TU Clausthal Institut für Mathematik

Programme in Studien- und Diplomarbeiten

Prof. Dr. M. Kolonko
Dipl.-Math. D. Wäsch
und
Dipl.-Math. B. Görder

Stand: 15. November 2007

1 Programmierrichtlinien

Im folgenden sind einige Regeln aufgestellt, die bei der Erstellung von Programmen eingehalten werden sollten.

1.1 Kommentare im Quellcode

Alle Klassen, Funktionen und Methoden sollten mit einem einheitlichen Kopf versehen sein, der durch automatische Dokumentationstools (wie Javadoc oder Doxygen) verstanden wird. Javadoc ist speziell für Java-Programme und produziert eine HTML-Dokumentation, Doxygen kann für Java, C, C++ u. a. Programmiersprachen benutzt werden und kann neben HTML und anderen auch LaTeX-Dokumentationen erzeugen.

Der Kopf einer Datei sollte folgende Gestalt haben (Bsp. für C++, in anderen Sprachen entsprechend, Metakommentare in Klammern

```
// ********************************************************
                          (55 mal ´*´)
// Dateiname: ...
//
// (Hier eine kurze Beschreibung des Inhalts der Datei
    z.B. Auflistung aller hier eingeführten Klassen)
//  ...
// --------------------------------------------------------
                          (55 mal ´-´)

// AUTOR/IN:  (mit Datum)
// (Änderungsinformation)
// ********************************************************
                          (55 mal ´*´)
```

Die Festlegung erscheint vielleicht etwas kleinlich, insbesondere die Vorgabe der Anzahlen von ´*´. Dies dient aber dem einheitlichen Erscheinungsbild auf verschiedenen Medien (z.B. Editoren und Ausdruck), ohne ungewollten Zeilenumbruch. Die Zeilenlänge sollte *maximal 80 Zeichen* betragen.

Jede Funktion/Methode muss einen Kopf der folgenden Gestalt tragen:

Doxygen

```
// ======================================================
                    (2 mal '/' und 55 mal '=')
/// (drei(!) '/' pro Zeile, gefolgt von der Beschreibung)
// ======================================================
                    (2 mal '/' und 55 mal '=')
```

Javadoc

```
/*********************************************************
                           (57 mal '*')
 * (Beschreibung)
 *********************************************************/
                           (57 mal '*')
```

Die Beschreibung sollte die Funktionsweise incl. der Nebeneffekte und Besonderheiten beinhalten und möglichst knapp gefasst sein. Außerdem können in der Beschreibung noch folgende Elemente zur automatischen Dokumentationserstellung enthalten sein:

`@brief <kurzbeschreibung>`: Beschreibung in einer Zeile

`@param <parametername> <beschreibung>`: zur Erstellung einer Paramenterbeschreibung

`@return <beschreibung>`: Erklärung für den Rückgabewert

`@see <methodenname>`: erzeugt einen Eintrag „siehe auch“

Kurze Kommentare (z. B. solche zu Variablen) müssen in der Zeile direkt davor stehen und mit einem `///` (Doxygen) bzw. `/** ...*/` (Javadoc) gekennzeichnet werden. Mit Doxygen kann der Kommentar auch hinter dem Bezeichner stehen. Dann muss der Kommentar aber mit `///<` eingeleitet werden, damit klar ist, dass sich der Text auf das vorangegangene bezieht.

Beispiel (Javadoc):

```
/** ein Integer */
int a;
/** ein double */
double b;
/** mein Objekt */
MeinObjekt c = new MeinObjekt();
```

Beispiel (Doxygen):

```
int a;      ///< ein Integer
int *b;     ///< ein Pointer auf ein Integer
int &c=a;   ///< eine Referenz auf a
```

1.2 Bezeichner

Eine besondere Bedeutung für die Verständlichkeit des Programmtextes kommt der geschickten Wahl der Namen (Bezeichner) zu. Grundsätzlich sollten als Bezeichner *deutsche Worte* oder Abkürzungen benutzt werden. Dies ermöglicht auch bei intensiver Benutzung externer Bibliotheken eine leichte Unterscheidung der selbstdefinierten Größen von solchen des Systems. Dazu genügt es u. U., dass die Bezeichner deutsche Bestandteile haben, z.B. `HauptPanel`.

