, Template-, Interface-, Strukturnamen : Hier wird als Bezeichner ein Hauptwort verwendet, dessen erster Buchstabe *groß* geschrieben wird. Bei zusammengesetzten Namen beginnen jeweils die Teilwörter auch mit einem Großbuchstaben, z.B.

```
class Customer
class NetworkController
```

Es sollen *keine* Underscores "_" zur Trennung von zusammengesetzten Wörtern benutzt werden.

[2] Bezeichner sind Namen für Klassen, Templates, Interfaces, Strukturen, Methoden bzw. Funktionen und Variablen.

Variablennamen : Hier sollte ebenfalls ein Hauptwort verwendet werden. Jedoch wird der erste Buchstabe nun *klein* geschrieben. Bei zusammengesetzten Namen beginnen jeweils die Teilwörter auch mit einem Großbuchstaben, z.B.

```
int numberOfNodes;
Vector nodeList;
```

Folgende Präfixe sollen für bestimmte Variablen benutzt werden:

Präfix	Variablenart
`m_`	Datenelemente (Member variables)
`s_`	Statische Variablen
`g_`	Globale Variablen

In der Regel sollten auch für **lokale Variablen** vernünftige Namen gefunden werden. Es gibt jedoch ein paar Ausnahmen, für die Bezeichner mit einem oder wenigen Buchstaben akzeptabel sind:

Typ	Variablenname(n)
`bool`/`boolean`	`b`
`char`	`c`
`double`	`x`, `y`, `z`
`int` (Zähler und Indizes)	`i`, `j`, `k`, `l`
`int` (nicht negativ)	`n`, `m`
`String`	`str`

Methoden-, Funktionsnamen : Hierfür sollten Verben verwendet werden, deren erster Buchstabe *klein* geschrieben wird. Bei zusammengesetzten Namen beginnen jeweils die Teilwörter mit einem Grossbuchstaben.

```
display()
isEmpty()
int numberOfRows;
```

Der Name einer Methode bzw. Funktion sollte deutlich widerspiegeln, was diese macht. Nichtssagende Namen wie z.B. `performServices` oder `handleOutput` sollten unbedingt vermieden werden. Wenn z.B. `handleOutput` durch `formatAndPrintOutput()` ersetzt wird, dann bekommt jeder, der die Methode benutzen will, direkt eine gute Vorstellung, was sie macht.

Konstanten : Hier sollten Bezeichner verwendet werden, die ausschließlich aus Großbuchstaben bestehen. Dabei dürfen ausnahmsweise Underscores für zusammengesetzte Wörter verwendet werden, z.B.

```
const double PI = 3.14;          // C++
```

3.4 Dokumentation der Implementierung

Die Dokumentation des Quellcodes ist ein äußerst wichtiger Bestandteil eines Softwareentwicklungsprojektes. Nur dokumentierter Code kann mit vertretbarem Aufwand erweitert und gewartet werden. Aus diesem Grund ist die sorgfältige Dokumentation des Quellcodes eine wesentliche Anforderung, die wir an jeden Praktikanten stellen. Ferner sollte die Dokumentation in *englisch* erfolgen, damit sie für jedermann lesbar wird.

Der Dokumentationstext wird in Kommentaranweisungen der benutzten Programmiersprache eingebettet. Wir unterscheiden zwei verschiedene Arten von Kommentaren, die **Dokumentations-Kommentare** (**Documentation Comments**) und die **Implementations-Kommentare** (**Implementation Comments**). Die Dokumentations-Kommentare sind für die *Benutzer* gedacht und die Implementations-Kommentare für die *Entwickler*.

3.4.1 Dokumentation für den anwendenden Entwickler

Dokumentations-Kommentare sind dafür gedacht, die Programmierschnittstelle zu den implementierten Klassen, Templates, Interfaces, Strukturen, Methoden und Funktionen zu erläutern. Es geht also darum jedes implementierte Element so zu beschreiben, dass ein Entwickler dieses *benutzen* kann. Details, die für die Benutzung irrelevant sind, werden hier nicht dokumentiert.

