Clean Code – Wie sauber ist Ihr SAS Code?

Jan Ferdinand Knoll
viadee Unternehmensberatung GmbH
Anton-Bruchausen-Straße 8
48147 Münster
jan-ferdinand.knoll@viadee.de

Lena Rother
viadee Unternehmensberatung GmbH
Anton-Bruchausen-Straße 8
48147 Münster
lena.rother@viadee.de

Zusammenfassung

Unter dem Begriff Clean Code versteht man verschiedene Maßnahmen und Prinzipien, um die Verständlichkeit und Wartbarkeit von Software zu verbessern. Gerade bei der Entwicklung von ETL Programmen mit SAS ist Wartbarkeit ein wichtiges Thema, da die Programme häufig jahrelang im Einsatz sind – oft unter einem wechselnden Entwicklerteam. Viele der Clean Code Prinzipien stammen eher aus dem objektorientierten Kontext. Lässt sich die Clean Code Thematik daher überhaupt auf die SAS Entwicklung übertragen?
Anhand verschiedener Code Beispiele wird die Anwendung ausgewählter Clean Code Prinzipien vorgestellt sowie damit einhergehende Anforderungen an das Entwicklerteam diskutiert.

Schlüsselwörter: ETL, DataWareHouse, Clean Code

1 Motivation

Zeitdruck, Kommunikationspannen, wechselnde Anforderungen oder eine hohe Fluktuation in Entwicklerteams sind mögliche Ursachen – oder auch Ausreden – für unsauberen Code. Es muss schließlich schnell gehen, da der Kunde einen Termin genannt hat oder die Auswertung zu einer Frist fertiggestellt werden muss. Dieses Denken führt zu schnellen, jedoch nicht ökonomisch oder längerfristig effizienten Lösungen.

Die Fertigkeit, sauberen *Clean Code* zu schreiben, trägt dazu bei, dass Weiterentwicklungen und Anpassungen an diesem Code effizienter und kostengünstiger sind. Beispielsweise werden bei Änderungen mögliche Seiteneffekte reduziert, die sonst gerne zu der Äußerung *„Never change a running system!“* führen. Außerdem bleibt der Code lesbar und verständlich, was es den Kollegen, Nachfolgern oder nach einiger Zeit auch einem selbst leichter macht, den Code zu durchdringen und an den geeigneten Stellen zu erweitern.

Die Vorteile von Clean Code kann implizit jeder nachvollziehen, der schon Stunden damit zugebracht hat, einen Fehler in schlecht kommentieren, falsch benannten, unnötig komplizierten und mit verschiedenen Abstraktionsniveaus vermengten Code zu finden.

2 Clean Code

Bei der Frage nach dem, was Clean Code eigentlich ist, würden zehn verschiedene Personen zehn verschiedene Antworten geben. Es gibt daher nicht die eine Denkschule oder das eine Buch, bei dem sich alle einig sind. Aber es gibt Standardwerke, darunter *Clean Code* von ROBERT C. MARTIN, welches im Folgenden häufiger zitiert wird (vgl. [1]). Im deutschsprachigen Raum existiert zudem die Initiative *Clean Code Developer*, die für viele verbreitete Best Practices einen Ordnungsrahmen entwickelt hat (vgl. http://clean-code-developer.de/). Was diese Denkschulen gemein haben, ist die Mentalität: Code effizient, verständlich, erweiterbar und korrekt zu entwickeln. Dabei sollen die Empfehlungen nicht dogmatisch und unhinterfragt angewandt werden, sondern situationsabhängig und angemessen Verwendung finden.

Das hört sich zunächst einfach und nachvollziehbar an, dahinter verbergen sich jedoch eine Reihe von Prinzipien und Praktiken, die bei der täglich zu erledigenden Arbeit schnell vernachlässigt werden. Die Ursachen dafür sind häufig Zeitdruck, Pannen bei der Kommunikation mit Kollegen sowie fehlendes Verständnis bei den Entscheidern. Warum soll auch funktionierender und bereits erfolgreich eingesetzter Code einem Refactoring unterzogen werden? Und was interessiert es mich als Entwickler, ob andere meinen Code verstehen? Die Rechnung dazu ist simpel.

Schon während der Entwicklungsarbeiten wird Code häufiger gelesen als geschrieben [1]. Angefangen bei der Definition erster DATA steps, gefolgt von der Auslagerung von Logik über Makros, Makrovariablen oder Funktionen: Um alles zu lesen, zu erweitern und erneut zu verstehen, wird häufig hoch und wieder herunter gescrollt. Wenn die Makros und Funktionen dann zusätzlich noch mehr tun, als im Makronamen beschrieben ist, muss noch mehr Zeit in das Lesen und Verstehen investiert werden. Während man sich im eigens geschriebenen Code noch gut auskennt, hat der Kollege vielleicht unnötig viel Zeit aufgewendet.

Neben der Verständlichkeit ist auch die Erweiterbarkeit des Codes ein wichtiger Hygienefaktor. Je mehr Erweiterungen und Änderungen *mal eben* an bestehendem Code vorgenommen werden, desto unübersichtlicher und *schmutziger* wird er in der Regel. Vor allem dann, wenn der Code bereits unverständlich geschrieben ist. Die Konsequenzen äußern sich in Form von technischen Schulden und hohen Aufwänden, um bestehende Funktionsweisen zu erweitern, ohne dass ungewünschte Nebeneffekte auftreten. Die Produktivität bei der Arbeit mit unsauberem Code sinkt entsprechend (vgl. Abbildung 1).

