Programmierrichtlinien

Stefan Beimel
Merz Pharmaceuticals GmbH
Eckenheimer Landstr. 100
60318 Frankfurt / Main
stefan.beimel@merz.de

Zusammenfassung

Dieser Beitrag möchte den Vortrag ‚Programmierrichtlinien – unterstützende Hilfe oder lästige Formalität?' vertiefen, der auf der 10. KSFE in Hamburg gehalten wurde.
Es hat sich gezeigt, dass ein großes Interesse an Programmierrichtlinien besteht und dass es die allgemeine Einsicht in deren Notwendigkeit gibt. Genauso groß ist aber auch das Diskussionspotential: Schon bei der Diskussion, wie eingerückt wird und wann eine neue Zeile beginnen soll, scheiden sich die Geister. Unter www.redscope.org kann man sich an einem Diskussionsforum beteiligen.
Hier können deshalb keine allgemein gültigen Regeln vorgegeben werden. Die hängen von vielen Faktoren ab. Es sollen aber Anregungen gegeben werden, welche Themengebiete eine Programmierrichtlinie abdecken sollte. Es werden viele Beispiele aus der täglichen Praxis gezeigt. Dieser Beitrag enthält alle Beispiele, die auf der 10. und der 11. KSFE präsentiert wurden.

Folgende Themen sollen genauer betrachtet werden:

- Dokumentation im Programmcode
- Fehlervermeidung
- Makro-Programmierung
- Performance
- Programmierumgebung

Neben dem Wissen, nach was man sich richten könnte, ist natürlich die tatsächliche Einführung der Regeln wichtig. Der Beitrag möchte Anregungen geben, wie diese Richtlinien zum Leben erweckt werden können. Dabei muss auch immer ein Kompromiss zwischen ‚reiner Lehre' und Praktikabilität gefunden werden.

Schlüsselwörter: Programmierrichtlinien, Dokumentation, Makro, Fehlervermeidung

1 Einführung von Programmierrichtlinien in die tägliche Praxis

1.1 Vorbereitung von Programmierrichtlinien

Der erste, vorentscheidende Schritt bei der Einführung von Programmierrichtlinien besteht darin, einen Verantwortlichen ernennen. Es ist sehr wichtig, dass dieser Kollege

von der Bedeutung von Programmierrichtlinien überzeugt ist und deren Entwicklung und Pflege soviel Aufmerksamkeit widmet, dass auch mal operative Arbeit liegen bleibt.

Die erste Aufgabe dieses Verantwortlichen ist es, die derzeitige Programmierkultur unter denjenigen Programmierern zu ermitteln, die später mit den Richtlinien arbeiten müssen. Eine Möglichkeit besteht darin, Programme der dann Betroffenen zu lesen. Er muss sehr aufmerksam spontane Bemerkungen registrieren, sowohl positive als auch negative. Viele Ideen entstehen auch beim einfachen Fragen nach Wünschen, Erfahrungen, Ideen und Problemen.

Dann müssen Vorschläge zu Programmierrichtlinien erarbeitet werden. Hierbei ist es wichtig, keine Vorschriften zu machen. Die endgültige Akzeptanz der Richtlinien hängt ganz stark davon ab, wie weit die Betroffenen zu Beteiligten gemacht wurden. Die Ideen zu diesen Vorschlägen können sich aus den derzeitigen Programmiergewohnheiten ableiten, aber auch Diskussionsforen [1], Vorträge [2] und Bücher [3] können Ideen liefern.

Bevor die Programmierrichtlinien verbindlich werden, sollten alle Programmierer in einer Schulung auf den neuesten Stand gebracht werden. Es ist sehr wahrscheinlich, dass nicht jeder jede Diskussion bis zum Ende verfolgt hat und über den letztendlichen Kompromiss im Bilde ist – und Diskussionen finden mit Sicherheit statt.

