„Best Practices“ für professionelle SAS-Programmierer

Patrick René Warnat
HMS Analytical Software GmbH
Rohrbacher Str. 26
69115 Heidelberg
patrick.warnat@analytical-software.de

Andreas Menrath
HMS Analytical Software GmbH
Rohrbacher Str. 26
69115 Heidelberg
andreas.menrath@analytical-software.de

Zusammenfassung

In der professionellen Softwareentwicklung gibt es einige nützliche Prinzipien, die sich auch gut auf die SAS-Programmierung anwenden lassen, aber unter SAS-Programmierern noch nicht so weit verbreitet sind, wie es wünschenswert wäre. Dieser Beitrag gibt einen Überblick zu solchen „Best Practices“, die jedem SAS-Programmierer bei der täglichen Arbeit eine große Hilfe sein können und letztendlich auch zu einer besseren Qualität der erstellten Software beitragen. Konzepte wie Iteratives Entwickeln, Unit Testing, Design by Contract, Defensives Programmieren, Versionsverwaltung und weitere Themen werden erläutert und die Vorteile ihrer Anwendung präsentiert.

Schlüsselwörter: SAS, Iteratives Entwickeln, Unit Testing, Design by Contract, Defensives Programmieren, Versionsverwaltung, Build-Tools, Quelltextdokumentation, Refactoring

1 Einleitung

Was macht einen guten SAS-Programmierer aus?
In diesem Beitrag soll ein Überblick zu Prinzipien und Praktiken gegeben werden, die sich im Arbeitsalltag professioneller SAS-Programmierer als nützlich erwiesen haben.

Es ist selbstverständlich, dass ein professioneller SAS-Programmierer sehr gute technische Programmierfähigkeiten (Verständnis der Funktionsweise des SAS-Systems, Syntax der Sprache, usw.) benötigt. Dies lässt sich sehr gut mittels Büchern und Kursen lernen und wird in den SAS-Zertifizierungen zur Programmierung geprüft. Dies ist die fundamentale Wissensbasis für SAS Programmierer und soll nicht Gegenstand dieses Beitrags sein. Weitere, generelle Fähigkeiten, die in verschiedenen Berufsgruppen wichtig sind, wie z. B. Kommunikationsfertigkeiten oder Selbstorganisation sollen auch nicht Gegenstand dieses Beitrags sein. Stattdessen soll der Fokus auf Prinzipien und Praktiken liegen, die spezifisch die Arbeit eines Programmierers betreffen und die eine wichtige Ergänzung zu den reinen SAS-Programmierkenntnissen sind.

Die Diskussion solcher Praktiken ist insbesondere für SAS-Programmierer wichtig, weil diese oft Quereinsteiger sind, die keine formale Ausbildung in einer Informatik-Fachrichtung durchlaufen haben.

Die Liste der im Folgenden beschriebenen Punkte ist nicht vollständig, sondern nur eine Auswahl im Umfang einer für diesen Konferenzbeitrag handhabbaren Menge. Außerdem stellt sie eine persönliche Selektion der Autoren dar. Von den beschriebenen Themen wurde das Thema *Design by Contract* exemplarisch für eine ausführlichere Beschreibung gewählt. Jedes Thema für sich ist nicht neu und sind ausführlich in verschiedenen Programmier-Büchern beschrieben (z. B. [1, 2, 3]). Allerdings sind sie eben auch für SAS-Programmierer relevant, so dass eine Zusammenstellung in diesem Kontext lohnenswert ist.

2 „Best Practices“ - Ein Überblick unter besonderer Berücksichtigung von *Design by Contract*

2.1 Rund um den Software-Lebenszyklus

Die folgenden Praktiken können eine große Hilfe, während der täglichen Arbeit eines SAS-Programmierers, sein.