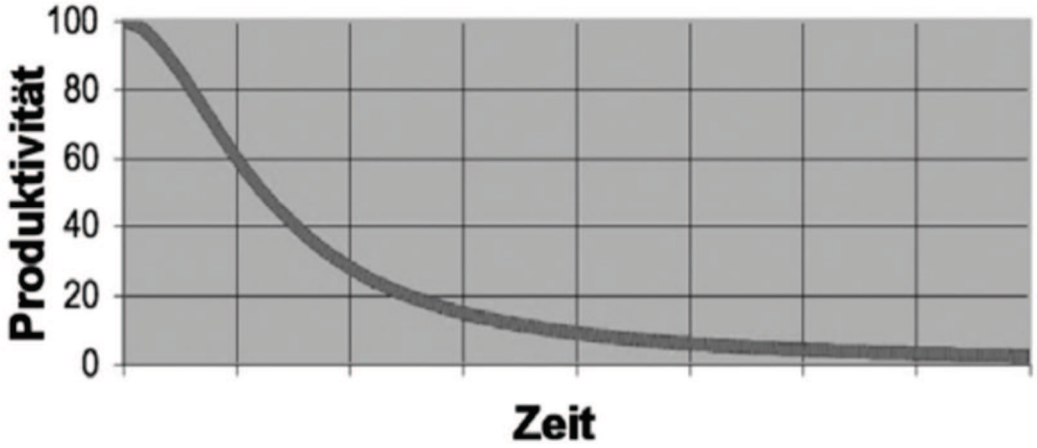

Abbildung 1: Effizienzeinbußen durch unsauberen Code nach Robert C. Martin [1]

Unter der unzureichenden Erweiterbarkeit und Verständlichkeit leidet auch die Korrektheit des Codes. Eine beispielhafte Ursache ist die Verletzung des DRY-Prinzips. Das Akronym steht für *Don't Repeat Yourself.* Es ist eines der wichtigsten Prinzipien der Softwareentwicklung und besagt, dass Wiederholungen von Codepassagen unbedingt zu vermeiden sind. Ein Kopieren und Einfügen ist zwar eine schnelle Lösung, aber genauso schnell schleichen sich Fehler ein, wenn erforderliche Anpassungen oder Fehlerbehebungen nur bei einem der Duplikate durchgeführt werden. Inkonsistenzen verleiten zu Fehlern, während ausgelagerte und abstrahierte Code-Passagen zur Verständlichkeit beitragen [1].

Neben verschiedenen Prinzipien wie DRY helfen auch diverse Praktiken, Code sauber zu halten. Ein Beispiel sind Code-Konventionen [5], die vor allem zu einer besseren Verständlichkeit des Codes beitragen. Gemeint sind damit Richtlinien innerhalb von Teams oder Instituten, mit denen beispielsweise die Formatierung, Benennung und Einstellungen der Entwicklungsumgebung standardisiert werden.

Während sich ein Großteil der Literatur zur Clean Code Thematik auf den Kontext von objektorientierten Sprachen bezieht, sind die Kern-Prinzipien und Praktiken eine Ansammlung von Best Practices, die auf jede Art der Softwareentwicklung übertragen werden kann – unserer Ansicht nach auch auf SAS.

3 Anwendbarkeit auf SAS Programme

Um die Anwendung von Clean Code Prinzipien auf SAS Programme zu veranschaulichen wird ein fiktives Beispiel-Szenario verwendet, das in Kapitel 3.1 vorgestellt wird. Der initiale SAS Code zur Verarbeitung dieses Beispiels (dargestellt im Anhang A) widerspricht verschiedenen Clean Code Prinzipen, was in den Kapiteln 3.2 bis 3.4 diskutiert wird. Der vollständige Vorschlag für eine alternative Programmierung unter Beachtung der entsprechenden Prinzipien findet sich in Anhang B.

3.1 Beispiel-Szenario

Bei dem fiktiven Beispiel-Szenario werden Daten über eine dreischichtige Verarbeitungsarchitektur eingelesen, verarbeitet und schließlich für einen Report aufbereitet ausgegeben. Inspiriert durch das sashelp-Dataset sashelp.gulfoil (vgl. [6]), handelt es in dem Beispiel inhaltlich um jährliche Öl- und Gas-Volumina, die in einem bestimmten Gebiet produziert werden. Diese Daten werden aus drei verschiedenen Quellsystemen eingelesen, um weitere Attribute angereichert, miteinander verknüpft und zu einem Report aufbereitet.

Die Gas- und Öl-Volumina werden über eine Schnittstelle bereitgestellt. Für die Gasdaten erfolgt die Datenübermittlung durch das verwaltende System immer zum 31. Januar eines Jahres. Die Öldaten werden erst einen Tag später, am 01. Februar, geliefert. Die Gebietsinformationen werden aus einer weiteren Tabelle hinzugelesen. Diese Zusammenhänge sind in Abbildung 2 dargestellt.