Bei der Wahl von Namen für wichtige, häufig auftretende Größen und Funktionen sollte man versuchen, die Bedeutung möglichst präzise in dem Namen wiederzugeben ("selbsterklärend"). Z. B. sollte eine Variable, die die Länge eines Feldes angibt `Laenge` heißen. Im Zweifelsfall sollten lange Worte oder Zusammensetzungen benutzt werden, die insbesondere bei ihrem Aufruf ihre Wirkung erklären.

Auch lokale Hilfsvariablen sollten intelligent benannt werden. Wird z. B. ein Feld durchlaufen, so sollte die Laufvariable nicht `i` sondern (z. B.) `pos` oder `idx` (für Position oder Index) heißen. Wird das Feld an verschiedenen Stellen des Programms immer wieder durchlaufen, so sollten immer wieder dieselben Bezeichner benutzt werden (Kopieren von (fehlerfreien) Programmsegmenten).

Grundsätzlich sollte ein Bezeichner die Kategorie des Bezeichneten erkennen lassen (soweit dies möglich ist). Z. B. könnte ein Pointer auf eine Optimierungsprozedur OptimierungsProcPtr heißen, wird zusätzlich ein Typ für diese Pointer deklariert, so sollte er OptimierungsProcPtrTyp heißen.

1.3 Klassen

Bei der Deklaration von Klassen sollten die members in folgender Reihenfolge eingeführt werden:

- zunächst die einfachen member-Variablen, sortiert nach private, protected, public.

- dann die Konstruktoren, Destruktor, eventl. Operatoren sowie weitere Methoden, wieder sortiert nach Sichtbarkeit/Zugriffsrechten.

Klassen sollten so weit wie möglich gekapselt werden, d.h. auf die Deklaration von public member-Variablen sollte möglichst vollständig verzichtet werden. Der Zugriff auf member-Variablen sollte indirekt über so genannte Getter- und Setter-Methoden erfolgen.

1.3.1 Klassen in C++

Bei der Implementierung in C++ sollte die Aufteilung in separat zu compilierende Files (*.h und *.C , bitte nicht *.cc!) mit aussagekräftigen Namen erfolgen, die Rückschlüsse auf die enthaltenen Klassen zulassen. Bei der Implementierung der Klassen (in den .C-Dateien) sollte die Reihenfolge der Deklaration möglichst beibehalten werden.

Die im folgenden angegebenen Konstruktoren etc. können als Vorlage angesehen werden, die in den Kopf einer Klassendeklaration zu kopieren ist. "`XXXTyp`" muss dabei durch den Namen der Klasse ersetzt werden.

```
// ******************************************************
// Dateiname: klasseXXX.h
//
// ------------------------------------------------------
// AUTOR/IN:  4.1.2005, Dominic Wäsch
//
// ******************************************************

#ifndef _KLASSEXXX_H_
#define _KLASSEXXX_H_

// hier kommen nur die includes hin, die auch für dieses
// header-file wichtig sind! Die anderen includes gehören
// in die .C-Datei. Dies erhöht die Compile-Geschwindigkeit.

// ======================================================
/// @brief Eine Beispielklasse.
/// Sie enthält nur die nötigsten Methoden, die jede etwas
/// kompliziertere Klasse haben sollte.
// ======================================================
class XXXTyp
```

```
{
  // hier stehen die Member-Variablen, sortiert nach
  // public, protected und private

  public:

// ########################################################
// Konstruktoren, Deskruktor und Zuweisung
// ########################################################

// ======================================================
/// Standard-Konstruktor
// ======================================================
XXXTyp();

// ======================================================
/// Copy-Konstruktor, benutzt copy()
/// @param rSeite was kopiert werden soll
/// @see copy(const XXXTyp& rSeite)
// ======================================================
XXXTyp(const XXXTyp& rSeite)
  {copy(rSeite);};

// ======================================================
/// Destruktor, ruft dest() auf
/// @see dest()
// ======================================================
~XXXTyp()
  {dest();};

// ======================================================
/// Zuweisung, benutzt im wesentlichen copy()
/// @param rSeite was kopiert werden soll
/// @see copy(const XXXTyp& rSeite)
// ======================================================
XXXTyp& operator=(const XXXTyp& rSeite);

// ########################################################
// öffentliche Methoden
// ########################################################

// hier kommen die öffentlichen Methoden hin
```

```
private:

// ########################################################
// Hilfsmethoden
// ########################################################

// ======================================================
/// legt eine Kopie aller interner Strukturen an
/// wird im Copy-Konstruktor und im Zuweisungsoperator
/// verwendet
/// @param rSeite was kopiert werden soll
/// @see XXXTyp(const XXXTyp& rSeite) und
///      operator=(const XXXTyp& rSeite)
// ======================================================
void copy(const XXXTyp& rSeite);

// ======================================================
/// zerstoert die Skruktur; Methode ist abgesichert gegen den Fall,
/// dass noch keine Strukturen angelegt sind
// ======================================================
void dest();
};

#endif
```

