Auswirkungen von Code Conventions auf Software Wartung und Evolution

Heinrich Moser (mail@heinzi.at)

08.06.2003

Diese Arbeit beschreibt die Charakteristika von Code Conventions, die generellen Vor- und Nachteile sowie die Vor- und Nachteile in Bezug auf die Wartung.

1 Einleitung

> First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.
>
> Linux kernel coding style [Tor]

Im Allgemeinen hat man bei einer Programmiersprache Freiheiten bezüglich der Form des Codes, der Benennung von Dateien, Variablen u.ä. und des Programmierstils (zu verwendende Patterns und Statements). *Code Conventions* (bzw. *Coding Guidelines*) bestimmen Normen, in denen diese Freiheiten eingeschränkt werden. Dies geschieht mit den Zielen, einerseits den Code besser lesbarer zu machen und andererseits eine einheitliche Richtlinie für Projekte, an denen mehrere Entwickler arbeiten, zur Verfügung zu stellen.

Es gibt keinen einheitlichen Standard, sodass Firmen und Entwicklungsgruppen ihre eigenen Codierungsnormen festlegen. Als Leitfaden wird dazu meist die Empfehlung des Herstellers oder Entwicklers der Programmiersprache genommen.

2 Übersicht

> Completed source code should reflect a harmonized style, as if a single developer wrote the code in one session.
>
> aus [Micf]

Code-Konventionen können in die folgenden Bereiche gegliedert werden:

- Dateinamen und Verzeichnisstrukturen
- Namenskonventionen
- Formatierung des Codes
- Kommentare
- Programmierpraktiken

3 Dateinamen und Verzeichnisstrukturen

Eine der ältesten Konventionen besteht darin, die Dateierweiterung entsprechend des Typs der Datei zu wählen, z.B. C-Headerfile (`.h`) oder Java-Code (`.java`). Spezielle Dateien haben auch traditionell vordefinierte Dateinamen, z.B. `Makefile`.

Weiters ist es bei objekt-orientierten Sprachen, die das Konzept von Packages oder Namespaces unterstützen, üblich, dass die Verzeichnis- und Dateinamen die Namenshierarchie widerspiegeln. Bei Java wird das beispielsweise für öffentliche Klassen bereits vom Compiler gefordert.

Der Wartungsnutzen dieser Maßnahme ist offensichtlich: Es ist nicht notwendig, den Inhalt aller Dateien zu durchsuchen, um den Source-Code einer gewünschten Klasse zu finden. Dies beschleunigt die Wartung und verringert die Wahrscheinlichkeit, dass Klassen „übersehen“ werden.

4 Namenskonventionen

Für Bezeichner oder *identifier* (z.B. Methodennamen, Funktionsnamen, Klassennamen, Variablennamen) können vom Programmierer Namen vergeben werden. Es gibt einen allgemeinen Konsens, dass „sprechende“ Namen vergeben werden sollten, die etwas über den Inhalt des benannten Konstrukts aussagen also z.B. `ProgressIndicator` statt `Class27` oder `isFinished` statt `flag3`. Weiters ist es auch anerkannte „best practice“, konsistent bei Namen und Abkürzungen zu sein. Gut gewählte Bezeichner können den Aufwand für spätere Änderungen drastisch senken, da die Semantik des Bezeichners bereits aus dem Namen herausgeht und nicht erst über reverse engineering ermittelt werden muss.

4.1 Groß-/Kleinschreibung

Üblich sind die vier Konventionen *Kleinschreibung* (`my_identifier`), *Großschreibung* (`MY_IDENTIFIER`), *Pascal casing* (`MyIdentifier`, auch „Infix Caps“ genannt) und *camel casing* (`myIdentifier`). Die Verbindung von Identifier-Typen mit jeweils einer dieser Schreibweisen (z.B. Java [Suna]: Großschreibung für Konstanten, Pascal casing für Klassennamen und camel casing für Variablen- und Methodennamen) wird gerne verwendet, um aus dem Textbild heraus den Typ eines Identifiers deutlich zu machen.

Beispiel. `ProgressIndicator.MAX` bezieht sich auf die Konstante `MAX` der Klasse `ProgressIndicator`, während bei `progressIndicator.step` die Methode `step` des von der Variable `progressIndicator` referenzierten Objekts gemeint ist.