1.2 Pflege und Kontrolle

Sind die Regeln einmal verabschiedet, müssen sie auch gepflegt und regelmäßig kontrolliert werden. Eine Checkliste kann dabei gute Dienste leisten. Zum einen zur 'Freiwilligen Selbstkontrolle', aber auch zur Überprüfung während der Validierung eines Programms. Während dieser Checks werden sicher Abweichungen von den Programmierrichtlinien aufgedeckt. Das kann mehrere Ursachen haben. Einerseits ist der Programmierer vielleicht nicht über die Regeln im Bilde, andererseits sind die Regeln möglicherweise nicht praxistauglich. In beiden Fällen muss nachgebessert werden, entweder durch Nachschulung oder durch Überarbeitung der Regeln selbst.

1.3 Eigenschaften von Regeln

Damit Regeln praxistauglich sind, müssen sie gewisse Kriterien erfüllen.
Sie müssen **konkret** sein. Allgemeine Regeln wie 'defensiv programmieren' nützen im Alltag wenig und können auch nur sehr schwer kontrolliert werden.

Es sollten nur **Regeln zu tatsächlich vorkommende Problemen** aufgestellt werden. Wird z. B. im Arbeitsumfeld kein PROC IML benutzt, werden auch keine speziellen Regeln dafür benötigt.

Regeln müssen **effizient** sein, d. h. Aufwand und Nutzen müssen in einem vernünftigen Verhältnis zu einander stehen. Viele Überlegungen zur Performancesteigerung sind nicht effizient, wenn nur kleine Datensätze verarbeitet werden sollen. Die Überlegungen und deren Umsetzung dauern dann länger als die Zeit, die jemals damit gespart werden kann.

Regeln dürfen **nicht zu restriktiv** sein. Z. B. ist es wohl unbestritten, dass Einrückungen im Programmcode existieren sollten, um Zusammenhänge deutlich zu machen. Geschmackssache ist dagegen, wie tief diese Einrückung sein soll. Eine solche Verfeinerung würde nichts zur Verbesserung der eigentlichen Regel beitragen.

Regeln müssen **begründbar** und nachvollziehbar sein. Schulungsunterlagen sollten viele Beispiele enthalten.

2 Dokumentation im Programmcode

2.1 Kommentare

Eines der wichtigsten Elemente, Programmcode verständlicher zu machen, ist der Kommentar. Leider wird in der Programmierpraxis meist nicht ausreichend kommentiert, weil das Programm ja auch ohne Kommentare funktioniert. Kommentare sind so etwas wie Luxus, den sich nur der leistet, der zu viel Zeit hat.

Es ist sicher nicht nötig, jede Zeile zu kommentieren, aber insbesondere komplizierte, ungewöhnliche Abschnitte sollte man beschreiben - zeitnah, damit auch der Autor selbst noch versteht, was gemeint ist.

Am Anfang jedes Programms sollte ein (möglichst einheitlicher) Programmkopf stehen, der wenigstens zeigt, wer programmiert hat, wann das Programm geschrieben wurde und was es überhaupt macht.

Längere Programme können durch Zwischenüberschriften gegliedert werden.

Ein Programm ist kein Archiv! Während der Programmentwicklung kommt es oft vor, dass Abschnitte geschrieben und wieder verworfen werden. Im endgültigen Code sollten die verworfenen Zeilen auch tatsächlich gelöscht und nicht nur auskommentiert werden, weil auskommentierter Code üblicherweise zu großer Verwirrung führt.

SAS bietet mehrere Möglichkeiten, Kommentare zu setzen:

- Blockkommentare `/*  */`
- Zeilenkommentare* `;`
- Makrokommentare `%*  ;`

Blockkommentare sind bequem, um größere Abschnitte auszukommentieren. Leider können sie nicht geschachtelt werden - Code mit diesen Kommentaren kann also nicht ohne weiteres selbst auskommentiert werden. Deshalb sollte mit dieser Art der Kommentierung sehr sparsam umgegangen werden. Es gibt zwar die Möglichkeit, den Code zu einem Dummy-Makro zu machen und es dann nicht auszuführen - aber das ist sicher nicht jedem sofort präsent, besonders nicht Programmieranfängern.