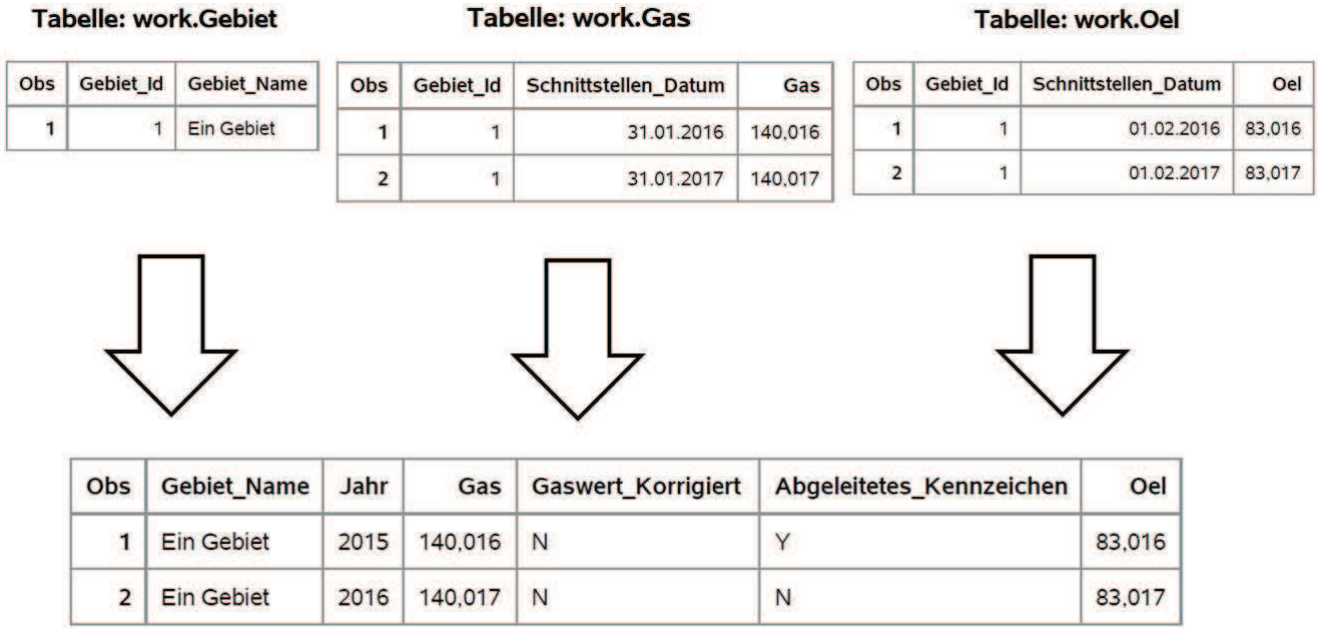

Abbildung 2: Tabellenzusammenhang im Beispiel-Szenario

In der Praxis könnte sich dieses Beispiel auf die ETL Verarbeitung in einem Data-Warehouse übertragen lassen, in dem Daten aus verschiedenen Quellsystemen eingelesen und für mehrere Anwendergruppen aufbereitet werden.

3.2 Schicht 1: Daten einlesen

Beim Einlesen von Quelldaten sind die unterschiedlichsten Formate denkbar, unter anderem Flat-Files oder regelmäßige Schnittstelleninhalte aus Datenbank-Systemen. Für dieses Beispiel wird das Einlesen der Quelldaten über drei cards statements simuliert (vgl. Anhang A für eine Übersicht des gesamten Code Beispiels). Das Einlesen der Gasdaten ist im nachfolgenden Programmbeispiel dargestellt.

```
data work.Gas;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Datum $10.
           @14 Gas 12.;
     format Datum $10.
            Gas  comma12.
            Schnittstellen_Datum ddmmyyp10.;

/* Todo: Code nach 2000er Umstellung aufräumen ! */
     if length(Datum) eq 10 then do;
          link datum_1;
     end;
     else do;
          link datum_2;
     end;
     return;

     datum_1:
          Schnittstellen_Datum = input(Datum,ddmmyy10.);
     return;
     datum_2:
          Schnittstellen_Datum = input(Datum,ddmmyy8.);
     return;
     cards;
1 31.01.2016 140016
1 31.01.2017 140017
run;
```

In diesem Programmteil sind Altlasten bei der Verarbeitung von Datumswerten zu erkennen. In der Vergangenheit wurden Jahreswerte über die Schnittstelle zweistellig angeliefert. Mit dem Jahrtausendwechsel erfolgte jedoch eine Umstellung der Jahresangaben auf vier Stellen. Der Code erlaubt über eine link Struktur noch die Verarbeitung beider Anlieferungen. Lediglich ein Kommentar weist darauf hin, dass dieser Teil nicht mehr benötigt wird und zu entfernen wäre. Im Rahmen der aktuellen Anpassung des Programms wird daher die so genannte **Pfadfinder-Regel** beachtet, die besagt, dass Code nach jeder Änderungen sauberer sein sollte als zuvor [1]. Das schließt beispielsweise das Entfernen von auskommentiertem Code ein sowie die Korrektur irreführender Kommentare. Außerdem finden bei Einhaltung dieser Praktik regelmäßige Refactorings statt: Beim Hinzufügen von neuen Funktionen, bei der Behebung von Fehlern sowie im Verlauf von Code Reviews. Nach MARTIN FOWLER ist das Refactoring ein Prozess der Veränderung von Software-Systemen in einer Weise, die das externe Verhalten nicht verändert, aber die interne Struktur des Codes optimiert. Diese kontinuierliche Verbesserung des Codes trägt dazu bei, Fehler zu finden, die Software besser zu strukturieren und den Code in Folge dessen schneller zu schreiben [4]. ROBERT C. MARTIN bezeichnet die Pfadfinder-Regel als „Random act of kindness“, wann immer bestehender Code angepasst wird [2].

Eine wichtige Voraussetzung für die Anwendung der Pfadfinder-Regel sind absichernde Tests, welche im Arbeitsumfeld von Business Intelligence leider wenig etabliert sind. Hierbei helfen Entwicklertests in Form von manuellen Vorher-/Nachher-Vergleichen der Ergebnismengen. Aber auch dann sind nicht alle Auswirkungen von Änderungen absehbar, sodass fachliche Tests erforderlich sind. Die Anwendung der Pfadfinder-Regel bietet sich also an, wenn fachliche Tests und Abnahmen beispielsweise bei Änderungsbeauftragungen vorgesehen sind.