1.4 Programmierstil

Während die Festlegung für das äußere Erscheinungsbild eines Programmtextes relativ einfach ist, ist die Vorschrift inhaltlicher Strukturen nur sehr allgemein möglich.

Grundsätzlich ist in allen Sprachen ein objektorientierter Stil mit möglichst stark voneinander abgekapselten logischen und programmtechnischen Einheiten zu benutzen. Die Schnittstellen zwischen den Einheiten sind so eng wie möglich zu wählen (z.B. möglichst wenig „public“ in Klassenheadern). Globale Variablen sollten vermieden werden, wo sie unumgänglich sind, ist dies zu begründen.

Besonders in C/C++ ist es möglich und üblich, wesentliche Schritte als Nebeneffekt von (u.U. unwesentlichen) Schritten zu programmieren. Z.B. können Wertzuweisungen in Abfragen untergebracht (d.h. versteckt) werden:

```
if( a=b<c ) {...}.
```

Dies ist in jedem Fall zu vermeiden! Übersichtlicher ist es hier, eine Zeile mehr zu verwenden:

```
a = b;
if( b<c ) {...}.
```

Gerade noch hinzunehmen sind Standardkonstruktionen wie:

```
while( datei.get( Zeichen ) )
{...}
```

In Zweifelsfällen geht Klarheit vor Effizienz. Nur in besonders laufzeit-kritischen Teilen darf auf maschinennahe Effekte zurückgegriffen werden, welche gut zu kommentieren sind. Fallunterscheidungen sind möglichst durch vollständige `if...else`- oder `switch`-Konstrukte zu beschreiben, z. B. nicht:

```
Annahme = false;
if( BedingungErfüllt )
    Annahme = true;
return Annahme;
```

sondern:

```
if( BedingungErfüllt )
  return true;
else     //  hier ist Bedingung nicht erfüllt
  return false;
```

1.5 Benutzung von Doxygen

Doxygen ist auf den Pool-Rechnern im Institut für Mathematik installiert. Die Benutzung erfolgt im Wesentlichen über die Konsole. Zunächst muss für das Projekt ein Konfigurationsfile erstellt werden, in der dann alle nötigen Einstellungen eingetragen werden. Dies geschieht automatisch mit

```
doxygen -g
```

Die erzeugte Datei heißt `Doxyfile` und muss an das Projekt angepasst werden. Dazu wird sie mit einem Texteditor geöffnet. Mindestens geändert werden sollten

```
PROJECT_NAME     = <projektname>
OUTPUT_LANGUAGE = German
```

Für ein reines Java-Projekt sollte außerdem der Schalter `OPTIMIZE_OUTPUT_JAVA` auf `YES` gesetzt werden. Statt die `FILE_PATTERNS` wie angegeben zu setzen kann auch die Option `INPUT` angepasst werden.

Sollen auch private und rein statische Methoden in der Dokumentation aufgelistet werden, müssen noch

```
EXTRACT_PRIVATE      = YES
EXTRACT_STATIC       = YES
```

gesetzt werden.

Um nun die Dokumentation zu erstellen, muss einfach nur

```
doxygen
```

eingetippt werden. Die Dokumentationen werden dann in Unterverzeichnissen `html`, `latex`, ... erstellt.

Eine genauere Beschreibung der Optionen und Möglichkeiten ist im Doxygen-Manual http://www.stack.nl/~dimitri/doxygen/manual.html zu finden.