Empfohlen	Nicht empfohlen

`data ...;`
`  set demo %* Demographie;`
`  ae  %* Adverse Events;`
	`data ...;`
	`  set demo /* Demographie */`
	`  ae  /* Adverse Events */`

2.2 Transparente Programmierung

Besser als jede Kommentierung ist es natürlich, den Code selbst sprechender zu machen:

- Sprechende Programm-, Datensatz-, Variablen- und Formatnamen verwenden
- Positiv denken

Empfohlen	Nicht empfohlen
`if age > 35 then ...;`	`if not (age <= 35) then ...;`

- Die Zahlen 0 und 1 wie Wahrheitswerte benutzen

Empfohlen	Nicht empfohlen
`if not first.patno then ...;`	`if first.patno=0 then ...;`

- IN-Operator statt einer Folge von ORs verwenden

Empfohlen	Nicht empfohlen

`if patno in(12, 14, 18)`
`  then ...;`
	`if patno=12 or patno=14 or`
	`  patno=18`
	`  then ...;`

2.3 Layout

Eine große Rolle bei der Verständlichkeit eines Programms spielt auch das Layout, d. h. die Strukturierung des Codes. Allerdings ist dieses Kapitel auch das streitbarste unter den Programmierrichtlinien. Es gibt hier kein richtig oder falsch.

- Einrücken und Ausrichten
  DO/SELECT sollten mit dem zugehörigen END in einer Spalte beginnen, THEN mit dem korrespondierenden ELSE:

```
data new;
   set old;
   do i=1 to 13;
      output;
   end;
run;
```

  Wie viele Leerzeichen eingerückt werden ist eine Frage des Geschmacks und des verfügbaren Platzes. Allerdings sollte nicht mit Tabulatoren eingerückt werden, weil Tabulatoren in jedem Editor anders interpretiert werden.

- ein Statement pro Zeile

- RUN und Leerzeile nach jedem DATA bzw. PROC step

- Kleinbuchstaben für reservierte Wörter
  - lassen sich besser lesen,
  - lassen sich einfacher programmieren (kein Betätigen der Shift-Taste; kein Nachdenken, was groß, was klein geschrieben werden muss).
  - Konsequentes Schreiben von Großbuchstaben ist schwer durchzuhalten, wodurch der Code beinahe zwangsläufig uneinheitlich wird.

3 Fehlervermeidung

3.1 ERRORs, WARNINGS und verdächtige NOTEs

ERRORs, WARNINGs und einige NOTEs weisen auf ein Fehlverhalten von Programmen hin. Diese Meldungen sollten im Log grundsätzlich vermieden werden.

NOTEs werden meist als harmlos betrachtet - dabei sollten manche NOTEs in den Status von ERRORs erhoben werden. Einige NOTEs können, müssen aber nicht auf Fehler hinweisen. Um diese Ungewissheit zu vermeiden, sollten sie ganz unterbunden werden - zumal sie meist mit sehr einfachen Techniken verhindert werden können:

```
NOTE: Variable ... is uninitialized
NOTE: MERGE statement has more than one dataset with repeats of ...
NOTE: At least one W.D ...
NOTE: Invalid argument ...
NOTE: Invalid data ...
NOTE: LOST CARD
```

```
NOTE: Numeric values have been converted ...
NOTE: Character values have been converted ...
NOTE: ... is already on the library
NOTE: Division by zero detected
NOTE: ... outside the axis range ...
NOTE: SAS went to a new line ...
NOTE: ... 0 variables ...
```

3.2 'Werterhaltendes' Runden von numerischen Variablen

Zahlen mit Nachkommastellen lassen sich i. A. nicht exakt im Binärcode darstellen. Das Problem wird erst dann offensichtlich, wenn die Zahl mit einem sehr langen Format dargestellt wird (z. B. 30.25).