3.3 Schicht 2: Daten verarbeiten

Im Rahmen der Programmverarbeitung werden die Gasdaten anschließend über einen weiteren DATA step aufbereitet (vgl. nachfolgendes Programmbeispiel). Auch hier lassen sich verschiedene Aspekte feststellen, die Clean Code Prinzipien wiedersprechen.

```
data work.Gas (drop=Datum);
     set work.Gas;
     format Gaswert_Korrigiert $1.;
     Gaswert_Korrigiert = 'N';
     if(Gas < 0 or Gas eq .) then do;
          Gas = 0;
          Gaswert_Korrigiert = 'Y';
     end;
     %derive_year(Schnittstellen_Datum);
run;
```

Zunächst fällt auf, dass sowohl für das eingelesene als auch für das ausgegebene Dataset der gleiche Tabellenname verwendet wird. Wahrscheinlich um den benötigten Speicherplatz zu reduzieren, wird das Eingabe-Dataset (work.Gas) bei jeder Programmausführung überschrieben. Hierdurch ist die Verarbeitung aber im Nachgang schwerer nachvollziehbar, da die Zwischenzustände nicht mehr vorhanden sind. In diesem Fall sollte das Prinzip der **Vorsicht vor Optimierungen** [7] beachtet werden. Zwar kann es bei sehr umfangreichen Berechnungen notwendig sein, die Effizienz sehr hoch zu priorisieren und auf die gute Lesbarkeit des Codes zu verzichten. Jedoch sollten die Optimierungen nur dann vorgenommen werden, wenn sie zwingend erforderlich sind und auch zu deutlichen Verbesserungen führen. Im Sinne der Langlebigkeit und Verständlichkeit des Codes ist in den meisten Fällen davon abzuraten.

Darüber hinaus werden in dem DATA step drei Operationen von ähnlicher Komplexität durchgeführt. Zunächst wird der Gaswert bei fehlendem oder fehlerhaftem Wert korrigiert. Danach wird das zugehörige Jahr berechnet und abschließend ein abgeleitetes Feld ermittelt. Die letzten beiden Operationen sind recht verschieden, aber dennoch in einem gemeinsamen Makro gekapselt, was die Nachvollziehbarkeit und Lesbarkeit erschwert. Damit widerspricht dieses Vorgehen dem Prinzip **Single Level of Abstraction (SLA)** was besagt, dass in einzelnen Bausteinen nur ein Abstraktionsniveau vorhanden sein soll. Eine beliebte Analogie dazu ist der Blick auf die Tageszeitung, in der zuerst das Allerwichtigste beschrieben ist und die Artikel im Verlauf immer detaillierter werden [5]. Programmcode sollte ähnlich aufgebaut sein, damit er vom Leser schnell erfasst

werden kann. Außerdem lädt der Verstoß gegen dieses Prinzip dazu ein, dass im Laufe der Zeit immer mehr Details zu einer Funktion dazu stoßen, schließlich ist zu diesem Zeitpunkt die klare Grenze zwischen verschiedenen Konzepten schon überschritten worden [1]. Im Rahmen der Überarbeitung des Codes sollte daher ein ähnliches Abstraktionsniveau innerhalb des DATA steps angestrebt werden.

Bei der Betrachtung des Makros selber fallen weitere Aspekte auf (vgl. nachfolgendes Programmbeispiel).

```
%macro derive_year(Datum);
     format Jahr 8.;
     Jahr = year(&Datum.) -1;
     link schwellwert;
     return;
     schwellwert:
          format Abgeleitetes_Kennzeichen $1.;
          if Gas > 10000
               /* 95, 96 und 2016 nicht! */
             and Jahr not in (1995, 1996, 2016)
          then do;
               Abgeleitetes_Kennzeichen = 'Y';
          end;
          else do;
               Abgeleitetes_Kennzeichen = 'N';
          end;
     return;
%mend;
```

Zum einen ist es - gerade bei wechselnden Entwicklerteams - hilfreich, sich auf **Source Code Konventionen** bezüglich **Namensregeln** oder **Kommentierungen** zu einigen [5]. Da die Variablen- und Tabellennamen im weiteren Programm auf Deutsch verwendet werden, sollte der Makroname entsprechend gewählt werden. Darüber hinaus trägt der Kommentar bei der Berechnung des abgeleiteten Kennzeichens nicht zum besseren Verständnis bei. Informativer wäre hier eine Begründung, warum diese Jahre (aus fachlicher Sicht) ausgeschlossen werden.

Ein weiteres Problem entsteht in der (überflüssigen) Verwendung des Links. Durch die Inklusion der Link-Definition „schwellenwert“ in das Makro würde jeder Code, der im aufrufenden DATA step hinter dem Makroaufruf angefügt würde, nicht mehr ausgeführt. Dieses unerwartete Verhalten ist ein Verstoß gegen das **Principle of least Surprise**. Wenn gemäß des Prinzips der geringsten Überraschungen gehandelt wird, stehen Funktionen im Code an der Stelle, wo der Leser sie intuitiv erwarten würde. Außerdem sind Variablen und Makros nach dem benannt, was sie repräsentieren. Die Folge von solchen Überraschungen ist das schwindende Vertrauen in die eigene Intuition und in das von Funktionsnamen versprochene Verhalten [1]. Alternativ ließe sich dies wie folgt korrigieren (vgl. nachfolgendes Programmbeispiel).

```
data work.Gas_Aufbereitet;
     set work.Gas (keep = Gebiet_Id
                          Schnittstellen_Datum
                          Gas);

     %initDatenfeld(DatenFeld=Gas, MerkerSpalte=Gaswert_Korrigiert);
     %jahrAbleiten(DatumFeld=Schnittstellen_Datum, JahrFeld=Jahr);
     %kennzeichenAbleiten(DatenFeld=Gas,  Kennzeichen-
     Feld=Abgeleitetes_Kennzeichen, JahrFeld=Jahr);
run;
```

Zunächst werden die drei Operationen über Makros von gleichem Umfang ausgelagert um den **Single Level of Abstraction** zu folgen. Die abgeleiteten Tabellen werden unter neuen Namen abgelegt, da eine **Optimierung** von Speicherplatz an dieser Stelle nicht notwendig ist.