2.1.1 Versionskontrolle

Eine Versionskontrollsoftware sollte verwendet werden, um Änderungen am Quelltext leichter administrierbar und nachvollziehbar zu machen. Zusammengehörende Änderungen können über Dateigrenzen hinweg als eine Einheit (ein *Commit*) in einem Repository gespeichert und kommentiert werden. Insbesondere die Arbeit an Quelltexten im Team wird durch Versionskontrollsysteme erheblich reibungsloser, aber auch für einzelne Entwickler ist der Einsatz eines Versionskontrollsystems immer empfehlenswert. Außerdem wird durch Versionskontrollsysteme das Risiko minimiert durch Änderungen ein Programm unwiderruflich zu beschädigen, da man immer auf einfache Weise eine frühere Version wiederherstellen kann.
Ein weit verbreitetes und frei verfügbares Versionskontrollsystem ist die Software Subversion [4].

2.1.2 Build-Tools

Der Einsatz eines Build-Tools, wie Apache Ant [5] oder ähnliche Anwendungen, kann die Ausführung von Routine-Arbeitsschritten vereinfachen und der Fehlervermeidung dienen. Zum Beispiel wird das Kopieren, Umbenennen und anschließende Verpacken von Quelltexten in eine Archivdatei automatisiert, um auf einfache Weise Auslieferungspakete schnüren zu können. Oder es werden Quelltexte in eine andere Ausführungsumgebung kopiert und dort automatisierte Tests angestoßen. Einmal eingerichtet,

erspart man sich Zeit bei der Ausführung von Routineaufgaben, die gewonnene Zeit kann man für die eigentlichen Programmieraufgaben verwenden.

2.1.3 Unterscheidung von Ausführungs-Umgebungen

Umgebungen zum Entwickeln und zum Produktiveinsatz sollten voneinander getrennt sein, um den reibungslosen Produktionseinsatz von Software möglichst wenig zu gefährden. Änderungen sollten stets nur in der für die Entwicklung bestimmten Umgebung durchgeführt werden. Bevor geänderte Quelltexte in die Produktivumgebung gelangen, sollten die Änderungen (je nach Projekt verschieden aufwendige) Qualitätsprüfungen bestanden haben. Die Änderungen werden dann in einem Arbeitsgang auf die Produktivumgebung übertragen.

2.2 Unit-Testing

Je früher eine Software getestet wird, umso besser. Daher sind für einen Programmierer Unit-Tests ein sehr wichtiges Konzept. Dabei werden Tests als ausführbare Quelltexte programmiert. Dadurch kann ein Test sehr einfach durch erneute Programmcodeausführung wiederholt werden. Bei Softwareänderungen können negative Seiteneffekte sehr schnell durch wiederholte Ausführung vorhandener Unit-Tests erkannt werden. Daher sollte man schon bei der initialen Erstellung eines Programms gleich auch Unit-Test-Programme mit erstellen. Als Nebenprodukt kann dies schon bei der Entwicklung unterstützen, da man einen klar definierten Ausführungsrahmen hat. Zusätzlich erschafft man gleichzeitig eine Sammlung von Beispielaufrufen, die anderen Programmierern beim Verständnis der Verwendung eines Programms helfen können.
Unit-Testing für SAS kann mit dem Framework SASUnit [6] umgesetzt werden.
Selbstverständlich sind andere Arten von Tests auch wichtig (z. B. Integrationstests), allerdings sind Unit-Tests der wichtigste Testansatz in der täglichen Arbeit eines Programmierers.

2.3 Quelltextdokumentation

Software-Dokumentation ist wichtig für die Anwendung und Wartung von Software. Als professioneller Programmierer sollte man insbesondere der Quelltextdokumentation Aufmerksamkeit schenken. Dies umfasst Syntaxkonventionen, lesbare und sinnhafte Namen für Makro-Variablen und Kommentarköpfe in Quelltextdateien. Jeder, der mit unter diesen Aspekten gut dokumentierten Quelltexten arbeitet, wird davon profitieren, insbesondere weil Struktur und innere Logik der Quelltexte schneller erfasst und verstanden werden können. Zusätzlich sollte man Kommentarköpfe in einem maschinenlesbaren Format verfassen, damit daraus mit einer entsprechenden Dokumentationssoftware (wie etwa das Tool Doxygen [7]) externe Repräsentationen (z. B. in Form von HTML-Seiten) der Dokumentation erstellt werden können.

2.4 Professionalität bei der Arbeit mit Quelltexten

Als ein professioneller Programmierer ist es wichtig, die eigene Einstellung und Herangehensweise in Bezug auf Programmierarbeiten zu reflektieren. Natürlich sind dabei viele Aspekte relevant, einige sehr wichtige sollten als Basis dienen und werden hier kurz beschrieben.