Das Ergebnis der Aufgabe 4,6 + 0,1 + 0,1 ist in SAS nicht 4,8, sondern 4,79999999999999. Probleme entstehen, wenn dieses Ergebnis zum Vergleichen oder Sortieren benutzt wird, z. B. beim Vergleich von Laborwerten mit Normbereichen oder in PROC NPAR1WAY mit der Option WILCOXON.

Dieses Problem kann gelöst werden, wenn man berechnete, numerische Variablen vor dem Vergleichen oder Sortieren auf die Genauigkeit von 10^{-12} rundet:

```
round(num_var,10E-12)
```

10^{-12} ist ein Erfahrungswert, der sowohl den Wert der Variablen nicht verändert (es sei denn, die 12. Stelle hinter dem Komma enthält tatsächlich relevante Informationen) als auch die 4,7999... in eine 'echte' 4,8 verwandelt.

Beim Importieren aus externen Datenbanken (z. B. Oracle) sollten grundsätzlich alle Zahlen gerundet werden.

3.3 Das SELECT Statement

Angenommen, je nach Geschlecht eines Patienten soll ein Index vergeben werden: Frauen die 0,6, Männer erhalten etwas weniger, nämlich 0,5.

```
if sex='male' then index=0.5;
              else index=0.6;
```

Diese Lösung ist nahe liegend, birgt aber ein Risiko: Alle Werte außer 'male' erhalten den Wert 0,6, also auch 'Male' und ' '. Es sollte also jeder Wert explizit genannt und auch unerwartete Werte abgefangen werden. Das ist mit sequentiellen IF - THEN - ELSE erreichbar:

```
if sex='male'
```

```
   then index=0.5;
   else if sex='female'
           then index=0.6;
           else put 'WARNING: Unerwarteter Wert ' sex=;
```

Je nach Komplexität der Abfrage, der Anweisungsblocks und Strukturierung des Codes kann das Programm leicht unübersichtlich werden.

Eine andere, meist sehr elegante Möglichkeit, dieses Problem zu lösen, ist das SELECT Statement:

```
select(sex);
    when('male')     index = 0.5;
    when('female')   index = 0.6;
    otherwise put 'WARNING: Unerwarteter Wert ' sex=;
end;
```

3.4 LENGTH Statement für neue Textvariablen benutzen

Die Länge einer Textvariablen wird von SAS während der Kompilierungsphase in dem Moment festgelegt, in dem die Variable das erste Mal erwähnt bzw. gebraucht wird. Geschieht das unbedacht, 'rät' SAS manchmal eine Länge, die nicht ausreicht, um alle gewünschten Texte richtig darzustellen. Die Länge einer Textvariablen sollte deshalb vor ihrem ersten Gebrauch explizit mit dem LENGTH Statement festgelegt werden.

Empfohlen

```
data alter;
   length txt $5;
   alter = 20;
   if alter > 100
       then txt="alt";
       else txt="jung";
   put txt=;
run;

Log: txt=jung
```

Nicht empfohlen

```
data alter;

   alter = 20;
   if alter > 100
       then txt="alt";
       else txt="jung";
   put txt=;
run;

Log: txt=jun  <== NICHT ERWÜNSCHT
```

3.5 TRIM Funktion beim Verketten von Text benutzen

Beim Verketten von zwei Textvariablen wird die zweite Variable hinter das Ende der gesamten ersten Variablen angefügt. Dabei spielt es keine Rolle, wie lang der tatsächliche Text ist: Auch Leerzeichen zählen zur Gesamtlänge mit. Um also den Inhalt der zweiten Variablen direkt an das Ende des Textes der ersten Variablen anzufügen, müssen die Leerzeichen am Ende der ersten Variablen mit der Funktion TRIM( ) abgeschnitten werden.