Bei der Definition der Makros wird die Einhaltung von Namensregeln geachtet, indem ausschließlich deutsche Namen verwendet werden. Darüber hinaus wird in den Kommentaren das „Warum“ und nicht das „Was“ geschildert. Letzteres sollte durch die saubere Programmierung ersichtlich sein. Um zusätzlich Wiederholungen zu vermeiden, wird das **DRY**-Prinzip berücksichtigt. Dafür werden die Makros (vgl. nachfolgendes Programmbeispiel) so generisch definiert, dass sie durch die Übergabe der zu verwendenden Spalten sowohl für die Aufbereitung der Gas- als auch der Öldaten verwendet werden können.

```
%macro initDatenfeld(DatenFeld=, MerkerSpalte=);
     format &MerkerSpalte. $1.;
     &MerkerSpalte. = 'N';
     if(&DatenFeld. < 0 or &DatenFeld. eq .) then do;
        &DatenFeld.    = 0;
        &MerkerSpalte. = 'Y';
     end;
%mend;

%macro jahrAbleiten(DatumFeld=, JahrFeld=);
     format &JahrFeld. 8.;
     /* Angeliefert werden immer Daten des Vorjahres */
     Jahr = year(&DatumFeld.) - 1;
%mend;
```

Darüber hinaus wird die fachliche Prüfung valider Jahre über eine Funktion gekapselt (vgl. nachfolgendes Programmbeispiel), um diese an anderen Stellen wiederverwenden zu können, und nicht (evtl. sogar leicht unterschiedlich) erneut programmieren zu müssen.

```
proc fcmp outlib= sasuser.funktionen.paket;
     function validesJahr(Jahr);
          if Jahr in (1995, 1996, 2016)
          then return (0);
          else return (1);
     endsub;
run;
```

```
options cmplib=sasuser.funktionen;

%macro kennzeichenAbleiten(DatenFeld=, KennzeichenFeld=, JahrFeld=);
     format &KennzeichenFeld. $1.;
     /* fachlich werden bestimmte, nicht-valide Jahre
     ausgeschlossen */
     if (&DatenFeld. > 10000 and validesJahr(&JahrFeld.))
     then do;
          &KennzeichenFeld. = 'Y';
     end;
     else do;
          &KennzeichenFeld. = 'N';
     end;
%mend;
```

Neben der Aufbereitung der Gasdaten werden im initialen Code die Gebietsinformationen über einen weiteren Datastep aus der Tabelle work.Gebiet hinzugelesen (vgl. nachfolgendes Programmbeispiel).

```
data work.Gas   (drop=Rc)
      work.Error (drop=Rc);
     set work.Gas;

     format Gebiet_Name $17.;
     Gebiet_Name = "";

     if _N_ eq 1 then do;
          declare hash gebietsHash (dataset:'work.Gebiet');
          gebietsHash.definekey    ('Gebiet_Id');
          gebietsHash.definedata   ('Gebiet_Name');
          gebietsHash.definedone();
     end;

     Rc = gebietsHash.find();
     if Rc eq 0 then do;
          output work.Gas;
     end;
     else do;
          output work.Error;
     end;
run;
```

Auch in diesem Programmteil fallen verschiedene Aspekte auf, die unter Beachtung von Clean Code Prinzipen evaluiert werden können.

Zum einen wird zusätzlich zu der Anreicherung der Gasdaten um Gebietsinformationen auch eine Fehler-Tabelle befüllt, falls zu einer Gebiet_Id in der Gas-Tabelle kein Eintrag gefunden wurde. Durch constraints in dem zugrundeliegenden Datenbankschema wird dies jedoch bereits sichergestellt. Diese doppelte Absicherung widerspricht dem **YAGNI**-Prinzip. Das Akronym steht für "*You Ain't Gonna Need It*". Das Prinzip sagt aus, dass bei der Entwicklung zunächst nur klar definierte Anforderungen entsprechend der Priorisierung programmiert werden sollen [9]. Zusatzabfragen und –funktionen, die

eventuell in der Zukunft gebraucht werden, sind hinten anzustellen. Dazu zählen unter anderem Sicherheitsabfragen, die auf mangelndes Vertrauen oder Verständnis von Quelldaten zurückzuführen sind. Mit der Einhaltung dieses Prinzips werden Kosten für unnötige Entwicklungsarbeiten eingespart.

Zum anderen wird das Nachlesen der Gebietsinformationen über ein Hashobjekt realisiert. Dies ist neben der Fehlerausgabe ebenfalls nicht erforderlich, da ein einfacher Join der Tabellen ausgereicht hätte. Da eine einfachere Implementierung möglich ist, wird hier das **KISS**-Prinzip [7] verletzt. Hinter diesem Akronym verbirgt sich der Leitspruch „*Keep it simple, stupid*“. Die einfache Lösung ist der komplizierten Lösung vorzuziehen. Das Berücksichtigen dieses Prinzips dient der Verständlichkeit des Codes, und damit auch dessen Erweiterbarkeit und Fehlerfreiheit.

3.4 Schicht 3: Daten ausgeben

In der Ausgabeschicht wird schließlich der Report 1 über einen gesonderten Datentopf erstellt (vgl. nachfolgendes Programmbeispiel).

```
proc sql noprint;
     create table work.Report_Daten as
     select o.*,
            g.*
     from   work.Oel o,
             work.Gas g
     where  o.Gebiet_Id = g.Gebiet_Id
            /* Gasdatum plus 1 */
             and o.Schnittstellen_Datum - 1 = g.Schnittstellen_Datum
     ;
quit;
```

