SASUnit in der Praxis: Aufbau eines Sicherheitsnetzes für angstfreie Änderungen an SAS Programmen

Johannes Lang
HMS Analytical Software
Rohrbacher Str.26
69115 Heidelberg
johannes.lang@analytical-software.de

Zusammenfassung

In diesem Beitrag wird am Beispiel des Unittesting-Frameworks SASUnit (Open Source) gezeigt, wie man für bestehende SAS Programme nachträglich ein Sicherheitsnetz aus automatisierten Tests aufbaut, um künftige Funktions- und Strukturänderungen angstfrei angehen zu können. Dabei spielt die Reduktion von Abhängigkeiten eine zentrale Rolle, um testbare Einheiten zu isolieren. Ein Versionskontrollsystem sollte eingesetzt werden, um sowohl den produktiven Code als auch die Tests abzusichern.

Schlüsselwörter: SASUnit, Testautomatisierung, Legacy code, Refactoring

1 Einführung

Nicht nur bei einem SAS Versionswechsel stellt sich die Frage: Wie stellt man sicher, dass die SAS-Programme noch genauso funktionieren wie vorher? Auch bei so genannten Standard-Makros, die viele Abteilungen pflegen und gemeinsam nutzen, sind Änderungen oft nicht „angstfrei“ möglich, da kein Sicherheitsnetz für automatische Regressionstests existiert.

1.1 Gründe für Code-Änderungen (und für die Angst davor)

Gründe für Änderungen an SAS Programmen gibt es viele: Man möchte z.B. eine neue Funktionalität hinzufügen, einen Fehler beseitigen, den Ressourcenverbrauch optimieren oder die Kompatibilität zu einer bestimmten SAS Version herstellen.
Dabei ist jedoch nicht gesagt, dass die intendierte Änderung keine unvorhergesehenen (und ggf. unerwünschten) Seiteneffekte hat: Beim Hinzufügen einer neuen Funktion kann beispielsweise der Ressourcenverbrauch ansteigen. Hierfür wünscht man sich ein „Sicherheitsnetz“ aus automatisierten Tests, welches bei jeder Programmänderung sicherstellt, dass keine bisher zugesicherte Eigenschaft unbemerkt „durchfällt“.
Ist keine solche Absicherung vorhanden, dann ist jede Veränderung des Codes mit der Angst verbunden, etwas zu beschädigen, da man nicht (unmittelbar) erkennen kann, welche Auswirkungen die Änderung hat („edit and pray“-Methode, vgl. [1]). Dies führt dazu, dass Codeänderungen oft nach folgendem Schema ablaufen:

- Der bestehende fachliche Code wird möglichst nicht verändert, sondern es wird nur neuer Code hinzugefügt.
- Die bisherigen Strukturen (z.B. Makro-Schnittstellen) werden nicht verändert, selbst wenn diese durch neue oder geänderte Funktionalität hinfällig werden.

Dadurch wird der Code immer unflexibler und unverständlicher, was die Angst (und den Aufwand) bei künftigen Änderungen weiter ansteigen lässt.

1.2 Gründe für ein Sicherheitsnetz aus automatisierten Tests

Wer den Aufwand investiert, um ein Sicherheitsnetz nachträglich aufzubauen, der profitiert in mehrerer Hinsicht davon:

- Die Hemmschwelle für künftige Codeänderungen sinkt, da durch automatisierte Tests die Auswirkungen sofort sichtbar sind. Unerwünschte Seiteneffekte können schnell behandelt werden.
- Die Bereitschaft, auf neue bzw. geänderte Anforderungen zu reagieren („Agilität“) steigt.
- Die innere Software-Qualität kann leichter erhöht werden, da auch strukturelle Änderungen durch Tests abgesichert sind.

2 Wie macht man Code besser testbar?

Um das in der Einleitung beschriebene Sicherheitsnetz aufzubauen, müssen automatisierte Tests nachträglich erstellt werden. Eine Herausforderung ist dabei, dass der bisherige Programmcode (der ja nicht testgetrieben entwickelt wurde) oft als Monolith wahrgenommen wird, der sich gegen eine Aufspaltung in kleinere, leichter testbare Strukturen wehrt. Außerdem wird das Testen bzw. Debuggen dadurch erschwert, dass man zur Laufzeit nicht alle relevanten Teile analysieren kann.
Die Lösung für dieses Dilemma besteht in folgenden Ansätzen (vgl. [1]):

- **Verringerung von Abhängigkeiten:**
  - Viele implizite Abhängigkeiten müssen durch wenige explizite ersetzt werden.
  - Beispiel: Hartcodierte Bibliotheks- und Tabellennamen werden durch Makrovariablen ersetzt, die zentral initialisiert werden.
- **Einfügen von Kontrollpunkten:**
  - Variablen bzw. Datasets müssen an kritischen Punkten analysiert werden können.
  - Beispiel: Durch den Aufruf eines Hilfsmakros zum Sichern der aktuellen Work-Tabellen in eine permanente Bibliothek stehen diese nach der Programmausführung zur Verfügung.
- **Einfügen von Aktivierungspunkten:**
  - Das Programmverhalten an relevanten Stellen muss von außen steuerbar sein.