Empfohlen	Nicht empfohlen

`data name;`
`  length name $20;`
`  name = "Anna";`
`  name = trim(name)||"-Maria";`
`  put name=;`
`run;`
 	
`Log: name=Anna-Maria`
	`data name;`
	`  length name $20;`
	`  name = "Anna";`
	`  name = name||"-Maria";`
	`  put name=;`
	`run;`
	
	`Log: name=Anna  <== NICHT ERWÜNSCHT`

4 Performance

Die Zeit, die ein Programmlauf benötigt, lässt sich grob in zwei Kategorien einteilen: Zum ersten in die Zeit, die für Lese- und Schreiboperationen auf einen Datenträger benötigt wird, zum anderen in die eigentliche Rechenzeit, die CPU-Zeit. Erfahrungen zeigen, dass das Lesen und Schreiben üblicherweise deutlich länger dauert als Rechenzeit benötigt wird. Soll also die Verarbeitungsgeschwindigkeit erhöht werden, müssen vor allem überflüssige Lese- und Schreiboperationen vermieden werden.

4.1 Permanente Datensätze sortiert ablegen

Sortieren ist eine der langwierigsten Operationen. Programmierregeln sollten also sicherstellen, dass möglichst wenig sortiert wird. Eine Möglichkeit ist, permanent abgelegte Datensätze nach einem logisch sinnvollen, eindeutigen Schlüssel sortiert abzuspeichern. Das spart viele Sortiervorgänge während der Programmentwicklung von Programmen, die auf diese Datensätze lesend zugreifen. Außerdem lassen sich sortierte Datensätze bequemer betrachten, z. B. im SASViewer oder mit ViewTable.

4.2 Sortieren vermeiden durch page-Dimension

Es ist unter Umständen möglich, das Sortieren ganz zu vermeiden, wenn man geschickte Statements der Prozeduren verwendet.

Eine 'dritte' Dimension (neben Spalte und Zeile) lässt sich in PROC TABULATE auf verschiedene Weisen erzeugen. Zum einen ist es möglich, ein BY Statement zu benutzen. Zum anderen gibt es die so genannte page-Dimension im TABLES Statement. Das Ergebnis ist bis auf kosmetische Feinheiten das gleiche. Der entscheidende Vorteil der page-Dimension ist, dass der Datensatz nicht sortiert vorliegen muss.

Um den Laufzeitunterschied einschätzen zu können, wurde ein Datensatz mit 10 Millionen zufälligen, unsortierten Beobachtungen erzeugt. Erstaunlicherweise gab es bei den Laufzeiten zwischen sortierter BY-Variable und unsortierter page-Variable kaum Unterschiede. PROC TABULATE benötigte jeweils ca. 6 Sekunden. Der ent-

scheidende Laufzeitunterschied entstand durch die Sortierung des Datensatzes mit PROC SORT, wenn das BY Statement benutzt wird. Das Sortieren dauerte ca. 70 Sekunden.

Empfohlen	Nicht empfohlen

`proc tabulate data=sashelp.prdsale;`
`  class region prodtype country;`
`  table **country**, region, prodtype;`
`run;`
	`proc sort data=sashelp.prdsale`
	`  out=prdsale;`
	`  by country;`
	`run;`
	
	`proc tabulate data=prdsale;`
	`  **by country;**`
	`  class region prodtype;`
	`  table region, prodtype;`
	`run;`

4.3 Keine Datasteps, wenn nur Metadaten geändert werden sollen

Metadaten sind Daten, die die Datenstruktur beschreiben, also Label, Formate u. ä. Um diese Daten zu ändern, müssen nicht die Beobachtungen selbst angefasst werden, sondern nur der so genannte Datensatzheader. Sollen also ausschließlich Metadaten geändert werden, sollte auch nur dieser Header bearbeitet werden. Da ein Datastep immer eine Kopie eines Datensatzes erzeugt, ist er für diese Aufgabe ungeeignet. Alternativen sind PROC DATASETS bzw. das Zuweisen von Labels und Formaten in den Ausgabeprozeduren, in denen sie benötigt werden.