An dieser Stelle erfolgt ein Zugriff aus der Ausgabeschicht auf die eingelesenen und unbehandelten Öldaten aus der Einleseschicht. Um das **Law of Demeter** [8] zu beachten, sollten jedoch alle Daten in der Ausgabeschicht über die Verarbeitungsschicht eingebunden werden. Wie viele der genannten Prinzipien entstammt auch das Law of Demeter der objektorientierten Programmierung. Ursprünglich besagt es, dass Methoden nur andere Methoden kennen und aufrufen sollen, die Teil der eigenen Klasse, der erhaltenen Parameter oder selbst erzeugter Objekte sind. Damit werden Abhängigkeiten über mehrere Objekttypen verringert, sodass der Code leichter zu testen und zu warten ist. Für dieses Prinzip hat sich auch der Spruch „*Don’t talk to strangers“* durchgesetzt. Im Bereich der Datenverarbeitung, ob mit SAS oder PL/SQL, gibt es diese Art von Objekten und Objektverkettungen nicht. Es gibt jedoch Abhängigkeiten zwischen Verarbeitungsschichten und Abfragefolgen, die vermeidbar sind, wie in diesem Beispiel. Die Vorteile dieses Prinzips werden vor allem im Fehlerfall deutlich, beispielsweise bei einer unvollständigen Anlieferung der Daten. Wenn die Ausgabeschicht direkt auf der Quelldatenschicht arbeitet, sind die abgeleiteten Daten eventuell inkonsistent. Bei einem Zugriff auf die Daten der Verarbeitungsschicht vom Vortag, wären die Ergebnisse zwar nicht aktualisiert worden, dafür aber konsistent. Je komplexer die Struktur der aufbereiteten Daten, desto wichtiger wird die Reduzierung von Abhängigkeiten zwischen ein-

zelnen Verarbeitungsschritten, damit im Fehlerfall nicht die gesamte Anwendung zum Erliegen kommt.

Darüber hinaus wurde ein weiteres Mal das Prinzip der **Vorsicht vor Optimierungen** verletzt. Da bekannt war, dass die Schnittstelle mit den Gaswerten immer einen Tag vor der Schnittstelle mit den Öldaten liefert, wurde der Join über ein -1 auf die Datumswerte realisiert. Da es sich jedoch immer um Jahreswerte handelt, wäre auch ein Join über ein Aufruf der year Funktion auf beide Felder möglich gewesen. Hier wurde die „günstigere“ Variante bezogen auf Rechenintensität gewählt, die jedoch tiefere Kenntnisse der Quelldaten erfordert. Der besser lesbare Funktionsaufruf der year Funktion ist nicht besonders rechenintensiv, würde aber zur Verständlichkeit beitragen.

Des Weiteren ist der gewählte Kommentar nicht hilfreich. Ein Kommentar, warum der Join auf diese Weise vorgenommen wurde, wäre angemessener gewesen.

Neben der Abfrage über mehrere Schichten hinweg widerspricht auch das SELECT * dem **Law of Demeter**. Dieser Befehl eignet sich ideal für die Analyse von Tabellen. Ohne genauere Angaben werden alle Spalten einer Tabelle zurückgegeben. Die Ergebnismenge ist dabei unbekannt. In manchen Situationen ist eine eingeschränkte Ergebnismenge jedoch von Vorteil, beispielsweise bei der automatischen Erzeugung von Tabellen aus einer `SELECT`-Abfrage. Wenn in weiteren Arbeitsschritten eine konkrete Tabellenstruktur erwartet wird, beispielsweise nur fünf statt sechs Spalten, kann es schon zum Fehler kommen. Wenn zudem zwei Tabellen abgefragt werden, bei denen ein Spaltenname in beiden Tabellen vorkommt, wird im SAS log zwar eine Warnung wegen gleichlautenden Spalten in beiden Eingaben ausgegeben (hier z.B. die Spalte Schnittstellen_Datum), das Programm bricht aber nicht ab. Stattdessen wird der Wert genommen, der als erstes im SELECT * aufgeführt wird (in diesem Fall aus dem Dataset work.Oel). Kommen hier weitere Spalten hinzu (gleichlautend zu Spalten in work.Gas), überschreiben diese die Werte aus work.Gas in der Ausgabe. Dieses Verhalten ist nicht intuitiv und daher durch konkrete Spaltenangaben oder Alias-Vergabe zu vermeiden. Des Weiteren werden aufbauende Verarbeitungsschritte performanter, wenn nur mit ausgewählten Spalten gearbeitet wird.
Die Verwendung des SELECT * führt auch dazu, dass die Spalte Schnittstellen_Datum der Gasdaten nicht weiter verwendet werden kann, da sie durch die Öldaten überschrieben wurde. Das Feld wurde daher nur entfernt, anstatt die Belegung der Felder an einer vorgelagerten Stelle im Code zu bereinigen (vgl. nachfolgendes Programmbeispiel).

```
title "Report 1";
proc print data = work.Report_Daten;
     var Gebiet_Name Jahr Gas Gaswert_Korrigiert Abgeleite-
     tes_Kennzeichen Oel;
run;
title "";
```

Diese Problematik hätte mit Einhaltung der **Root Cause Analysis** bereits in der Aufbereitungsschicht verhindert werden können. Ähnlich wie die **Pfandfinder-Regel** zielt auch die Root Cause Analysis [7] auf eine kontinuierliche Verbesserung des Codes ab – insbesondere im Fehlerfall. Mit dieser Praktik werden nicht nur die Symptome behan-

delt, sondern zunächst die grundlegenden Ursachen für den Fehler analysiert. Auf dieser Basis sind deutlich verlässlichere Lösungsansätze ermittelbar. Diese Praktik mag kurzfristig mehr Zeit in Anspruch nehmen als eine rein kosmetische Problembehandlung. Langfristig amortisieren sich die Kosten jedoch, da der Code nicht mit gutgemeinten Bug-Fixes übersäht ist und Entwicklungsarbeiten effizient bleiben.

