8 Programmierkonventionen für FREAK

8.1 Ziele

- **Lesbarkeit** und „Schreibbarkeit“ des Codes für Projektteilnehmer erhöhen
  - ≠ Für den Autor
  - ≠ Für Projektneulinge
  - – Für Code-Reviews
  - – Für Wartung (mehrere Jahre)
  - – Lesegeschwindigkeit, Navigation, Verständlichkeit
- Annährung an selbstdokumentierenden Code (hoher Informationsgehalt)
- Fehlerquellen ausschalten
- Inkonsistenzen vermeiden
- Portabilitätsprobleme verhindern

8.2 Präambel

§ **1.** Diese Programmierkonventionen haben immer recht.

§ **2.** Sollten die Programmierkonventionen einmal nicht recht haben, so tritt automatisch Paragraph 1 außer Kraft. (Beispiel: wiederverwendeter Code.)

8.3 Grundlagen

8.3.1 Programmiersprache

ISO-C++.

C-Relikte vermeiden wie

- `printf`
- `struct`
- `malloc` und `free`
- Durch `inline`, `template` und `const` vermeidbare `#define`s

Folgende C++ Eigenschaften werden *nicht* empfohlen:

- Namespaces
- Ausnahmebehandlung
- Mehrfachvererbung
- „Advanced Features“ von `templates`

8.3.2 Sprache

Bezeichner englisch

Kommentare und Meldungen deutsch.

8.4 Bezeichner

Informationsgehalt, Merkbarkeit, Lesbarkeit

Daten: klein anfangen (Ausnahmen? math. Konventionen $G = (V, E)$, ...)

Typen: Groß anfangen

Konstanten: ALLES GROSS

8.4.1 Abkürzungen

- Sparsam Benutzen
- Gängige Abk. suchen
- Einheitlich. Beispiel (`Intf`)
- Ganze Silben
- Vokale weglassen aber Zungenbrecher vermeiden (Cnstrnt)

8.5 Zusammengesetzte Bezeichner

- Objektorientierung, Überladen hilft (z.B. printCircle, printTriangle, ...)
- Nicht übermäßig abkürzen
- Probleme mit Unterstrichen („_“):
  - Kostet Platz
  - Typografische Probleme (zu tief)
  - Bezeichner schwerer als Einheit wahrnehmbar
  - Weglassen einzelner _ erzeugt Chaos Beispiel: `PCL_L1_D_Cache_Miss`

⇒ Teilwort mit Grossbuchstaben beginnen lassen

- Verwandte Ding – ähnliche Bezeichner (Sender, AmateurSender)

8.6 Formatierung

8.6.1 White Space

- Keine `TAB`-Zeichen. Einrückungen bestehen aus Leerzeichen und die Standardeinrückung ist 2 Leerzeichen.
- Mindestens eine (besser zwei) Leerzeilen vor jeder Funktionsdefinition.
- Eine Leerzeile zum Trennen von logischen Codeblöcken innerhalb einer Funktion.

Operatoren

Leerzeichen um Binäroperatoren. (Ausnahme: „`.`“ und „`->`“, `x = 2*a + b;`

For-Schleifen

Zwei Leerzeichen nach dem Semikolon in `for`-Schleifen:

```
for (i = 0;  i < n;  i++) {...}
```

Kontrollstrukturen vs. Funktionsaufrufe

Ein Leerzeichen vor und nach einer Bedingung für die Kontrollstrukturen `if`, `for`, `while`. Aber *kein* Leerzeichen beim Prozeduraufruf `x = foo(2);`.

8.6.2 Klammerung

- Öffnende Klammern für Kontrollstrukturen auf der gleichen Zeile wie das Schlüsselwort. Bei Funktionen aber in Spalte 1 der folgenden Zeile.
- Alle Kontrollstrukturen sind zu klammern. Ausnahme: else-if Ketten.
- Lieber 10 Klammern zu viel, als eine zu wenig z.B. `2 + ((x & 3) << 4)`. Nicht jeder Programmierer beherrscht die 42 Klammerebenen von C im Schlaf.

8.7 Kommentare

Quelltext nicht paraphrasieren sondern ergänzen.