4.4 PROC APPEND benutzen beim Verketten von strukturgleichen Datensätzen

Oft sollen zwei strukturgleiche Datensätze in dem Sinne verknüpft werden, dass die Beobachtungen des zweiten Datensatzes als zusätzliche Zeilen zu den Beobachtungen des ersten Datensatzes hinzugefügt werden. Hier liefert das SET Statement des Datastep das gewünschte Ergebnis. Aber auch hier gilt wieder: Der Datastep liest und schreibt sämtliche Datensätze. Werden keine weiteren Funktionen des Datastep benötigt, ist PROC APPEND (oder PROC DATASETS mit APPEND Statement) für diese Aufgabe viel effizienter, da es nur die Beobachtungen des zweiten Datensatzes liest und den ersten unberührt lässt.

Empfohlen	Nicht empfohlen

`proc datasets;`
`  append base=grosser_datensatz`
`  new =kleiner_datensatz;`
`quit;`
	`data grosser_datensatz;`
	`  set grosser_datensatz`
	`  kleiner_datensatz;`
	`run;`

4.5 Variablen mit KEEP/ DROP entfernen

Unnötiges Lesen und Schreiben kann man auch vermeiden, indem nur die tatsächlich benötigen Variablen verarbeitet werden. Die Selektion kann über KEEP bzw. DROP erfolgen. KEEP/ DROP kann als Datensatz-Option, aber auch als Statement im Datastep verwendet werden. Überdies dient das 'positive denkende' KEEP auch noch einer besseren Dokumentation.

4.6 Observations mit WHERE/ subsetting IF entfernen

Genau wie Variablen sollten auch Beobachtungen frühzeitig auf die wirklich notwendigen beschränkt werden. Das kann mit Hilfe von WHERE (als Statement oder Datensatz-Option) oder dem so genannten subsetting IF im Datastep geschehen.

4.7 Kombination von Datasteps

Häufig kann man beobachten, dass in einem Datastep Datensätze erzeugt werden, die ausschließlich als Eingabe für einen nächsten Datastep dienen. Das mag historisch gewachsen sein oder zur Vereinfachung des Debuggens dienen – effizient ist es nicht, da diese Datensätze mindestens einmal zuviel geschrieben und gelesen werden. Im Allgemeinen lassen sich Datasteps wie im Beispiel unten zusammenfassen. Schwierigkeiten mit der Zusammenfassung kann es allerdings geben, wenn im ersten Datastep mehr als ein OUTPUT Statement enthalten ist.

Empfohlen	Nicht empfohlen

`data nachdem2ten;`
`  set wasauchimmer;`
`  SAS statements;`
`  weitere SAS statements;`
`run;`
	`data nachdem1sten;`
	`  set wasauchimmer;`
	`  SAS statements;`
	`run;`
	
	`data nachdem2ten;`
	`  set nachdem1sten;`
	`  weitere SAS statements;`
	`run;`

5 Makro-Programmierung

Alles bisher Geschriebene gilt ganz besonders auch für die Makroprogrammierung. Insbesondere Makros, die nicht nur für eine bestimmte Studie bzw. ein bestimmtes Programm geschrieben wurden, müssen natürlich besonders robust (im Sinne der Fehlervermeidung) und effizient programmiert sein. Außerdem ist es sehr wahrscheinlich, dass auch andere Programmierer als der Autor dieses Makro pflegen werden, was besonders hohe Ansprüche an die Dokumentation stellt. Zusätzlich zu den oben genannten gibt es noch spezielle Regeln für Makros.