Um eine solche Dokumentation optisch aufzubereiten und durch Navigationselemente leichter handhabbar zu machen, gibt es **Dokumentationswerkzeuge**. Diese extrahieren aus dem Quellcode Kommentaranweisungen, die einer gewissen Syntax folgen, und generieren Dokumentation in verschiedenen Formaten, z.B. HTML, LaTeX, PDF, Man Pages.

Für C und C++ Programme gibt es ein adäquates bzw. noch leistungsfähigeres Werkzeug, welches Dokumentations-Kommentare verarbeitet. Dieses heißt **doxygen** und ist unter

`http://www.stack.nl/~dimitri/doxygen/`

http://www.stack.nl/ dimitri/doxygen/ frei erhältlich. Als Ausgabeformate sind HTML, LaTeX, PDF und Man Pages möglich.

Folgende Elemente sollten mit Dokumentations-Kommentaren versehen werden:

- Klassen-, Template-, Interface- und Strukturdeklarationen
- Deklaration eines Datenelementes (Member variable), einer Klasse, Struktur oder eines Templates
- Deklaration einer Methode einer Klasse, eines Templates, einer Struktur (C++) oder eines Interfaces
- Deklaration einer globalen Funktionen

Für Methoden und globale Funktionen sollten die Parameter und der Rückgabewert gesondert erläutert werden. Es gibt spezielle Schlüsselwörter, welche die Dokumentation eines Parameters bzw. Rückgabewertes anzeigen. Diese sind `\param` und `\return` für doxygen.

Die Beispieldateien im Abschnitt 6 sind alle mit den entsprechenden Dokumentationskommentaren versehen.

3.4.2 Dokumentation für den Entwickler der Software

Mit Implementations-Kommentaren werden Details der Implementierung kommentiert, so dass es möglich ist diese zu *verstehen*. Diese sind also für den Entwickler selbst bzw. für andere Entwickler, welche die Software erweitern oder warten.

Implementations-Kommentare für C++ sind von der Form:

```
// Implementations-Kommentar
int ExampleClass::exampleFunction(int i, char j) {
    ...
}
```

bzw.

```
//
// Langer Implementations-Kommentar
//
int ExampleClass::exampleFunction(int i, char j) {
    ...
}
```

für mehrzeiligen Text und werden den zu dokumentierenden Elementen (hier der Implementation der Methode `ExampleClass::exampleFunction`) vorangestellt. In C muß man natürlich die üblichen Kommentaranweisungen verwenden.

Folgende Elemente sollten mit Implementations-Kommentaren versehen werden:

- Implementation der Methode einer Klasse, eines Templates oder einer Struktur (C++)
- Implementation einer globalen Funktionen
- Innerhalb von Methoden jeweils vor erklärungsbedürftigen, komplexeren Anweisungsfolgen

3.5 Programmierkonventionen

3.5.1 Verständlichkeit des Codes

An dieser Stelle sollen ein paar simple Regeln aufgeführt werden, deren Beachtung wesentlich die Verständlichkeit des Codes fördern und außerdem Quellen subtiler Fehler eliminieren können.

- Vermeiden Sie (extrem) verschachtelte Anweisungen, die zur Beschleunigung der Laufzeit dienen sollen. Überlassen Sie dem Compiler das Optimieren in diesem Fall. Zum Beispiel sollte die Anweisung

  ```
  d = (a = b + c) + r;   // Vermeiden!
  ```

  geschrieben werden als

```
a = b + c;
d = a + r;
```

- Verwenden Sie Klammern großzügig, selbst wenn Ihnen die Priorität der Operatoren klar ist. Anderen Entwicklern ist dies vielleicht nicht so klar. Beispiel:

```
if (a == b && c == d)      // Vermeiden!

if ((a == b) && (c == d))  // Anstattdessen verwenden
```

3.5.2 Stabilität und Fehlerbehandlung

Ein Programm wird als **stabil** bezeichnet, wenn es durch beliebige Eingaben oder Bedienung *nicht* zum Absturz gebracht werden kann. Für Bibliotheken muß man den Begriff der Stabilität dahingehend erweitern, dass beliebige Verwendung der enthaltenen Klassen, Templates oder Funktionen nicht zu Abstürzen oder Speicherkorruption führen können.