- Beispiel: Durch das Auslagern von Programmlogik in einen Makroaufruf wird es möglich, das Verhalten zur Laufzeit zu ändern, indem das Makro separat vom aufrufenden Programm geändert wird.

2.1 Prinzipien beim Ändern von bestehenden Systemen

Auch wenn man nur einen kleinen Teil der Funktionalität ändern, aber den größten Teil davon behalten möchte, ist es in der Regel nicht ausreichend, nur die neue Funktionalität zu testen, und den restlichen Code „in Ruhe zu lassen".
Bei jeder Änderung muss man klären:

- Welche Änderung muss ich vornehmen?
- Wie kann ich ermitteln, ob ich die Änderung korrekt vorgenommen habe?
- Wie kann ich sicher sein, dass ich nichts beschädigt habe?

Besondere Vorsicht ist geboten, wenn scheinbare Fehler oder Ungereimtheiten im bisherigen System entdeckt werden: Diese sollten nicht sofort behoben werden, da möglicherweise anderer Code davon abhängt, und das eigene Systemverständnis meist nicht vollständig genug ist, um die „richtige" Lösung sofort implementieren zu können. Solche Fundstellen werden am besten dokumentiert bzw. markiert, damit sie anschließend mit Hilfe von Experten bzw. zusätzlicher Dokumentation analysiert werden können.

3 Praxisteil: Schrittweise Einführung von SASUnit

Im folgenden Abschnitt soll nun der schrittweise Aufbau des beschriebenen Sicherheitsnetzes veranschaulicht werden.

3.1 Vorbereitung des abzusichernden Systems

3.1.1 Visualisierung erstellen

Falls keine grafischen Darstellungen der Programmabläufe existieren, sollten diese unbedingt erstellt werden.
Wichtig sind vor allem folgende Diagrammtypen:

- Datenflussdiagramm: Auf welche Quelldaten wird zugegriffen, und wie werden diese schrittweise weiterverarbeitet? Welche Zieldaten werden geschrieben?
- Kontrollflussdiagramm: Welche Haupt-Verarbeitungsschritte werden mit welchen Fallunterscheidungen getroffen? Wie wirken sich diese auf den Datenfluss aus?

Dabei ist es zweitrangig, in welcher Notation oder mit welchem Werkzeug diese Diagramme erstellt werden. Im Vordergrund steht der Nutzen für das Systemverständnis, um neuralgische Punkte (Sollbruchstellen, Schnittstellen o.ä.) erkennen zu können.

3.1.2 Autoexec-Datei aufbauen

Manche Makrobibliotheken arbeiten bereits mit einer gemeinsamen Autoexec-Datei, welche gemeinsame Starteinstellungen (Pfade, Librefs etc.) kapselt.
Falls keine solche existiert, sollte eine Autoexec-Datei aufgebaut werden, um umgebungsspezifische, technische Einstellungen zentral vornehmen zu können.
In unserer Projektpraxis hat diese Datei meist folgenden Aufbau:

```
%Macro startup(Env=);

   %GLOBAL
      /* Technische Makrovariablen für Pfade etc. */
   ;
   […]
   /* Pro Umgebung einen eigenen If-Block einfügen */
   %IF ( "&Env." = "HMS_SASUNIT_LANG" ) %THEN %DO;
      […]
   %END;
   %ELSE %IF ( "&Env." = "CUSTOMER_LEV1_PROD" ) %THEN %DO;
      […]
   %END;
   […]

/* - Allgemeine Einstellungen wie Libnames, Options vornehmen
  - Setzen von Autocall-Pfaden
  - Aufruf von Hilfsmakro %createconstants zur Definition von fach-
lichen Konstanten für alle Umgebungen
  - Aufruf von Hilfsmakro %reseterr zum Zurücksetzen von Fehlercodes
*/

%Mend startup;
```