2.4.1 Pfadfinder-Regel

Ein Ort sollte immer in einem besseren Zustand zurückgelassen werden als man ihn vorgefunden hat. Angewandt auf die SAS-Programmierung bedeutet dies: Wenn man ein bestehendes SAS-Programm ändert, sollte man nicht nur die absolut nötigen Änderungen machen, sondern, falls etwas Verbesserungswürdiges (z. B. in Bezug auf Namenskonventionen) auffällt, sollte man dies auch im bestehenden Code ändern, der für die Umsetzung der aktuellen Änderungsanforderung nicht zwingend geändert werden müsste. Der Aufbau und die Gestaltung der Quelltexte werden somit kontinuierlich verbessert.

2.4.2 Quelltext-Reviews

Man sollte offen sein gegenüber Quelltext-Reviews mit Kollegen. Dadurch kann man noch besser von Kollegen lernen oder man gibt anderen eine bessere Chance von sich zu lernen. Auch zu vermeiden ist, dass ein Quelltext zwingend nur von der Person geändert werden darf, die den Quelltext ursprünglich entwickelt hat. Dadurch wird man flexibler und erreicht im Idealfall, dass verständlicherer Code geschrieben wird, da niemand Quelltexte als den „eigenen“ Code ansieht, den nur er/sie verstehen muss.

2.4.3 Iterative Entwicklung

Man sollte nicht versuchen zu viel Schritte auf einmal zu machen und stattdessen in kleinen Schritten arbeiten. So kann man zur Entwicklung eines Makros zunächst nur ein SAS-Programm mit den nötigen SAS PROC- und DATA-Steps schreiben, dies auf Lauffähigkeit prüfen und erst danach Makrovariablen und Makro-Kontrollstrukturen einführen. Dieses Vorgehen macht es einfacher komplexe Abläufe zu programmieren und ermöglicht es Test-Läufe so früh wie möglich in der Entwicklung auszuführen.

2.4.4 Refactoring

Hinter dem Begriff *Refactoring* steht das Konzept, dass man die interne Struktur von Quelltexten modifiziert, ohne das resultierende Verhalten einer Software zu verändern. Wenn man einen Quelltext so weit bearbeitet hat, dass das erwünschte Laufzeitverhalten erreicht worden ist, sollte sich möglichst oft zusätzlich die Zeit nehmen um Quelltextstrukturen noch mal zu prüfen und gegebenenfalls zu verbessern, insbesondere in Bezug auf die Lesbarkeit und Wartbarkeit des Quelltextes.

2.5 Arbeit mit Anforderungen

Es ist sehr hilfreich Wissen über den Kontext und das Anwendungsgebiet einer Software zu haben, die man programmieren soll. Dadurch können oftmals bessere Lösungen geschaffen werden, weil man sich über die bestehenden Anforderungen hinaus in den Anwendungsbereich eindenken muss und somit bessere Implementierungslösungen und bessere Fehlervermeidung umsetzen kann. Dieses Prinzip ist insbesondere im Bereich der SAS Programmierung relevant, da SAS Programme oftmals in sehr spezialisierten Anwendungsbereichen eingesetzt werden (Pharmazeutische Industrie oder Firmen im Finanzbereich), für die es eigenes Expertenwissen und eigene Fachbegriffe gibt.

Wenn man als SAS-Programmierer Anforderungen zur Umsetzung vorgelegt bekommt, ist es wichtig diese immer kritisch auf Vollständigkeit und Konsistenz zu prüfen. Wenn man sich als Programmierer für solch eine Prüfung Zeit nimmt, bevor man beginnt zu implementieren, kann man in der Regel effizienter programmieren, da Probleme mit den Anforderungen früh entdeckt werden. Ein weiterer wichtiger Punkt der Anforderungsprüfung aus Sicht des Programmierers ist es, für jede Anforderung zu prüfen, was die Kriterien sind, um sagen zu können, dass die Anforderung von der Softwareimplementierung erfüllt sind. Dies führt automatisch zu fokussierter Programmierung und zu eine guten Basis für die Tests der Software.