Allein der Aspekt der Speicherverwaltung ist die Fehlerquelle Nummer eins bei C und C++ Programmen und muß daher sehr sorgfältig behandelt werden. An dieser Stelle würde es jedoch den Rahmen sprengen, detailliert auf diese Problematik einzugehen. Daher empfehlen wir für C++ Programmierer das hervorragende Buch "Effektiv C++ programmieren" von Scott Meyers [Mey99a], in dem u.a. auf die Hauptursachen für Speicherkorruption eingegangen wird.

Sonstige Fehler treten oft durch fehlerhafte Parameter bzw. Werte auf, die übergeben oder berechnet werden. Grundsätzlich sollten die Eingabeparameter von Programmen, Methoden oder Funktionen auf Gültigkeit und Plausibilität überprüft werden.

In C und C++ gibt es dafür das `assert()` Makro, welches für einen gegebenen booleschen Ausdruck prüft, ob dieser wahr ist. Wenn dies nicht der Fall ist, wird das Programm mit einer Meldung, welche die betreffende Datei und Zeilennummer beschreibt, abgebrochen. Der Vorteil dieser Makros besteht darin, dass sie durch das Setzen des Symbols `NDEBUG` ausgeschaltet werden können. Bei oft durchlaufenen Routinen kann sich die Abfrage von Paramtern durchaus in der Laufzeit bemerkbar machen.

Wenn man selbst Fehler abfragt, muss man entscheiden, wie auf einen auftretenden Fehler zu reagieren ist. Es gibt zum Beispiel die Möglichkeiten

- das Programm abzubrechen,
- die aktuelle Routine zu verlassen und zur aufrufenden zurückzukehren,
- einen Standardwert anzunehmen und fortzufahren.

Grundsätzlich sollte *immer* eine Meldung auf die **Standardfehlerausgabe** geschrieben werden, welche die Methode oder Funktion, in der der Fehler aufgetreten ist, sowie eine aussagekräftige Beschreibung des Fehlers liefert.

Für eine Fehlerbehandlung im objektorientierten Sinne sollte der in C++ vorhandene Mechanismus der **Ausnahmebehandlung** (**Exception Handling**) verwendet werden. Der Vorteil hierbei ist, dass sog. **Ausnahmeklassen** (**Exception classes**) benutzt werden, die, im Gegensatz zur herkömmlichen Fehlerspezifikation über Werte, komplexe Informationen zur Fehlerursache enthalten können. Darüber hinaus kann man den Benutzer einer

Methode sogar dazu zwingen, dass er eine ggf. geworfene[3] Ausnahme behandeln muss. Für das obige Beispiel würde die Ausnahmebehandlung wie folgt aussehen:

```
// Funktion der Klasse Vector, die das Element der
// Position index zur\"uckgibt.
// elementCount enthaelt die Anzahl der Element des Vectors.
// Ausnahmeklasse ArrayIndexOutOfBoundsException wird geworfen,
// falls index fehlerhaft ist.
public Object elementAt(int index) throws ArrayIndexOutOfBoundsException{
    if ((index < 0) || (index >= elementCount)) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }

    ...
}
```

Eine Methode, die nun eine Variable vom Typ `Vector` verwendet und mit der Methode `elementAt()` auf eine bestimmte Position zugreifen will, *muss* diesen Aufruf in einer `try-catch` Anweisung durchführen.

```
// Zugriff auf Position i des Vectors aVector
    Object vectorValue = null;
    try {
        vectorValue = aVector.elementAt(i);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println(e);
        statements;    // Fehlerbehandlung
    }
```

3.5.3 Portabilität

Generell sollte Ihr Programm nicht die Eigenschaften der Plattform, des Betriebssystems oder des Compilers ausnutzen.