Diese Autoexec-Datei wird nicht direkt im Programm eingebunden, sondern über eine vorgelagerte **Wrapper-Datei**, welche pro Umgebung den Pfad zur echten Autoexec-Datei kennt. Dort wird die konkrete Umgebungskennung eingetragen bzw. anhand von Systemvariablen ermittelt und anschließend die passende Autoexec-Datei eingebunden.
Hier ein Beispiel für diese Wrapper-Datei:

```
%GLOBAL
   g_sEnvironment
;

/* Hier wird die aktuelle allgemeine Umgebungskennung festgelegt*/
%LET g_sEnvironment     = HMS_SASUNIT;

%Macro Start;

   %GLOBAL
      g_sAutoexecPath
   ;
```

```
   […]
   %LET l_sHostNameUpper   = %UpCase (&SysHostName.);
   %IF ("&g_sEnvironment." EQ "HMS_SASUNIT") %THEN %DO;
      %IF ("&l_sHostNameUpper." EQ "[…]") %THEN %DO;
         /* Johannes Lang */
         %LET g_sEnvironment     = HMS_SASUNIT_LANG;
         %LET g_sAutoexecPath = […];
      %END;
      […]
   %END;
   %ELSE %IF ("&g_sEnvironment." EQ "CUSTOMER_LEV1_PROD") %THEN %DO;
      %LET g_sAutoexecPath = […];
   %END;
   […]

   /* Hier wird die echte Autoexec-Datei eingebunden */
   %Include "&g_sAutoexecPath./autoexec.sas";
   %StartUp (Env  = &g_sEnvironment.);

%MEND Start;
%Start;
```

Der Hintergrund dieser zweistufigen Initialisierung ist, dass in allen Umgebungen die gleichen Dateien verwendet werden sollen, nachdem jede Umgebung einmalig registriert wurde. Die Autoexec-Dateien werden dann – zusammen mit den anderen SAS Programmen – in einem Versionskontrollsystem wie z.B. Subversion (vgl. [5]) registriert. Damit wird das Sicherheitsnetz von Anfang an für mehrere Umgebungen vorbereitet.
Bei SAS Plattform Projekten, wo ein eigener Serverkontext für die Programmausführung verwendet wird, bildet die Wrapper-Datei auch den Inhalt der benutzerspezifischen Autoexec-Datei appserver_autoexec_usermods.sas.

3.1.3 Autocall-Pfade verwenden

Wenn möglich, sollten separate SAS-Programme nicht mit %include in das Hauptprogramm eingebunden, sondern in Form von Autocall-Makros aufgerufen werden. Dadurch wird das Gesamtsystem flexibel erweiterbar, da zusätzliche Makros lediglich in den bekannten Autocall-Ordnern abgelegt werden müssen, und damit automatisch zur Verfügung stehen. Außerdem werden die Aufrufschnittstellen im Hauptprogramm expliziter, wenn zusätzlich mit Makroparametern gearbeitet wird (was grundsätzlich zu empfehlen ist).
Damit dies funktioniert, müssen nur wenige Regeln beachtet werden:

- Jede SAS-Datei enthält genau ein Makro, welches genauso heißt wie die Datei (z.B. createconstants.sas enthält nur die Definition des Makros %createconstants). Wenn plattformübergreifende Ausführung benötigt wird, dann sollte der Dateiname nur aus Kleinbuchstaben bestehen, da ansonsten die Autocall-Einbindung unter Unix-Systemen nicht funktioniert.

- Alle Ordnerpfade, unter denen Makros abgespeichert sind, müssen über die SAS-Option SASAUTOS für die aktuelle Sitzung registriert werden. Dies geschieht am Einfachsten über die vorgelagerte Autoexec-Datei, da dort die entsprechenden Pfade definiert werden.

3.1.4 Pfade und Libname-Anweisungen in Autoexec-Datei auslagern

Hartcodierte Pfade in SAS Programmen erzeugen Abhängigkeiten, welche die Testbarkeit behindern. Darum sollten sie in die Autoexec-Datei ausgelagert und pro Umgebung in Form von globalen Makrovariablen definiert werden.
Ebenso sollten Libname-Anweisungen nicht direkt in fachlichen SAS Programmen abgesetzt, sondern möglichst zentral in der Autoexec-Datei platziert werden. Eine zusätzliche Entkopplung wird erreicht, wenn der Libref selbst nur über eine umgebungsspezifische globale Makrovariable verwendet wird.
In SAS Plattform Projekten, wo mit vorab zugewiesenen Librefs gearbeitet wird, müssen die Libname-Anweisungen nur für Testumgebungen definiert werden, in denen die Ausführung der Programm nicht über einen Serverkontext erfolgt.