4.2 Wortarten

Im Allgemeinen wird bei objekt-orientierten Programmiersprachen empfohlen, dass die Klassennamen Hauptwörter sind und die Methodennamen mit einem Verb beginnen (siehe z.B. [Suna], [Mice]). Damit wird sichergestellt, dass die objekt-orientierte Methapher beim Programmieren eingehalten wird.

4.3 Prä- und Suffixes

> As it turns out, the Hungarian naming convention is quite useful—it's one technique among many that helps programmers produce better code faster.
>
> Dr. GUI, Microsoft [Sim]

> Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged [...]. No wonder Microsoft makes buggy programs.
>
> Linus Torvalds [Tor]

Die Verwendung von vor- oder nachgestellten Buchstaben oder Buchstabenfolgen zur Angabe des Datentyps oder der Sichtbarkeit ist eine umstrittene Praktik. Vor allem von Microsoft wurde eine Variante dieses Programmierstils, die *hungarian notation* forciert (z.B. für C++ [Sim], für Visual FoxPro [Micb] und für Visual Basic 6.0 [Micd]), während andere Firmen wie z.B. Borland diesen Programmierstil ablehnen (für Delphi [Cal]). Seit Release des .NET-Frameworks rät jedoch auch Microsoft in [Micc] von dieser Technik ab:

> „Do not use Hungarian notation for field names. Good names describe semantics, not type."

Die Verwendung von Präfixes hat zweifellos Ihre Vorteile in Bezug auf Software Wartung: Das Lesen des Codes vereinfacht sich, da der Datentyp der Variable, den man normalerweise erst nachschlagen oder implizit aus dem Kontext erkennen müsste, bereits klar sichtbar ist. Dadurch wird bei überladenen Operatoren („/" sowohl für Ganzzahl- als auch für Fließkommadivision bzw. „+" sowohl für Stringverkettung als auch für Addition) klar, welche Operation gemeint ist. Weiters sind bei Sprachen, die (möglicherweise ungewünschte) implizite Konvertierungen vornehmen, diese deutlicher sichtbar.

Beispiel. Hungarian notation ist vor allem bei user interface-Designern beliebt: `txtDateiname` ist das Texteingabefeld für den Dateinamen; `lblDateiname` das danebenstehende

Bezeichnungsfeld „Dateiname:“; `cmdDateiname` ist die Schaltfläche, mit der man das „Datei öffnen“-Dialogfenster öffnet und `strDateiname` ist der tatsächliche String-Wert, der im Programm verwendet wird.

In der Praxis hat sich jedoch gezeigt, dass auch Nachteile mit diesem System verbunden sind, z.B. wird der Lesefluss gehemmt („madblXValues“ vs. „xValues“). Weiters ist es sehr aufwändig, bei der Verwendung dieses Systems konsistent zu bleiben, vor allem in objekt-orientierten Systemen, in denen laufend neue Klassen und damit neue Datentypen dazukommen. Eine Analyse der Vor- und Nachteile findet sich z.B. in [Lit]. Als Alternative wird z.B. von [Tor] empfohlen, die Funktionen so klein zu halten, dass die Anzahl der Variablen übersichtlich bleibt.

Vom Standpunkt der Software Evolution wäre noch anzumerken, dass Namenskonventionen, damit sie sinnvoll sind, über Jahre hinweg konstant bleiben müssen. Das ist ein nicht zu unterschätzendes Problem, da sich bald nach Einführung der neuen Präfix-Richtlinie der Wunsch einstellen wird, sie zu verbessern. Sobald diese jedoch geändert wird, wird alter Code möglicherweise schwerer lesbar als Code ohne Präfix-Richtlinien.

5 Formatierung des Codes

Programmiersprachen bieten unterschiedlich große Freiheiten bei der Verwendung von Whitespaces (Leerzeichen, Tabulatoren, Zeilenumbrüche, ...) im Code. Während z.B. bei C-basierten Sprachen (ANSI C, C++, Java, C#, ...) Whitespaces fast komplett ignoriert werden, verwenden andere Sprachen (z.B. Basic, Python, Haskell) Einrückungen und/oder Zeilenumbrüche als Syntaxelemente.