2 Anfertigung der Programmbeschreibung

2.1 Einführung

Wird im Rahmen einer Studien- und Diplomarbeit auch ein Programm geschrieben, so gehört neben einer Beschreibung der Theorie im Normalfall auch eine Beschreibung des Programmes dazu. Dies hat unter anderen folgende Gründe:

1. Die Programmierung wird als Teil der Arbeit dokumentiert.
2. Der Programmcode wird durch eine sprachliche Umschreibung besser verständlich.
3. Die Fortentwicklung des Programms wird erleichtert, indem bestimmte Tricks, Kniffe und Sackgassen bei der Implementierung erläutert werden.
4. Die Arbeit des Studierenden im Zusammenhang mit der Programmierung kann besser beurteilt werden.

Dieses Dokument soll eine kurze Anleitung zu Struktur und Inhalt von solchen Programmbeschreibungen dienen.

Absichtlich nicht erwähnt wird hier die Benutzung des Programms im Rahmen der Diplom- oder Studienarbeit. Wurde das Programm geschrieben, um bestimmte Ergebnisse der Arbeit zu berechnen, so sind diese natürlich nicht innerhalb der Programmbeschreibung, sondern in anderen Teilen der Arbeit zu finden. In solchen Arbeiten tritt die Programmbeschreibung vielleicht nur in einem Anhang auf.

2.2 Aufbau

Eine Programmbeschreibung sollte sich grob in vier Teile gliedern:

1. Überblick und Aufbau,
2. Beschreibung der Klassen und Methoden,
3. Bedienungsanleitung und

4. Programmverhalten und Verifikation.

Im Folgenden wird genauer auf diese Punkte eingegangen.

2.3 Teil 1: Überblick und Aufbau

Zunächst soll in einem Überblick der grobe Aufbau beschrieben werden. Was ist der Zweck des Programms? Was ist die Funktionalität (d. h. wie erreicht es den Zweck)? Welche Programmiersprache wurde verwendet? Gibt es Programme oder Bibliotheken die benutzt werden? Wie ist der grundsätzliche Programmablauf? Welche Klassen gibt es? Welche zentralen Objekte werden wann instanziiert? Welche wichtigen Methodenaufrufe werden getätigt?

Dies dient vor allem der Orientierung des Lesers und dazu, nicht das Ziel aus den Augen zu verlieren. Dabei helfen auch vor allem grafische Überblicke wie z. B. UML-Diagramme, Ablaufschemata und Grafiken, die verdeutlichen, wann welche Methode aufgerufen wird. Aus der Zielstellung heraus ist klar, dass dieser Teil nicht besonders lang und ausführlich, sondern kurz und prägnant sein sollte.

2.4 Teil 2: Beschreibung der Klassen und Methoden

Als nächstes folgen dann Beschreibungen der Klassen und Methoden. In diesem größten und wichtigsten Teil sollte die Reihenfolge der Beschreibung möglichst einer inneren Logik folgen. Es sollte (was nicht immer möglich ist) in einer Beschreibung keine Klasse bzw. Methode erwähnt werden, die nicht zuvor beschrieben wurde.

Jede einzelne Klassenbeschreibung sollte ähnlich wie die gesamte Programmbeschreibung aufgebaut sein. Zunächst soll ein Zweck der Klasse angegeben werden: Wozu dient die Klasse? Was passiert? Welches sind die wichtigen Member-Variablen?

Zu jeder Klasse wird auf die nicht selbsterklärenden Methoden eingegangen. Dazu zählen nicht `get`- oder `set`-Methoden, sondern jede Methode, bei der nicht direkt aus dem Namen ersichtlich ist, was passiert. (Dabei sollte auch noch einmal überprüft werden, ob die Namen der Objekte, Methoden und Klassen gut gewählt sind. Wird der Inhalt durch den Namen charakterisiert? Haben alle Namen einen deutschen Anteil?) Dabei lohnt es sich, nicht nur den Methodennamen, sondern auch die Parameter anzugeben. In der Beschreibung kann so auf diese Namen einfach Bezug genommen werden. Komplizierte Algorithmen sollten zusätzlich durch einen Pseudocode und/oder durch Grafiken veranschaulicht werden.