In C und C++ sollte man ausschließlich gemäß ANSI programmieren. Dies kann man gewährleisten, indem man beim Übersetzen die entsprechende Option einschaltet (bei `gcc` und `g++` lautet die Option `-ansi`). Verwendet man Bibliotheken, sollte man sicherstellen, dass diese auch auf anderen Plattformen verfügbar sind.

3.6 Funktionalität, Bedienbarkeit des Programmes

3.6.1 Eingabe/Ausgabe

Grundsätzlich sollte ein Programm, welches **Parameter von der Kommandozeile** akzeptiert, einen **Hilfetext** ausgeben, wenn es ohne die erforderlichen Parameter oder mit fehlerhaften Parametern aufgerufen wird. Der Hilfetext sollte auch mittels der Option `-?` und `-help` ausgegeben werden und in etwa so aussehen:

[3] Man sagt, dass eine Methode eine Ausnahme **wirft**, wenn sie bei einem Fehler eine Ausnahmeklasse kreiert und diese mit dem Befehl `throw` an die aufrufende Methode weiterreicht.

```
Usage: myApplication [-options] file

where options include:
    -? -help  print this help message
    -x        ...
```

Stellt ein Programm komplexe Einstellmöglichkeiten zur Verfügung, die nicht mehr über die Kommandozeile zu bewerkstelligen sind, so bietet es sich an, eine **Konfigurationsdatei** vorzusehen, in der die Einstellungen mittels eines Editors gemacht werden können. Hat das Programm eine graphische Benutzeroberfläche, so sollten diese Einstellungen auch über einen entsprechenden Dialog möglich sein.

Unter UNIX wird der Name einer Konfigurationsdatei in der Regel aus einem Punkt "." und dem Namen der Applikation gebildet, z.B. `.myApplication`.

Für Programme, die viel Ausgabetext erzeugen (können), sollte es möglich sein, die **Ausführlichkeit der Ausgabe** zu bestimmen. Dies sollte über den Parameter `printlevel` erfolgen. Für den Wert 0 sollte keine Ausgabe erzeugt werden.

3.6.2 Konventionen für GUI

Die Festlegung von Konventionen für graphische Benutzeroberflächen ist eine komplexe Angelegenheit. Deshalb wollen wir hier keine festen Vorgaben machen, sondern nur soweit gehen, zu sagen, dass die graphische Benutzeroberfläche in der Bedienung ähnlich zu denen bekannter Software sein soll.

3.7 Testen

Das Testen einzelner Module bzw. des gesamten Programmpaketes ist ein sehr wichtiger Bestandteil der Softwareentwicklung. Dieser wird jedoch gerne vernachlässigt oder nur sehr rudimentär durchgeführt. Wir wollen daher ein paar Vorgaben zu diesem Thema machen.

Versuchen Sie so früh wie möglich mit dem Testen zu beginnen, um schon während der Entwicklung die meisten Fehler auszumerzen. Schreiben Sie kleine Testprogramme oder Testklassen, welche ihre Programmteile testen.

Generieren Sie eine Reihe von **Testdaten** für ihr Programm. Eine einzige Testdatei ist in der Regel zu wenig und deckt i.a. nicht alle vorhanden Fehler auf. Einige dieser Testdateien sollten auch sehr groß sein, um eventuelle Speicherfehler aufzuspüren.

Oftmals existieren bereits Testdaten für gewisse Probleme. Es lohnt sich diesbezüglich eine Recherche im Internet anzustellen. Wenn man hier fündig geworden ist, so findet man in der Regel auch Ergebnisse anderer Verfahren. Diese bieten einem die Möglichkeit, sein Programm bezüglich Qualität der Lösung und/oder Laufzeit zu vergleichen.