5.1 Einrückungen

Die gängigen Coding Guidelines sind sich darin einig, dass in Sprachen, die Blockstrukturen unterstützen, bei Verschachtelungen eingerückt werden sollte. Wie tief und mit welchen Whitespace-Zeichen (Leerzeichen oder Tabulator) einzurücken ist, ist umstritten. Da die Zeilenbreite aus Gründen der Lesbarkeit begrenzt sein sollte (sonst wird bei einer späteren Wartung möglicherweise Code übersehen, der sich außerhalb der Bildschirmbreite befindet), hat die Einrückungstiefe Einfluss auf die Verschachtelungstiefe. Während Code mit kurzen Einrückungen im Allgemeinen kompakter lesbar ist, hat das Verhindern von tiefen Verschachtelungen durch weite Einrückungen ([Tor] empfiehlt 8 Leerzeichen) den Vorteil, dass Programmierer dazu genötigt werden, komplizierte (und damit potentiell schlecht lesbare und fehleranfälle) Verschachtelungskonstrukte zu vereinfachen oder in mehrere Funktionen aufzuteilen. Allerdings besteht bei dieser Methode auch die Gefahr, dass der Programmierer zu allgemein als „harmful“ angesehenen Konstrukten wie z.B. `goto`-Sprüngen zurückgreift, um die fehlende Verschachtelungstiefe auszugleichen.

Die Verwendung von Tabulatoren hat den Vorteil, dass der Leser des Codes die Einrückungen ensprechend seiner persönlichen Tabulatorbreite angezeigt bekommt. Mögliche Nachteile wären die notwendige Konsistenz in der Verwendung, die mangelnde Flexibilität gegenüber Leerzeichen und die vom Betrachter abhängige maximale Zeilenbreite.

5.2 Klammerungen

```
//AVOID!
//THIS OMITS THE BRACES {}!
if (condition)
    statement;
```

[Suna] über Klammerung

Beispiel. Folgende C/Java-Codeteile sind äquivalent:

```
1: if (a == b) foo();

2: if (a == b)
     foo();

3: if (a == b) {
     foo();
   }

4: if (a == b)
     {
       foo(i);
     }
```

Falls nun, im Zuge einer Wartung des Codes, nach `foo()` auch noch `bar()` aufgerufen werden soll, kann Variante zwei zu einem Problem führen:

```
if (a == b)
  foo();
  bar();
```

macht nämlich nicht das, was man aufgrund der Einrückung erwarten würde. In den verschiedenen Coding Conventions werden unter anderem folgende Varianten zur Lösung des Problems vorgestellt:

Andere Einrückungsweite bei Klammerung In [Fre] wird empfohlen, bei Klammerungen prinzipiell Variante vier zu verwenden. Damit haben `if`-Statements, die mehr als eine Aktion durchführen, eine andere Einrückungsweite als Variante zwei.

Zwingende Klammerung In [Suna] wird empfohlen, immer Variante drei zu verwenden.

Ein weiteres Problem, das durch die zwingende Klammerung, nicht jedoch durch die unterschiedliche Einrückungsweite gelöst wird, ist das Problem der *dangling else ambiguity*. Der Code

```
if (a)
  if (b)
    foo();
else
  bar();
```

macht auch nicht das, was man aufgrund der Einrückung erwarten würden. In Pascal- oder C-basierten Sprachen wird der `else`-Zweig immer dem nächsten `if` zugeordnet, und das ist in diesem Fall `if (b)`.

Zwei andere – von den Code Conventions unabhängige – Ansätze zur Lösung der beiden Probleme bestehen darin, von Seiten der Programmiersprache aus immer automatisch einen Block zu beginnen und einen explizites Block-Ende-Statement einzuführen (z.B. `If ... Else ... End If` in Basic oder `if ... else ... fi` in Algol68). Die andere Möglichkeit besteht darin, einen Editor zu verwenden, der die Einrückung automatisch selbst vornimmt. Dadurch werden solche Fehler offensichtlich.

6 Kommentare

> The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.
>
> [Suna] über Kommentare

In die Kategorie der Kommentare fallen zwei Probleme der Software Evolution: die kontinuierliche Lesbarkeit des Programmcodes, um spätere Änderungen zu ermöglichen, und die Synchronisation des Programmes mit der dazugehörigen Dokumentation, wenn Änderungen vorgenommen werden.