2.6 Defensives Programmieren

Während des Programmierens sollte man stetig prüfen, was während der Programmausführung schief gehen oder generell an Ausnahmesituationen auftreten können und entsprechende Prüfungen und Fehlerbehandlungsroutinen implementieren.
Dieses Vorgehen, auch „defensives Programmieren“ genannt, sollte die Entdeckungswahrscheinlichkeit von möglichen Problemen erhöhen.
Ein Beispiel dafür kann sein, dass man die Möglichkeit berücksichtigt, dass in den Eingabedaten Missing Values vorhanden sind. Ein weiteres Beispiel ist, dass man berücksichtigt, dass in einer Spalte mit Gruppierungsbezeichnern ein nicht erwarteter Wert vorkommen kann (z. B. anstatt ‘Dezember‘ der Bezeichner ‘December‘).

Es ist daher sinnvoll von Zeit zu Zeit mehr wie ein Tester anstatt wie ein Programmierer zu denken, um „robustere“ Software zu erstellen: Ein Programmierer setzt den Fokus darauf, wie ein Programm gestaltet sein muss um die gegebenen Anforderungen zu erfüllen. Ein Tester legt dagegen den Schwerpunkt darauf was die Schwächen eines gegebenen Programms sind und welche Eingabewerte zu einem Problem führen könnten.

Noch konsequenter verfolgt das Konzept *Design by Contract* diesen Ansatz, was im folgenden Abschnitt detailliert betrachtet werden soll.

2.7 Design by Contract

Die Software-Entwurfsmethode *Design by Contract* (abgekürzt: DbC) wurde erstmals in der Programmiersprache Eiffel angewandt und findet seitdem auch in anderen Programmiersprachen und Frameworks zunehmend Anhänger.
Das Konzept von Design by Contract sieht vor, sämtliche Schnittstellen eines Softwaremoduls zu seiner Umwelt als formelle Verträge zu beschreiben und die Einhaltung der Verträge während der Laufzeit des Programms fortlaufend sicherzustellen.

2.7.1 Ansätze von DbC in SAS

In der SAS Welt erstellen wir typischerweise wiederverwendbare Module in Form von SAS Makros. Ein SAS Makro existiert jedoch selten ohne weitere Abhängigkeiten zu seiner Umwelt, z.B. benötigt das Makro zur Laufzeit eine SAS Tabelle oder ein bestimmtes Benutzerformat für den Lookup von bestimmten Werten oder setzt eine bestimmte Systemoption voraus (z.B. VALIDVARNAME).
Ohne diese Abhängigkeiten ist das Makro selbst nicht lauffähig oder schlimmer noch: Es läuft ohne offensichtlichen Fehler weiter, produziert jedoch an anderer Stelle Nonsens-Ausgaben. Dann fallen die Probleme erst im Bericht für die Endanwender auf und der verantwortliche Programmierer muss sich unter Zeitdruck durch möglicherweise tausende von Codezeilen und dutzende Zwischenergebnistabellen „durchkämpfen“, um dann festzustellen, dass sich eine der Datenquellen geändert hat.
Als professioneller SAS Entwickler, sind Sie bestimmt schon einmal auf solche Probleme gestoßen und haben seitdem in ihren Makros eine Validierung der Eingabeparameter eingebaut, z. B. indem Sie überprüfen, dass die erwartete Tabelle mit Eingabewerten existiert und auch Datensätze enthält.
DbC geht jedoch noch einen Schritt weiter und formalisiert sämtliche Erwartungen an die Umwelt in Form von maschinell überprüfbaren Annahmen (im Englischen: *Assertion*). Diese Annahmen werden nun während der Laufzeit als Vor- und Nachbedingungen innerhalb des Makros überprüft. Wird eine der Bedingungen verletzt, wirft das Programm einen Fehler und stoppt gegebenenfalls die Verarbeitung.