4 Dokumentation des Programmpaketes

Zu einem vollständigen Programmpaket gehört auch eine **Bedienungsanleitung** (**User Manual**), die für einen Benutzer ohne Vorwissen[4] geeignet ist. Diese beschreibt ausschließlich die Bedienung des Programmes. Wie auch die Dokumentation der Implementierung sollte die Bedienungsanleitung in *englischer* Sprache verfasst sein.

Folgende Aspekte gehören in die Bedienungsanleitung:

- Erklärung der Benutzeroberfläche (falls vorhanden). Dazu gehört die Aufteilung des Hauptfensters, die Menuleiste und diverse Unterfenster.
- Erläuterung der einzelnen Komponenten des Programmes, deren Konzept und deren Zusammenspiel mit den übrigen Komponenten.
- Exemplarisches Durchführen mehrerer komplexer Aktion mit detaillierter Erklärung jedes einzelnen Schrittes.

Die Bedienungsanleitung soll mittels LaTeX erfolgen. Insbesondere soll das Paket **Latex2html** benutzt werden[5]. Dieses bietet die Möglichkeit, das LaTeX-Dokument nach HTML zu übersetzen. Ferner lassen sich damit Verweise auf Webseiten in das Dokument einbauen, die dann in der HTML Version anklickbar sind. Das Paket ist unter http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html erhältlich.

[4] Ein sogenannter **DAU** (= dümmster anzunehmender User).
[5] Das vorliegende Dokument benutzt ebenfalls Latex2html.

5 Abgabe des Programmpaketes und Abschlusspräsentation

5.1 Checkliste

Bevor Sie das **Programmpaket** abgeben, sollten sie die folgende Checkliste nochmal durchgehen, um zu überprüfen, ob Sie die von uns gestellten Richtlinien eingehalten haben.

- Haben Sie die *Code Konventionen* (Abschnitt 3.3) eingehalten?
- Sind alle Teile ihrer *Implementierung* sowohl für Anwender als auch Entwickler *dokumentiert* (Abschnitt 3.4) und läßt sich mit Hilfe des Dokumentationswerkzeuges automatisch die Dokumentation erzeugen?
- Ist ihr *Code verständlich* und *gut lesbar* (Abschnitt 3.5.1)? Läuft ihr Programm *stabil*, d.h. werden fehlerhafte Eingaben oder unsachgemäße Bedienung abgefangen und mit aussagekräftigen Meldungen kommentiert (Abschnitt 3.5.2)?
- Läuft ihr Programmpaket auch auf einer anderen Plattform (Abschnitt 3.5.3)?
- Haben Sie geeignete Mechanismen für die *Konfiguration* ihres Programmes und *Eingabe von Daten* bereitgestellt (Abschnitt 3.6.1)?
- Enthält ihr Programmpaket *Testmodule* oder *Testprogramme*, welche die einwandfreie Funktionsweise des Paketes demonstrieren? Haben Sie *Testinstanzen* beschafft oder generiert? (Abschnitt 3.7)
- Haben Sie eine Bedienungsanleitung erstellt (Abschnitt 4)?

Wenn Sie alle diese Fragen mit "Ja" beantworten konnten, dann haben Sie es fast geschafft.

5.2 Abgabe

Das Programmpaket sollte als gepacktes Archiv abgegeben werden. Das **Packen** und **Archivieren** kann mit den Werkzeugen `gzip` und `tar` unter UNIX erfolgen.

Das Programmpaket sollte folgende **Verzeichnisstruktur** haben:

Wurzelverzeichnis : Das **Wurzelverzeichnis** sollte nach ihrem Programmpaket benannt sein. Es enthält eine `README` Datei sowie ein `Makefile` und die Unterverzeichnisse `./doc/`, `./src/`, `./test/` und eventuell `./lib/`.

Die `README` Datei sollte eine kurze Beschreibung ihres Programmpaketes enthalten und erklären, wie das Programm übersetzt und gestartet wird.