5.1 Alle Makrovariablen als %LOCAL deklarieren

In jeder Programmiersprache haben Variablen bestimmte Gültigkeitsbereiche. Das gilt auch für Makrovariablen in SAS. Im Allgemeinen werden Makrovariablen, die innerhalb eines Makros zugewiesen werden, auch nur von diesem selbst gebraucht. Diese Variablen sollten als lokale Variablen mit %LOCAL gekennzeichnet werden. Geschieht das nicht, können sich die Gültigkeitsbereiche überlappen und zwei Variablen mit dem gleichen Namen sind plötzlich tatsächlich identisch mit allen unerwarteten und unerwünschten Beeinflussungen.

Im unten gegebenen Beispiel benutzen zwei Makros die gleiche Zählvariable I. Durch die Deklaration mit %LOCAL existieren die Variablen nur innerhalb des entsprechenden Makros und die Aufgabe wird wie erwartet gelöst. Ohne %LOCAL wären die beiden I's ein und dieselbe Variable mit entsprechenden Folgen.

Aufgabe

```
Outer loop: 1
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4

Outer loop: 2
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4
```

Lösung

```
%macro inner;
   %local i;
   %do i= 1 %to 4;
      %put %str(  ) Inner loop: &i;
   %end;
%mend;

%macro outer;
   %local i;
   %do i= 1 %to 2;
      %put Outer loop: &i;
      %inner;
   %end;
%mend;
```

5.2 Datensatz- und Formatnamen

Anders als bei Makrovariablen gibt es für Datensätze und Formate keine entsprechende Deklaration als lokal. Eine Möglichkeit, dies trotzdem zu simulieren und damit ein ver-

sehentliches Überschreiben von Datensätzen zu vermeiden ist die Verwendung spezieller Namenskonventionen. Bewährt hat sich folgende Nomenklatur:
Datensatzname innerhalb eines Makros =
Unterstrich + Makroname + (laufende Nummer oder sprechende Endung)
Daraus folgt zwingend, dass Datensätze außerhalb von Makros keine führenden Unterstriche haben dürfen.

5.3 Klammern bei 'Funktionen'

Makrotext, der wie eine Funktion verwendet werden soll, muss in Klammern eingeschlossen werden. Das stellt sicher, dass dieser Term auch als Einheit gesehen wird und nicht durch andere Regeln (z. B. Punkt- vor Strichrechnung) zerrissen wird.

Empfohlen

```
%macro plus_eins(var);
   (&var + 1)
%mend;

data p1;
   a = 5;
   b = 2 * %plus_eins(a);
   put b =;
run;

Log: b=12
```

Nicht empfohlen

```
%macro plus_eins(var);
   &var + 1
%mend;

data p1;
   a = 5;
   b = 2 * %plus_eins(a);
   put b =;
run;

Log: b=11        <== FALSCH
```

6 Programmierumgebung

Alles in den Kapiteln zur Dokumentation, Fehlervermeidung, Performance und Makroprogrammierung lässt sich innerhalb eines einzelnen Programms umsetzen. Die Regeln in diesem Abschnitt gehen darüber hinaus. Um sie einführen zu können, ist viel Konzeptarbeit und die Einrichtung einer Programmierumgebung notwendig. Die Umsetzung dieser Regeln hängt sehr stark von der vorhandenen EDV-Technik ab (Hardware, weitere Software, Betriebssystem, ...).

6.1 Batchmodus

Im Laufe einer interaktiven SAS-Session sammeln sich eine Menge Objekte an, die mit dem eigentlichen Programm nichts mehr zu tun haben, es aber trotzdem beeinflussen können: Optionen, Datensätze, Libnames, Makrovariablen, ...

Um den Einfluss solcher 'historischen Informationen' auf das Programm auszuschließen, sollte der produktive Programmlauf im Batchmodus erfolgen (unter Windows mit der rechten Maustaste im Explorer zu starten).