2.7.2 Beispiel

Am besten lassen sich die Vorzüge und die Methodik anhand eines Beispiels vermitteln: Ihr Auftraggeber möchte das Makro Businesslogik implementiert bekommen. Es soll sich innerhalb eines Datasteps, wie folgt, aufrufen lassen und soll den Wert der DataStep-Variablen X manipulieren:

```
data ergebnis;
  set daten;

  %BusinessLogik()
run;
```

Als Spezifikation für das Makro sind die folgenden Aufgaben zu erfüllen:

- Für Kundengruppe = ‘A‘ soll der Wert aus Spalte X um 2 erhöht werden
- Für Kundengruppe = ‘B‘ soll der Wert aus Spalte X um 3 erhöht werden

Die folgenden Testdaten:

	kundengruppe	x
1	A	3
2	B	2
3	A	5

sollen also die folgenden Referenzdaten erzeugen:

	kundengruppe	x
1	A	5
2	B	5
3	A	7

Wahrscheinlich haben Sie das Makro „%BusinessLogik“ in weniger als 5 Minuten herunterprogrammiert und freuen sich, dass Sie so schnell fertig geworden sind.
In den ungenauen Anforderungen in der Spezifikation lauern jedoch einige Probleme, die erst beim zweiten Blick sichtbar werden. So müsste man im Rahmen einer *Defensiven Programmierung* zumindest einige weiterführende Rückfragen an den Auftraggeber stellen. Hier eine Auswahl:

- Sind alle gültigen Kundengruppen stets als Großbuchstaben vorhanden, oder kann z.B. Kundengruppe ‘A‘ auch als ‘a‘ in den Daten vorkommen?
- Wie ist mit MISSING Werten in der Spalte X umzugehen?
- Existieren auch andere Kundengruppen? Wie sind diese zu behandeln?

Die letzte Frage mit den Ausprägungen der Kundengruppen soll hier exemplarisch, auch für die anderen Problemstellungen, tiefergehend analysiert werden. Ihr Auftraggeber versichert Ihnen natürlich, dass nur die beiden Kundengruppen A und B jeweils in Großbuchstaben aus den Quellsystemen geliefert werden. Zum Zeitpunkt der Implementierung ist diese Annahme demnach berücksichtigt worden.
Natürlich kommt es nun, wie es kommen muss: 6 Monate nach erfolgreichem Programmeinsatz ihres Moduls, wird festgestellt, dass in den Berichten die Zahlen für die Spalte X nicht korrekt sind. Nach stundenlangem Debuggen und Überprüfung der Zwischenergebnistabellen stellen Sie nun entsetzt fest, dass Ihr Modul für die falschen Zahlen verantwortlich ist, da in den Liefersystemen nun auch die Kundengruppe C eingeführt wurde.

Im Folgenden werden unterschiedliche Implementierungen vorgestellt und ihr unterschiedliches Verhalten auf die unbekannte Kundengruppe C analysiert:

Code 1:

```
if kundengruppe = "A" then x = x + 2;
if kundengruppe = "B" then x = x + 3;
```

Auswirkung: der Wert von Spalte X bleibt unverändert.

Code 2:

```
if kundengruppe = "A" then x = x + 2;
else x = x + 3;
```

Auswirkung: der Else-Zweig wird auch bei Kundengruppe C ausgeführt. Wert in Spalte X wird um 3 erhöht.

Code 3:

```
if kundengruppe = "B" then x = x + 3;
else x = x + 2;
```

Auswirkung: der Else-Zweig wird auch bei Kundengruppe C ausgeführt. Wert in Spalte X wird um 2 erhöht.

Code 4:

```
length summand 8;
if kundengruppe="A" then summand = 2;
if kundengruppe="B" then summand = 3;
x = x + summand;
drop summand;
```

Auswirkung: die Variable Summand wird nicht befüllt und bleibt Missing. Durch die Addition mit Missing wird der Wert von Spalte X nun auch Missing.