Inline-Kommentare (d.h. kurze, meist einzeilige Kommentare innerhalb einer Funktion oder Methode) erhöhen die Lesbarkeit des Codes, wenn sie Informationen hinzufügen, die für das Verständnis das Codes notwendig sind und die nicht klar aus dem Code allein hervorgehen. Prinzipiell sollte man prüfen, ob man die Information nicht auch durch andere Möglichkeiten deutlich machen kann (z.B. durch einen aussagekräftigeren Variablennamen oder durch das Ersetzen eines konstanten Werts durch eine Konstante mit einem aussagekräftigen Namen). Bei Änderungen darf nicht darauf vergessen werden, die Kommentare mitzuändern. Auch Kommentare zur Unterstützung des Wartungsprozesses sind möglich (z.B. „`/* folgende fünf Zeilen wurden von` <*Name*> `am` <*Datum*> `auskommentiert, da` <*Begründung*>`. */`“). Diese Methode kann bei kleinen Projekten als Ersatz für ein Versionsmanagementsystem verwendet werden.

Über dem Kopf einer Funktion, Methode oder Klasse sollte ein umfangreicherer Kommentar die Funktionalität auf einer höheren Abstraktionsstufe beschreiben. Während in [Fre] noch empfohlen wird, die Beschreibung von C-Funktionen in informellen Sätzen abzufassen, gibt es

inzwischen (vor allem in Verbindung mit moderneren Sprachen) auch Möglichkeiten, die Beschreibung mit Tags zu annotieren, die nachträglich zu einer Schnittstellendokumentation verarbeitet werden können. Beispiele hierfür wären *Javadoc* (siehe [Sunb]) oder die XML-basierten *C# Documentation Comments* (siehe [Mica].

Beispiel. Das ist ein Beispiel für C# Documentation Comments für die Methode `Show` einer `HelloWorld` Klasse:

```
/// <summary>
///     Displays a Hello World message.
/// </summary>
/// <param name="firstName">
///     The first name of the person to greet.
/// </param>
/// <remarks>
///     Uses the <see cref="fontSize"/> field and the
///     <see cref="Color"/> property to determine how
///     to display the text.
/// </remarks>
public void Show(string firstName)
{
    ... // do something
}
```

Wenn ein eigenes Dokument zur Dokumentation der Schnittstelle oder der Design-Entscheidungen verwendet wird, besteht die Gefahr, dass dieses Dokument und der tatsächliche Source-Code im Laufe der Zeit divergieren, da Änderungen am Code vorgenommen, nicht jedoch in der Dokumentation nachvollzogen werden. Javadoc und ähnliche Technologien versuchen, dieses Problem zu lösen, indem der Code und die Dokumentation nahe beieinander gehalten werden.

Ein weiterer Punkt, der in Bezug auf nachträgliche Änderungen des Programmes wichtig ist, besteht darin, implizite Annahmen, die man als Programmierer trifft (z.B. dass ein gewisser Eingabeparameter nicht `null` ist oder dass gewisse Instanzvariablen bereits gesetzt sind), schriftlich festzuhalten. Diese Vorbedingungen (oder *preconditions*) können, genauso wie Nachbedingungen (*postconditions*; Zusicherungen, auf die man sich nach Durchführung der Funktion verlassen kann), entweder informell im Kommentar festgehalten oder über spezielle Tags angegeben werden. Sprachen wie z.B. Eiffel haben bereits integrierte Unterstützung für diese als *design by contract* bekannte Technik. Über Zusatztools wie z.B. JTest (siehe [Par]) kann auch eine automatische Prüfung dieser Bedingungen über spezielle Javadoc-Tags durchgeführt werden.

Beispiel. So könnte *design by contract* über Javadoc implementiert werden:

```
/**
 * @pre s != null
```

```
 * @post $result >= 0
 */

public int countChars (String s) {
    ...
}
```

7 Programmierpraktiken

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of *go to* statements in the programs they produce.
>
> [Dij68]

Programmierpraktiken sind schwer zu verallgemeinern, da sie stark von der verwendeten Programmersprache bzw. dem Programmierparadigma abhängig sind. Eine Auflistung dessen, was als „saubere“ objekt-orientierte Programmierung (z.B. Vermeidung unnötiger public members) oder „saubere“ prozedurale Programmierung (z.B. Vermeidung von `goto`) verstanden wird, würde den Rahmen dieser Arbeit sprengen.

Im Allgmeinen reduziert es sich darauf, die Lesbarkeit und damit die Wartbarkeit des Codes zu erhöhen. Programmkonstrukte, die von weniger gut geschulten Programmierern leicht missverstanden werden können, sollten vermieden werden. [Suna] empfiehlt beispielsweise, fall-throughs in `switch`-Statements (= `case`-Blöck ohne `break`) extra hervorzuheben. (Andere Sprachen wie z.B. C# sind aufgrund der Fehleranfälligkeit dieser Praxis dazu übergegangen, fall-throughs in `switch`-Statements komplett zu verbieten.) Auch Ausdrücke, die tiefergehendes Wissen über die Operatorpräzedenzen erfordern (z.B. die C-Ausdrücke `i+++j` oder `*i++`) sollten vermieden und stattdessen durch Klammerungen verdeutlicht werden.