Das `Makefile` enthält die Anweisungen zur Übersetzung des Programmes (siehe Abschnitt 3.1.3 und Abschnitt 6.1).

Unterverzeichnis `./doc/` : Dieses Verzeichnis enthält die Dokumentation ihres Programmes.

Im Unterverzeichnis `./doc/manual/` sollten Sie die Bedienungsanleitung ihres Programmes ablegen (siehe Abschnitt 4).

Im Unterverzeichnis `./doc/generated/` sollte die Dokumentation der Implementierung mit Hilfe des Dokumentationswerkzeuges erzeugt werden (siehe Abschnitt 3.4). Um die verschiedenen Ausgabeformate zu unterscheiden sollen Verzeichnisse `./doc/generated/html/`, `./doc/generated/latex/` usw. erstellt werden.

Unterverzeichnis `./src/` : Hier werden alle Quelldateien abgelegt. Bei komplexeren Programmen macht es Sinn, noch weitere Unterverzeichnisse in `./src/` zu erstellen.

Unterverzeichnis `./test/` : Hier werden alle Testinstanzen gespeichert.

Unterverzeichnis `./lib/` : Es macht eventuell Sinn den Code als Bibliothek (ohne `main()`) zur Verfügung zu stellen.

5.3 Abschlusspräsentation

Die Abschlusspräsentation dient dazu, dass Sie ihr Programmpaket erklären und vor der Gruppe aller Praktikanten vorführen. Jeweils zu Beginn und Ende eines Semesters gibt es einen Präsentationstermin[6].

Sie sollten in etwa eine halbe bis dreiviertel Stunde dafür einplanen (bei Gruppen entsprechend mehr). Im wesentlichen sollte die Präsentation einen **theoretischen** und einen **praktischen** Teil haben.

Im theoretischen Teil erläutern Sie zunächst die Aufgabe. Dann schildern Sie, wie Sie das Problem modelliert haben, d.h. welche Datenstrukturen und welche Algorithmen Sie verwendet haben und wie diese zusammenspielen. Kurz gesprochen, sie stellen den **Entwurf** ihres Programmpaketes vor.

Im praktischen Teil überzeugen Sie uns davon, dass ihr Programm all das leistet, was es soll. Dazu gehört zunächst mal, dass es sich übersetzen läßt. Dann demonstrieren Sie die Funktionsfähigkeit anhand ihrer Testinstanzen und erklären die Bedienung des Programms. Zuletzt zeigen Sie uns die Dokumentation des Programmpaketes und der Implementation.

Machen Sie **frühzeitig** eine Gliederung Ihres Vortrages und sprechen Sie diese vor der Ausarbeitung Ihres Vortrages mit uns ab.

[6] Fragen Sie uns einfach wann die nächste Abschlusspräsentation stattfindet

6 Beispieldateien

6.1 Exemplarische Make Dateien

Für den eigenen Gebrauch sollten die folgenden Make Dateien heruntergeladen und an die eigenen Bedürfnisse angepasst werden. Folgende Dinge sind hier zu beachten:

- Das Makro `SOURCES` soll die Liste der eigenen Quelldateien enthalten. Hat man mehr als eine Quelldatei, so sollten die einzelnen Dateien der Übersichtlichkeit halber untereinander aufgeführt werden. Die Zeilenumbrüche müssen dann mit einem Backslash \ verdeckt werden.
- Für das Makro `EXE_NAME` sollte der gewünschte Applikationsname gesetzt werden.
- Die Makros `INCLUDE_PATH`, `LIBRARY_PATH` und `LIBRARIES` muss man an sein System anpassen. Meistens kennt der Compiler jedoch seine Standard Pfade, so dass man diese Makros auch leer setzen kann, indem die rechte Seite des Gleichheitszeichens gelöscht wird.