In diesem Fall kann schon in der Aufbereitungsschicht eine Tabelle erstellt werden, die als Basis für die Ableitung verschiedener Reports dient und bereits alle aufbereiteten Quelldaten beinhaltet (vgl. nachfolgendes Programmbeispiel). Hierbei werden die Öldaten unter Nutzung der generischen Makros ebenfalls im Vorfeld aufbereitet und unter dem Namen work.Oel_Aufbereitet zur Verfügung gestellt. Durch diese Bereinigungen ist es ebenfalls möglich, das Schnittstellen-Datum für beiden Quellen mit auszugeben.

```
proc sql noprint;
     create table work.Gas_Oel_Aufbereitet as
     select p.Gebiet_Id,
            p.Gebiet_Name,
            coalesce(g.Jahr,o.Jahr) as Jahr,
            g.Gas,
            g.Schnittstellen_Datum as Gas_Schnittstellen_Datum,
            g.Gaswert_Korrigiert,
            g.Abgeleitetes_Kennzeichen,
            o.Oel,
            o.Schnittstellen_Datum as Oel_Schnittstellen_Datum,
            o.Oelwert_Korrigiert
     from   work.Gas_Aufbereitet g
             full outer join work.Oel_Aufbereitet o
             on (g.Gebiet_Id = o.Gebiet_Id
               /* Lieferung der Quellsysteme erfolgt zu unterschied-
               lichen Zeitpunkten innerhalb des gleichen Jahres */
             and g.Jahr = o.Jahr)
             left join work.Gebiet p
             on (g.Gebiet_Id = p.Gebiet_Id)
     ;
quit;
```

Neben den konkreten Prinzipien und Anwendungsbeispielen gibt es noch eine weitere Praktik: **Erfahrungen weitergeben und austauschen**. Diese Praktik ist so selbstverständlich wie sie wichtig ist: Bei konkreten Problemstellungen auf bestehende Lösungen zurückgreifen, anstatt die Lösung von Grund auf neu zu entwickeln. Das beginnt mit der Verwendung von verbreiteten und bewährten Entwurfsmustern und reicht bis zur Abfrage von domänenspezifischem Wissen bei den Kollegen. Dabei profitieren alle Beteiligten, da nicht nur die Aufnahme, sondern auch die Weitergabe von gesammeltem Wissen zum Hinterfragen und Reflektieren anregt. Schließlich können auch bewährte Entwurfsmuster und Best Practices noch weiter ausgebaut werden. Außerdem hilft der Austausch im Team dabei, gemeinsame Standards für Lösungsmuster und Abfragestrukturen zu schaffen [3].

4 Fazit

Die Anwendung von bewährten Clean Code Prinzipien hat positive Auswirkungen auf die Langlebigkeit, Verständlichkeit und Wartbarkeit von Software. Zugleich bleibt die Beantwortung der Frage, was Clean Code eigentlich ist, eine Sache der persönlichen Erfahrung und des gegebenen Kontexts. Daher sollte die saubere Softwareentwicklung nicht dogmatisch, sondern den gegebenen Umständen entsprechend erfolgen. Wichtig ist dabei, ein gemeinsames Bewusstsein für sauberen Code zu schaffen und innerhalb eines Teams Standards zu etablieren. So können die Vorteile der Clean Code Mentalität auch bei der Entwicklung mit SAS genutzt werden.

Literatur

[1] R. C. Martin: Clean Code – Refactoring, Patterns, Testen und Techniken für saubere Code. mitp-Verlag, Heidelberg 2009.

[2] R. C. Martin: The Clean Coder – A Code of Conduct for Professional Programmers. Prentice Hall, Boston, 2011.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, 1995.

[4] M. Fowler: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston, 2000.

[5] Clean Code Developer – Orangener Grad, URL: http://clean-code-developer.de/die-grade/orangener-grad/, letzter Zugriff: 17.02.2018

[6] SAS Institute Inc. – Sashelp Data Sets, URL: https://support.sas.com/documentation/tools/sashelpug.pdf, letzter Zugriff: 17.02.2018

[7] Clean Code Developer – Roter Grad, URL: http://clean-code-developer.de/die-grade/roter-grad/, letzter Zugriff: 17.02.2018

[8] Clean Code Developer – Grüner Grad, URL: http://clean-code-developer.de/die-grade/gruener-grad/, letzter Zugriff: 17.02.2018

[9] Clean Code Developer – Blauer Grad, URL: http://clean-code-developer.de/die-grade/blauer-grad/, letzter Zugriff: 17.02.2018

Anhang A - Beispiel für unsauberen SAS-Code

A.1 Schicht 1 – Daten einlesen

```
data work.Gebiet;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Gebiet_Name $17.;
     cards;
1 Ein Gebiet
```

```
run;

options yearcutoff=1970;

data work.Gas;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Datum $10.
           @14 Gas 12.;
     format Datum $10.
             Gas  comma12.
             Schnittstellen_Datum ddmmyyp10.;

/* Todo: Code nach 2000er Umstellung aufräumen ! */
     if length(Datum) eq 10 then do;
          link datum_1;
     end;
     else do;
          link datum_2;
     end;
     return;

     datum_1:
          Schnittstellen_Datum = input(Datum,ddmmyy10.);
     return;
     datum_2:
          Schnittstellen_Datum = input(Datum,ddmmyy8.);
     return;
     cards;
1 31.01.2016 140016
1 31.01.2017 140017
run;

data work.Oel;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Schnittstellen_Datum ddmmyy10.
           @14 Oel 12.;
     format Schnittstellen_Datum ddmmyyp10.
             Oel  comma12.;
     cards;
1 01.02.2016 83016
1 01.02.2017 83017
run;
```