8 Automatisierung

Vor allem in Bezug auf die Formatierung des Codes kann ein gut konfigurierter Editor dem Programmierer einige Arbeit abnehmen (z.B. GNU Emacs, Microsoft Visual Studio, IBM Eclipse). Diese Editoren (bzw. Zusatztools wie z.B. `indent`) können auch bereits bestehenden Code umformatieren und damit lesbarer machen.

Weitere Aspekte der Code Conventions und potentielle Fehlerquellen können durch statische Analyse-Tools (z.B. JTest [Par]) überprüft werden.

Beispiel. Unlesbarer Code:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-
'-',i+++"hello,world!\\n",'/''/''/'));}read(j,i,
p){write(j/p+p,i---j,i/i);}
```

Etwas besser lesbarer Code nach Anwednung von `indent -l40` und syntax highlighting:

```
int i;
main ()
{
  for (; i["]<i;++i){--i;}"];
       read ('-' - '-',
             i++ + "hello,world!\\n",
             '/' / '/'));
}

read (j, i, p)
{
  write (j / p + p, i-- - j, i / i);
}
```

9 Fazit

Code Conventions haben zwei Auswirkungen:

1. weniger Fehler während der Codierung (sowohl während der initialen Codierung als auch während der Wartung) und
2. weniger Zeit, die zum Verstehen des Codes notwendig ist.

Da laut [ZSG79] zwei Drittel der Kosten eines Software Projekts in die Wartung fallen und die Hälfte der Wartungsaktivitäten mit dem Verstehen des Source Codes verbracht wird, helfen Code Conventions dabei, die Gesamtkosten des Softwarelebenszyklus zu senken. Punkt 1 reduziert die Menge an notwendiger korrektiver Wartung während Punkt 2 den für die Wartung notwendigen Aufwand reduziert.

Literatur

[Cal] Charles Calvert. Object pascal style guide. `http://community.borland.com/soapbox/techvoyage/article/1,1795,10280,00.html`.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. `http://www.acm.org/classics/oct95/`.

[Fre] Free Software Foundation, Inc. Gnu coding standards. `http://www.gnu.org/prep/standards_toc.html`.

[Lit] J. Ambrose Little. Hungarian in .net. `http://aspalliance.com/Ambrose/Articles/Hungarian.aspx`.

[Mica] Microsoft Corp. C# documentation comments. http://msdn.microsoft.com/library/en-us/csspec/html/vclrfCSharpSpec_B.asp.

[Micb] Microsoft Corp. Microsoft visual foxpro language reference – naming conventions. http://msdn.microsoft.com/library/en-us/fox7help/html/lnoriNamingConventions.asp.

[Micc] Microsoft Corp. .net framework general reference – design guidelines for class library developers. http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp.

[Micd] Microsoft Corp. Visual basic 6.0 coding conventions. http://msdn.microsoft.com/library/en-us/vbcon98/html/vbconcodingconventionsoverview.asp.

[Mice] Microsoft Corp. Visual basic 7.0 (.net) language concepts – program structure and code conventions. http://msdn.microsoft.com/library/en-us/vbcn7/html/vbconProgrammingGuidelinesOverview.asp.

[Micf] Microsoft Corp. Visual studio .net coding techniques and programming practices. http://msdn.microsoft.com/library/en-us/vsent7/html/vxconCodingTechniquesProgrammingPractices.asp.

[Par] Parasoft. Jtest. http://www.parasoft.com/jsp/products.jsp?/jtest/.

[Sim] Charles Simonyi. Hungarian notation. http://msdn.microsoft.com/library/en-us/dnvsgen/html/hunganotat.asp.

[Suna] Sun Microsystems, Inc. Code conventions for the java programming language. http://java.sun.com/docs/codeconv/.

[Sunb] Sun Microsystems, Inc. How to write doc comments for the javadoc tool. http://java.sun.com/j2se/javadoc/writingdoccomments/index.html.

[Tor] Linus Torvalds. Linux kernel coding style. http://www.purists.org/linux/.

[ZSG79] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design*. Prentice-Hall, 1979.