Neben diesen vier Implementierungsbeispielen sind auch eine Vielzahl von alternativen Implementierungen, z.B. über einen Format- oder Hashtable-Lookup, denkbar. Der Kernpunkt aller Implementierungen sollte jedoch deutlich geworden sein: Alle Implementierungen liefern für die Testdaten die korrekten Ergebnisse; jedoch bei unerwarteten Daten (in Form einer Kundengruppe C) werden vollkommen unterschiedliche, willkürliche Ergebnisse zurückgeliefert.
Wendet man das Best Practice *Unit-Testing* an, so werden für die Implementierung des Makros mehrere erfolgreiche Unit-Tests geschrieben und für die Testdaten werden für alle vier Implementierungen immer die korrekten Referenzdaten erzeugt. Schlimmer noch: die Unit-Tests für das Modul „%BusinessLogik“ können sogar eine Testabdeckung von 100% ausweisen und dem Entwickler damit eine falsche Sicherheit vorgaukeln! Hier zeigt sich, wie wichtig die Herangehensweise des *Defensiven Programmierens* ist: Ein tieferes Verständnis der Problemfälle und auch der Berücksichtigung aller Umstände, unter denen das Modul funktionieren soll, müssen bereits bei der Implementierung umfassend berücksichtigt werden.

2.7.3 Implementierung mit DbC

Dem zugrunde liegenden Problem (mit den unerwarteten Eingabedaten) kann nur dadurch entgegengewirkt werden, dass die Erwartungshaltung des Programmierers und der Spezifikation in SAS Code überführt wird und somit auch die Eingabedaten, auch während der Programmausführung, kontinuierlich überprüft werden.
In einigen Programmiersprachen, wie Java oder C#, werden *Assertions* (Annahmen) bereits direkt in der Sprache bzw. dem Framework unterstützt. In der SAS Welt stehen

Assertions jedoch leider nicht zur Verfügung. Doch seit SAS Version 9.3 kann man mit PROC FCMP glücklicherweise eigene SAS-Funktionen programmieren, ohne auf SAS/Toolkit und C-Programmierung zurückgreifen zu müssen. Eine SAS-Funktion, die die Überprüfung von Assertions (Annahmen) übernimmt, ist schnell definiert:

```
function assert(expr, description $);
  if (not expr) then do;
    /* TODO: Aktion bei Verletzung der Assertion definieren */
  end;
  return (expr);
endsub;
```

Die FCMP Funktion verfügt über zwei Parameter. Der erste Parameter ist ein beliebig komplexer Ausdruck, der von der Funktion überprüft werden soll. Der zweite Parameter ist ein Text, der die Annahme des Entwicklers als positiv formulierte Erwartungshaltung beschreibt. Für die Prüfung der Kundengruppe würde ein Aufruf der ASSERT-Funktion im Makro „%BusinessLogik“ daher, wie folgt, implementiert werden können:

```
rc = assert(kundengruppe IN('A', 'B')
            , CAT("Zeile ", _n_, "enthält gültige Kundengruppe")
            );
```

Sobald nun der Ausdruck „expr“ in der ASSERT-Funktion als unwahr ausgewertet wird, springt SAS innerhalb der FCMP Funktion in den IF-Block, der auf die Verletzung der Assertion (Annahme) reagieren soll. Hier sind viele unterschiedliche Aktionen denkbar, wie z.B. eine WARNING oder einen ERROR in das SAS Log schreiben, das Setzen einer Makrovariable, oder das Schreiben eines Fehlers in eine Protokoll-Tabelle. Hier bleibt es dem Entwickler überlassen, den Schweregrad einer Assertion-Verletzung zu definieren und zu entscheiden, wie der weitere SAS Programmablauf nach einer Assertion-Verletzung gestaltet werden soll.

Als Anregung für eine eigene Implementierung folgt nun ein vollständiges Beispiel, das die Kundengruppe und die Spalte X auf Missing Values prüft und im Fall, dass eine Assertion-Verletzung auftritt, einen Fehler in das Log schreibt und die automatische Makrovariable SYSCC auf den Fehlercode 8 setzt.
Um die Flexibilität den DbC Ansatzes zu demonstrieren, wird der Rückgabewert der Assert Funktion in der Datastep-Variable RC gespeichert. Wird eine Annahme verletzt, so wird der Datensatz im Programmbeispiel direkt gelöscht.

```
proc fcmp outlib=work.functions.test;
function assert(expr, description $);
  attrib errorline length=$100;

  if (not expr) then do;
    /* Fehler in Log schreiben */
    FILE LOG;
```

```
    errorline = CATS('ERROR: assertion "', description, '" is
invalid;');
    PUT errorline;

    /* Makrovariable für System Returncode setzen */
    CALL SYMPUT("SYSCC", "8");
  end;

  return (expr);
endsub;
run;
options cmplib=(work.functions);

%macro BusinessLogik();
  /* Annahmen prüfen */
  rc = assert(kundengruppe IN('A', 'B') and not missing(x)
           , CAT("Zeile ", _n_, ": Eingabedaten sind OK"));

  /* Daten bereinigen */
  if (not rc) then delete;

  /* Berechnung durchführen */
  if kundengruppe = "A" then x = x + 2;
  else if kundengruppe = "B" then x = x + 3;

  drop rc;
%mend BusinessLogik;

data daten;
  length kundengruppe $1 x 8;
  input kundengruppe $ x;

  datalines;
A 3
B 2
C 7
A .
a 3
;
run;

%let syscc = 0;
data ergebnis;
  set daten;
  %BusinessLogik()
run;

%put syscc=&syscc.;
proc print data=ergebnis;
run;
```