6.1.1 C

Eine Make Datei `MakefileC` für C:

```
################################### make ######################################
#
#  Module           : MakefileC
#  Description      :
#  Author           :
#  Email            :
#  Copyright        :
#  Created on       : Thu Apr 19 17:01:27 2001
#  Last modified by : ahr
#  Last modified on : Thu Apr 19 17:33:01 2001
#  Update count     : 6
#
###############################################################################

#
# M A C R O S

# Source files
SOURCES = source1.c \
          source2.c \
          sourcen.c

# Name of the application
EXE_NAME = myApplication

# Object files
OBJECTS = ${SOURCES:.c=.o}

# C Compiler
CC             = gcc
# Compiler flags
CFLAGS         = -g
# Add include paths prefixed by -I and separated by blanks
INCLUDE_PATH   = -I/usr/include
# Add library paths separated by colons :
```

```
LIBRARY_PATH   = /usr/lib:/usr/local/lib
# Add library names prefixed by -l and separated by blanks
LIBRARIES      = -lc

#
# T A R G E T S

# Create objects and link them with libraries
${EXE_NAME}: ${OBJECTS}
        ${CC} -o ${EXE_NAME} ${CFLAGS} -L${LIBRARY_PATH} ${OBJECTS} ${LIBRARIES}

# Produce object file from source file
%.o: %.c
        $(CC) $(CFLAGS) ${INCLUDE_PATH} -c $<

# Remove executable, object files and emacs autosave files
clean:
        rm -f ${EXE_NAME} ${OBJECTS} *~

# Generate documentation with doxygen
doc:
        doxygen

# Remove generated documentation
docclean:
        rm -rf ./doc/generated/
```

6.1.2 C++

Eine Make Datei `MakefileCPP` für C++:

```
################################### make ######################################
#
#  Module           : MakefileCPP
#  Description      :
#  Author           :
#  Email            :
#  Copyright        :
#  Created on       :
#  Last modified by :
#  Last modified on :
#  Update count     :
#
###############################################################################

#
# M A C R O S

# Source files
SOURCES = source1.cpp \
          source2.cpp \
          sourcen.cpp

# Name of the application
EXE_NAME = myApplication

# Object files
OBJECTS = ${SOURCES:.cpp=.o}

# C++ Compiler
CXX            = g++
# Compiler flags
```

```
CXXFLAGS       = -g
# Add include paths prefixed by -I and separated by blanks
INCLUDE_PATH   = -I/usr/include
# Add library paths separated by colons :
LIBRARY_PATH   = /usr/lib:/usr/local/lib
# Add library names prefixed by -l and separated by blanks
LIBRARIES      = -lstdc++

#
# T A R G E T S

# Create objects and link them
${EXE_NAME}: ${OBJECTS}
        ${CXX} -o ${EXE_NAME} ${CXXFLAGS} -L${LIBRARY_PATH} ${OBJECTS} ${LIBRARIES}

# Produce object file from source file
%.o: %.c
        $(CC) $(CFLAGS) ${INCLUDE_PATH} -c $<

# Remove executable, object files and emacs autosave files
clean:
        rm -f ${EXE_NAME} ${OBJECTS} *~

# Generate documentation with doxygen
doc:
        doxygen

# Remove generated documentation
docclean:
        rm -rf ./doc/generated/
```

6.2 Exemplarische C Dateien

Eine Definitionsdatei `example.h`:

```
/* ********************************** c ***************************************
 *
 *  Module           : example.h
 *  Description      :
 *  Author           :
 *  Email            :
 *  Copyright        :
 *  Created on       : Wed Aug 30 14:01:39 2000
 *  Last modified by : ahr
 *  Last modified on : Mon Apr 22 16:51:19 2002
 *  Update count     : 23
 *
 ********************************************************************************
 *
 *  Date        Name            Changes/Extensions
 *  ----        ----            ------------------
 *
 ******************************************************************************** */


#ifndef EXAMPLE_H
#define EXAMPLE_H

#include <includeFile.h>


/**
```

```
 * This function performs ... .
 * \param i an integer.
 * \param c an char.
 *
 * \return 0, if success.
 */
int exampleFunction(int i, char j);

#endif
```