Der beste Kommentar ist einer, der gar nicht da ist, weil die Programmierung so kristallklar ist.

8.7.1 In Codezeilen

Ausrichten

8.7.2 Kommentarzeilen

Leerzeile vorher und nach kommentiertem Code.

8.7.3 Spezielle Kommentare

`//?` Noch zu ändern? Fehler? Unverständlich

`//>↑` Hack (gefährliche Kurve)

8.7.4 Vor Funktionsdeklaration

1. Informelle Beschreibung
2. Formale Vorbedingungen (`Pre:`)
3. Formale Nachbedingungen (`Post:`)

8.7.5 Vor Klassendefinition

1. Klartextbeschreibung. Kurz fassen! Übersicht, Verweise
2. Klasseninvarianten, für mehrere members (`//Inv:`)

```
//Inv: x*x + y*y <= 1
class PointOnUnitCircle
{ double x; double y;}
```

8.7.6 Invarianten von Members

`//Inv:`

Fehlender Kommentar bedeuted, daß alle Werte des angegebnene Typen erlaubt sind. Beispiel:
`double range; // Reichweite //Inv: >= 0`

8.7.7 Integrationsrelevante Klassen

LEDA's `lweb` Ansatz (Abschnitt 1.2. im LEDA-Handbuch.

8.7.8 Unterteilung von Dateien

Eine Datei wird durch Zeilen der Form

```
//============================ bla bla bla =====
//---------------------------- bla bla bla -----
```

In Abschnitte respektive Unterabschnitte unterteilt.

Integrationsgruppe: „Ausführungsbestimmungen“.
Beispiel: Abschnitt = Klasse

8.7.9 Beginn von Headerdateien

1. Ursprünglicher Autor (+ Teilprojekt)
2. Einigen Zeilen Funktionsbeschreibung
3. Jeweils eine Zeile für Änderungen.
   Mindestens jede global sichtbare Revision
   - Datum
   - Gegenstand der Änderung. Beispiel: „Doppelt querseitiges knodeln statt einfachem quibbeln“.
   - Wer hat die Änderung vorgenommen

Alternativ kann das obige Change-Log auch mit RCS geführt werden (versuchen Sie mal `man rcsintro`.

Codedateien

Redundante inkonsistente Kommentare können Verwirrung stiften

In Code-Dateien nichts wiederholen

Im Change-log nur Änderungen, die Header nicht betreffen.

Redundante Kommentare:

\+ Lesbarkeit

– Konsistenz schwer zu wahren

⇒ Chaos

8.8 Aufbau einer Klassendefinition

Reihenfolge der verschiedenen Teile einer Klasse (falls vorhanden):

1. `friend`s
2. Public Members
3. Public `static` Members
4. Public Konstruktoren
5. Public Destruktor
6. Public Zuweisungsoperatoren
7. Public Member-Funktionen
8. Private Members
9. Private `static` Members
10. Private Member-Funktionen

8.9 Dateien

8.9.1 Headers

Jedes Modul hat eine eigene Headerdatei (`.h`). Diese hat folgenden Aufbau:

```
#ifndef <MODULNAME>
#define <MODULNAME>
// Kommentare wie oben erklaert
...
#endif
```

Modul = Klasse oder mehreren nahe verwandte Klassen

Der Modulname ist oft der Name der wichtigsten Klasse im

Im Headerfile sind alle inline-Funktionen zu definieren.

8.9.2 Code-Dateien

Non-inline Funktionen in `.C` Datei (nicht `.cc`, `.cpp`).

Module ohne `.C` File OK

- Nicht zu viel Code
- Keine häufig verwendeten längeren Funktionen

8.10 Was (bisher) nicht geregelt ist

- Detailliertere Kommentar- und Dokumentationsregeln.
- Die Anfangsbuchstaben eines Variablenbezeichners codieren den Typ. (Bei objektorientierten Projekten klingt das schwierig.)
- Festlegung der Dateihierarchie für die Module
- Benutzungsregeln für Bibliotheken. Beispiel: Nach einem `new` immer ein Test ob Allozierung erfolgreich war.