Um Programmcode-Teile, wie z. B. Variablennamen kenntlich zu machen, sollten sie durch eine andere Schriftart (beispielsweise `Maschinenschrift`) kenntlich gemacht werden.

Zuletzt soll ggf. noch auf die Benutzung der Klassen eingegangen werden, d. h. welche Methoden müssen in welcher Reihenfolge aufgerufen werden? Dies ist insbesondere nötig, wenn nachfolgende Programmierer diese Programmteile weiter benutzen sollen.

In den einzelnen Beschreibungen sollte auch immer durch Verweise ein Bezug zur Theorie hergestellt werden. Wenn verschiedene Programmansätze unternommen wurden, muss beschrieben werden, welcher von diesen aus welchen Gründen umgesetzt wurde. Warum sind die anderen Algorithmen/Datenstrukturen verworfen worden? Wo liegen die Vor- und Nachteile?

Auch besondere Kniffe oder ungewöhnliche Programmiertechniken sollen hier erläutert werden, insbesondere, warum nicht der übliche Weg gewählt wurde. Dies dient vor allem dazu, die Fortentwicklung des Programmes zu vereinfachen und spätere Programmierer nicht in dieselben Fallen tappen zu lassen.

2.5 Teil 3: Bedienungsanleitung

Der dritte Punkt ist die Beschreibung der Bedienung. Hierbei sollte erklärt werden, wie das Programm benutzt wird, d. h. welche Parameter wo eingestellt werden können und wie der Aufruf erfolgt. Wie ist die Syntax der Parameterdatei? Wo kann man das Programmergebnis ablesen?

Ist das Programm eine Bibliothek, so muss hier auf die zentralen Klassen und Methoden eingegangen werden: Wie kann welche Aufgabe mit der Bibliothek erledigt werden? Was ist bei der Programmierung zu beachten? Von welchen Klassen muss abgeleitet werden?

2.6 Teil 4: Programmverhalten und Verifikation

Als letztes gehört auch noch eine Beschreibung des Programmverhaltens und eine Verifikation dazu. Es sollten Laufzeiten bzw. Ergebnisse von Testläufen angegeben werden. Anhand dieser Ergebnisse sollten algorithmisch kritische Stellen verifiziert werden können – das Programm soll also anhand von “schwierigen” (aber übersichtlichen) Eingabedaten die (verifizierbar) richtigen Ergebnisse liefern. Die Testdaten sollten dabei so gewählt werden, dass eventuelle algorithmische Besonderheiten und Fallen überprüft werden können.

Werden Laufzeiten angegeben, ist die Angabe der verwendeten Plattform und anderer Rahmenbedingungen wichtig: Welche Compiler-Version unter welchem Betriebssystem auf welcher Hardware wurde verwendet? Wie viele Versuche wurden gemacht? Wie hoch war die Varianz der Laufzeiten?

2.7 Fazit und Ausblick

Grundsätzlich lässt sich sagen, dass obige Anleitung zur Programmbeschreibung natürlich nur ein Vorschlag ist. Allerdings ist es ein Vorschlag, an dem sich jeder andere Aufbau messen lassen muss. Bei vielen Programmierprojekten wird die beschriebene Struktur nicht ganz passend sein, die Teile werden vielleich fließend ineinander übergehen oder von der Reihenfolge muss aus bestimmten Gründen abgewichen werden.

Rückmeldungen und weitere Literatur

Sollten sich grobe Fehler in dieses Dokument eingeschlichen haben oder sollten wir wichtige Punkte vergessen haben oder besser machen können, so bitten wir um Rückmeldung per E-Mail an mailto:{kolonko,goerder}@math.tu-clausthal.de. Dieses Dokument soll ja als Hilfe für zukünftige Arbeiten dienen und nicht zu Schwierigkeiten führen.

Lesenswertes zur Programmierung im Internet:

C++ Richtlinien:

http://www.possibility.com/Cpp/CppCodingStandard.html

extreme programming:

http://www.frankwestphal.de/ExtremeProgramming.html

Doxygen manual:

http://www.stack.nl/~dimitri/doxygen/manual.html

Doc Comments for Javadoc:

http://java.sun.com/j2se/javadoc/writingdoccomments/