Im SAS Log erscheinen nun 3 Fehler und das Ausgabe Dataset wurde um die 3 ungültigen Datensätze bereinigt:

```
ERROR: assertion "Zeile 3: Eingabedaten sind OK" is invalid;
ERROR: assertion "Zeile 4: Eingabedaten sind OK" is invalid;
ERROR: assertion "Zeile 5: Eingabedaten sind OK" is invalid;
NOTE: There were 5 observations read from the data set WORK.DATEN.
NOTE: The data set WORK.ERGEBNIS has 2 observations and 2 variables.
```

2.7.4 Fazit zu DbC

Mit einer selbst gebauten FCMP Funktion, lassen sich einige Ansätze aus dem *Design by Contract*-Paradigma auch leicht in eigenen SAS Programme verwenden. Durch den Aufruf einer ASSERT-Funktion lassen sich Fehler frühzeitig erkennen und im weiteren Programmablauf lässt sich flexibel auf Fehlersituationen reagieren. Es empfiehlt sich daher, bereits bei der Entwicklung eines Moduls, die wichtigsten Annahmen über die Laufzeitbedingungen des Moduls in Form von automatisiert überprüfbaren Assertions direkt in den Programmcode einzubauen. Auch bestehende Programme können ohne großen Aufwand um Assertions erweitert werden.
Die ASSERT-Funktion ist äußerst flexibel und kann z.B. auch per „%SYSFUNC“ in Makroanweisungen aufgerufen werden oder auch für komplexe Prüfungen von Makrovariable, Systemoptionen, Dateien, u.V.m. verwendet werden.

3 Fazit

Die aufgeführten Prinzipien und Praktiken repräsentieren eine subjektive Auswahl ohne Anspruch auf Vollständigkeit. Sie sind als Inspiration zu verstehen, was neben der eigentlichen Programmierfertigkeit und neben Soft-Skills wichtige Fähigkeiten von professionellen Programmierern sind.

Wenn Sie nun mit neuen guten Vorsätzen an Ihren Arbeitsplatz zurückkehren, hier noch ein Tipp: Natürlich ist es nicht einfach alle empfehlenswerten Praktiken in der täglichen Arbeit anzuwenden. Man sollte immer versuchen sich nicht zu viel auf einmal vorzunehmen, sondern lieber versuchen konstant in kleinen Schritten voranzukommen. Manche der Praktiken sind sehr gut alleine umsetzbar, manche sind aber nur oder besser im Team umsetzbar, wenn möglich suchen Sie sich unter Ihren Kollegen Mitstreiter zur Umsetzung.

Wenn Sie ein erfahrener professioneller Programmierer sind und diese und andere Techniken schon beherrschen: Gratulation!

Literatur

[1] K. Henney (Hrsg.): 97 Things Every Programmer Should Know. O'Reilly, 2010.

[2] R. C. Martin: Clean Code. Prentice Hall International, 2008.

[3] K. Passig, J. Jander: Weniger schlecht programmieren. O'Reilly, 2013.

[4] *Subversion*-Webseite: http://subversion.tigris.org

[5] *Ant*-Webseite: http://ant.apache.org/

[6] *SASUnit*-Webseite: http://sourceforge.net/projects/sasunit/

[7] *Doxgen*-Webseite: http://www.stack.nl/~dimitri/doxygen/