Eine Quelldatei `example.c`:

```
/* ********************************** c ***************************************
 *
 *  Module           : example.c
 *  Description      :
 *  Author           :
 *  Email            :
 *  Copyright        :
 *  Created on       :
 *  Last modified by :
 *  Last modified on :
 *  Update count     :
 *
 ********************************************************************************
 *
 *  Date        Name            Changes/Extensions
 *  ----        ----            ------------------
 *
 ******************************************************************************** */

#include "example.h"

/*
 * e x a m p l e F u n c t i o n
 */
int exampleFunction(int i, char j){
    ...
}
```

6.3 Exemplarische C++ Dateien

Eine Definitionsdatei `example.hh`:

```
////////////////////////////////////// c++ ///////////////////////////////////
//
//  Module           : example.hh
//  Description      :
//  Author           :
//  Email            :
//  Copyright        : University of Heidelberg
//  Created on       : Wed Apr 18 16:09:30 2001
//  Last modified by : ahr
//  Last modified on : Mon Apr 22 16:53:14 2002
//  Update count     : 6
//
//////////////////////////////////////////////////////////////////////////////
//
//  Date        Name            Changes/Extensions
//  ----        ----            ------------------
```

```
//
////////////////////////////////////////////////////////////////////////////


#ifndef EXAMPLE_HH
#define EXAMPLE_HH


#include <includeFile.hh>


/**
 * This class ...
 */
class ExampleClass {


private:

    // --------------------------------------------------
    // -------- V a r i a b l e s -----------------------
    // --------------------------------------------------

    /**
     *
     */
    int m_exampleVariable;


public:

    // --------------------------------------------------
    // ------- M e t h o d s   ( p u b l i c ) ----------
    // --------------------------------------------------

    /**
     * Constructor.
     */
    ExampleClass();

    /**
     * This function performs ... .
     * \param i Meaning of this parameter.
     * \param c Meaning of this parameter
     *
     * \return 0, if it worked
     */
    int exampleFunction(int i, char j);


    // ------- from class YYY ------------------------------


protected:

    // --------------------------------------------------
    // ------- M e t h o d s ( p r o t e c t e d ) ------
    // --------------------------------------------------


private:

    // --------------------------------------------------
```

```
    // ------- M e t h o d s ( p r i v a t e ) -------------
    // -----------------------------------------------


}; // class ExampleClass


#endif
```

Eine Quelldatei `example.cpp`:

```
/////////////////////////////////////// c++ ///////////////////////////////////////
//
//  Module           : example.cpp
//  Description      :
//  Author           :
//  Email            :
//  Copyright        : University of Heidelberg
//  Created on       :
//  Last modified by :
//  Last modified on :
//  Update count     :
//
////////////////////////////////////////////////////////////////////////////////
//
//  Date        Name            Changes/Extensions
//  ----        ----            ------------------
//
////////////////////////////////////////////////////////////////////////////////


#include "example.hh"


//
// C o n s t r u c t o r
//
ExampleClass::ExampleClass(){
    ...
}

//
// e x a m p l e F u n c t i o n
//
int ExampleClass::exampleFunction(int i, char j){
    ...
}
```

Literatur

[Bal98] H. Balzert. *Lehrbuch der Softwaretechnik, Band 1: Software-Entwicklung.* Lehrbücher der Informatik. 1998.

[CLR94] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional Computing Series. 1994.

[KR90] B. W. Kernighan and D. M. Ritchie. *Programmieren in C.* 2nd edition, 1990.

[Mey99a] S. Meyers. *Effektiv C++ programmieren.* Professionelle Programmierung. 3rd edition, 1999.

[Mey99b] S. Meyers. *Mehr Effektiv C++ programmieren.* Professionelle Programmierung. 1999.

[OW96] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* 3rd edition, 1996.

[Sed95] R. Sedgewick. *Algorithmen.* 1995.

[Str98] B. Stroustrup. *Die C++ Programmiersprache.* 3rd edition, 1998.