Zwei Klassiker von Programmen, die nie im Batchmodus gelaufen sein können:

- unvollständige Programme
  Hier wird der Datensatz A benutzt, ohne dass er vorher erzeugt wurde. Solche Programme sind im Allgemeinen unbrauchbar:

```
data b;
    set a;  *** ?????? *;
run;
```

- Programme mit falscher Chronologie
  Hier wird der Libname L benutzt, bevor er zugewiesen wurde. Durch Umsortieren kann man solche Programme oft retten.

```
data b;
    set l.a;   *** l ist noch nicht zugewiesen **;
run;

libname l "C:\daten";
```

Ein weiterer Vorteil des Batchmodus ist, dass automatisch eine vollständige, in sich geschlossene Log-Datei angelegt wird. Ohne dieses Log gibt es keinen Beweis, dass ein Programm jemals gelaufen ist und fehlerfrei funktioniert hat.

6.2 Scannen des Logs

Im Abschnitt 'ERRORs, WARNINGs und verdächtige NOTEs' wurde beschrieben, welche Zeilen im Log eines Programms vermieden werden sollten. Die erste Herausforderung ist, diese Zeilen zu finden. Ein unbefriedigender Ansatz ist es, jedes mal das Log manuell zu durchsuchen. Deshalb sollte es einen Prozess geben, der automatisch, d. h. ohne aktives Eingreifen des Programmierers, das Log scanned. Das ist zumindest beim Einrichten eines Batchmodus und beim Lauf eines Programms über ein 'Metaprogramm' (s. u.) möglich.

6.3 AutoExec.sas / SAS.cfg

In diesen beiden Dateien können global gültige Einstellungen für die SAS Umgebung gespeichert werden, z. B. Optionen und spezielle Verzeichnisse. Sie sollten physikalisch nur einmal vorhanden sein, wobei alle SAS Programmierer auf diese eine Datei zugreifen. Das hat den Vorteil, dass diese Einstellungen unabhängig von einer lokalen Installation sind und auch nicht in jedem SAS Programm neu gesetzt werden müssen.

6.4 Metaprogramme

Die Analyse einer klinischen Prüfung besteht aus einer Reihe von Programmen zum Erzeugen von Formatkatalogen, permanenten Datensätzen, Rohdatenlisten, Auswertungstabellen und Statistiken. Die Anzahl dieser Programme kann je nach Umfang der Studie leicht 30 und mehr betragen. Die Anzahl der Ausgabedateien, die die Listen, Tabellen und Statistiken enthalten, ist in der Regel noch höher, da jedes Programm mindestens eine Ausgabedatei erzeugt. Daher ist es erwünscht, eine zusammenfassende Auswertung auf 'Knopfdruck' zu ermöglichen.

Zwei Metaprogramme bzw. Makros, die bereits auf früheren KSFEs vorgestellt wurden, können dabei Hilfe leisten. Ein Programm führt die Programme aus, scanned das Log und sucht nach dem Output der SAS Programme [4]. Ein zweites verarbeitet den Output (Listen, Tabellen und Grafiken) und fasst ihn in einer PDF-Datei zusammen [5].

Literatur

[1] www.redscope.org/coding

[2] www.redscope.org/coding/links

[3] Harald Kellerwessel: Programmierrichtlinien in der Praxis, 2002, mitp-Verlag Bonn

[4] S. Beimel: Der HotKey RunPgms - ein Metaprogramm zur Ausführung der Auswertung einer klinischen Prüfung auf 'Knopfdruck'. Proceedings der 3. Konferenz für SAS-Anwender in Forschung und Entwicklung (KSFE), Heidelberg, 1999

[5] S. Beimel: Erzeugen von PDF-Dateien ohne ODS: Das Makro Lst2Pdf. Proceedings der 8. Konferenz für SAS-Anwender in Forschung und Entwicklung (KSFE), Schmalkalden, 2004