A.2 Schicht 2 – Daten aufbereiten

```
%macro derive_year(Datum);

     format Jahr 8.;
     Jahr = year(&Datum.) -1;
     link schwellwert;
```

```
     return;
     schwellwert:
          format Abgeleitetes_Kennzeichen $1.;
          if Gas > 10000
               /* 95, 96 und 2016 nicht! */
             and Jahr not in (1995, 1996, 2016)
          then do;
               Abgeleitetes_Kennzeichen = 'Y';
          end;
          else do;
               Abgeleitetes_Kennzeichen = 'N';
          end;
     return;
%mend;

data work.Gas (drop=Datum);
     set work.Gas;
     format Gaswert_Korrigiert $1.;
     Gaswert_Korrigiert = 'N';
     if(Gas < 0 or Gas eq .) then do;
          Gas = 0;
          Gaswert_Korrigiert = 'Y';
     end;
     %derive_year(Schnittstellen_Datum);
run;


data work.Gas   (drop=Rc)
      work.Error (drop=Rc);
     set work.Gas;

     format Gebiet_Name $17.;
     Gebiet_Name = "";

     if _N_ eq 1 then do;
          declare hash gebietsHash (dataset:'work.Gebiet');
          gebietsHash.definekey    ('Gebiet_Id');
          gebietsHash.definedata   ('Gebiet_Name');
          gebietsHash.definedone();
     end;

     Rc = gebietsHash.find();
     if Rc eq 0 then do;
          output work.Gas;
     end;
     else do;
          output work.Error;
     end;
run;
```

A.3 Schicht 3 – Daten ausgeben

```
proc sql noprint;
     create table work.Report_Daten as
     select o.*,
            g.*
     from   work.Oel o,
             work.Gas g
     where  o.Gebiet_Id = g.Gebiet_Id
            /* Gasdatum plus 1 */
             and o.Schnittstellen_Datum - 1 = g.Schnittstellen_Datum
     ;
quit;

title "Report 1";
proc print data = work.Report_Daten;
var Gebiet_Name Jahr Gas Gaswert_Korrigiert Abgeleitetes_Kennzeichen
Oel;
run;
title "";
```

Anhang B – Vorschlag für angepassten SAS-Code

B.1 Schicht 1 – Daten einlesen

```
data work.Gebiet;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Gebiet_Name $17.;
     cards;
1 Ein Gebiet
run;

options yearcutoff=1970;

data work.Gas;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Schnittstellen_Datum ddmmyy10.
           @14 Gas 12.;
     format Schnittstellen_Datum ddmmyyp10.
             Gas  comma12.;
     cards;
1 31.01.2016 140016
1 31.01.2017 140017
run;

data work.Oel;
     infile datalines;
     input @1  Gebiet_Id 1.
           @3  Schnittstellen_Datum ddmmyy10.
```

```
           @14 Oel 12.;
     format Schnittstellen_Datum ddmmyyp10.
             Oel  comma12.;
     cards;
1 01.02.2016 83016
1 01.02.2017 83017
run;
```

B.2 Schicht 2 – Daten aufbereiten

```
%macro initDatenfeld(DatenFeld=, MerkerSpalte=);
     format &MerkerSpalte. $1.;
     &MerkerSpalte. = 'N';

     if(&DatenFeld. < 0 or &DatenFeld. eq .) then do;
        &DatenFeld.    = 0;
        &MerkerSpalte. = 'Y';
     end;
%mend;

%macro jahrAbleiten(DatumFeld=, JahrFeld=);
     format &JahrFeld. 8.;
     /* Angeliefert werden immer Daten des Vorjahres */
     Jahr = year(&DatumFeld.) - 1;
%mend;

proc fcmp outlib= sasuser.funktionen.paket;
     function validesJahr(Jahr);
          if Jahr in (1995, 1996, 2016)
          then return (0);
          else return (1);
     endsub;
run;

options cmplib=sasuser.funktionen;

%macro kennzeichenAbleiten(DatenFeld=, KennzeichenFeld=, JahrFeld=);
     format &KennzeichenFeld. $1.;
     /* fachlich werden bestimmte, nicht-valide Jahre ausgeschlossen
*/
     if (&DatenFeld. > 10000 and validesJahr(&JahrFeld.))
    then do;
          &KennzeichenFeld. = 'Y';
     end;
     else do;
          &KennzeichenFeld. = 'N';
     end;
%mend;
data work.Gas_Aufbereitet;
     set work.Gas (keep = Gebiet_Id
                          Schnittstellen_Datum
                          Gas);
```

```
     %initDatenfeld(DatenFeld=Gas, MerkerSpalte=Gaswert_Korrigiert);
     %jahrAbleiten(DatumFeld=Schnittstellen_Datum, JahrFeld=Jahr);
     %kennzeichenAbleiten(DatenFeld=Gas, Kennzeichen-
Feld=Abgeleitetes_Kennzeichen, JahrFeld=Jahr);
run;

data work.Oel_Aufbereitet;
     set work.Oel (keep = Gebiet_Id
                          Schnittstellen_Datum
                          Oel);
     %initDatenfeld(DatenFeld=Oel, MerkerSpalte=Oelwert_Korrigiert);
     %jahrAbleiten(DatumFeld=Schnittstellen_Datum, JahrFeld=Jahr);
run;

proc sql noprint;
     create table work.Gas_Oel_Aufbereitet as
     select p.Gebiet_Id,
            p.Gebiet_Name,
             coalesce(g.Jahr,o.Jahr) as Jahr,
             g.Gas,
             g.Schnittstellen_Datum as Gas_Schnittstellen_Datum,
             g.Gaswert_Korrigiert,
            g.Abgeleitetes_Kennzeichen,
             o.Oel,
             o.Schnittstellen_Datum as Oel_Schnittstellen_Datum,
             o.Oelwert_Korrigiert
     from   work.Gas_Aufbereitet g
             full outer join work.Oel_Aufbereitet o
             on (g.Gebiet_Id = o.Gebiet_Id
             /* Lieferung der Quellsysteme erfolgt zu unterschiedli-
chen
                Zeitpunkten innerhalb des gleichen Jahres */
             and g.Jahr          = o.Jahr)
             left join work.Gebiet p
             on (g.Gebiet_Id = p.Gebiet_Id)
     ;
quit;
```

B.3 Schicht 3 – Daten ausgeben

```
title "Report 1";
proc print data = work.Gas_Oel_Aufbereitet;
var Gebiet_Name Jahr Gas Gaswert_Korrigiert Abgeleitetes_Kennzeichen
Oel;
run;
title "";
```