3.1.5 Batch-Fähigkeit herstellen

Es ist wichtig, dass die zu testenden Programme keine interaktiven Eingaben erwarten, und ihre Ausgaben in definierten Speicherorten bereitstellen. Nur dann können automatisierte Tests ihr Potenzial entfalten, da sie grundsätzlich im Hintergrund ausgeführt werden. In den meisten Fällen ist der Anpassungsaufwand hierfür gering.

3.1.6 Besonderheiten bei Verwendung von SAS Stored Processes

Werden die SAS Programme über Stored Processes aufgerufen, dann sind zusätzliche Punkte zu beachten.
Grundsätzlich sollte man für die Erhöhung der Testbarkeit anstreben, alle Spezifika der Stored Process Ausführung (z.B. automatische Makrovariablen, spezielle SAS-Optionen, Filerefs, ODS Destinations) zu kapseln, so dass die fachlichen Programme keinerlei Annahmen darüber enthalten. Letztere sollten in einer normalen SAS Workspace Server Sitzung testbar sein. Dies kann durch folgendes Vorgehen erreicht werden:

- Für jedes SAS Programm, welches als Stored Process bereitgestellt wird, existiert ein Wrapper-Programm, das die Stored Process Aufrufschnittstelle enthält. Hier ein Beispiel:

```
%GLOBAL
   g_bForce
;

%Macro stp_createmetadb (p_bForce = );

   %createMetaDB ( p_sLibrary     = &g_sMetaLib.
                  ,p_bForce       = &p_bForce.
```

```
                    );

      %PrintMessageTable;

%Mend stp_createmetadb;

%stpbegin;
%stp_createmetadb (p_bForce = &g_bForce.);
%stpend;
```

Im obigen Beispiel ist die fachliche Logik im Makro %createmetadb gekapselt. Das Wrapper-Programm stp_createmetadb.sas bildet die Schnittstelle zum Stored Process, indem es den öffentlichen Parameter g_bForce als globale Makrovariable bereitstellt. Damit kann das Wrapper-Programm als Vorlage für die Erstellung des Stored Process verwendet werden.

- Können die Systemmakros %stpbegin, %stpend nicht verwendet werden (z.B. weil manuelle Kontrolle über die ODS Destinations benötigt wird), dann kann die Stored Process Initialisierung in ein zusätzliches Hilfsmakro (z.B. %initodsdestinations) gekapselt werden, welches im Wrapper-Makro aufgerufen wird.

3.2 Erstellung eines ersten SASUnit-Testszenarios

Nachdem nun die Weichen für das Sicherheitsnetz gestellt sind, kann ein erstes Testszenario aufgesetzt werden. Dieses dient zunächst nur dazu, das zu ändernde Makro automatisiert aufrufen zu können, ohne dass Fehler im SAS-Log entstehen.
Als Vorlage können die SASUnit-Beispielszenarien („SASUnit Examples“) herangezogen werden. In diesem Beitrag wird ein SAS Makro „analyzecustomerltv.sas“ als Ausgangspunkt verwendet, welches aus der Domäne „Customer Relationship Management (CRM)“ stammt.
Die Analyse der Datenflüsse hat ergeben, dass das Programm Daten aus einer Oracle-Datenbank liest und in drei SAS Tabellen CLIENTE, CDE und PROSP aufbereitet, welche dann die Basis für die Weiterverarbeitung bilden. Am Ende wird eine CSV-Ausgabedatei und eine PNG-Grafik erzeugt.
Zum Aufbau des Sicherheitsnetzes wurde das Programm daher einmalig leicht modifiziert in der Produktivumgebung aufgerufen, um die SAS Work-Inhalte an der passenden Stelle zu sichern. Daraufhin wurde der Oracle-Datenabzug deaktiviert, und die gewonnenen Testdaten wurden zusammen mit dem geänderten SAS Programm im Versionskontrollsystem gespeichert.
Das Grundgerüst des ersten Testszenarios sieht dann wie folgt aus:

```
%MACRO analyzeCustomerLTV_test;

      /* Allgemeines Testsetup */
      %LOCAL
         l_sDescr
         l_sTestItemName
         l_sExpected
```

```
         l_sActual
      ;

      %LET l_sTestItemName       = analyzeCustomerLTV.sas;

   %t1:

      /* Testfall-spezifisches Setup */

      %LET l_sDescr              = %STR(Fehlerfreie Ausführung);

      /* Hinweis: Libref Testdata wird von SASUnit automatisch er-
zeugt, auf Basis des bei der Initialisierung angegebenen Ordnerpfa-
des */

      DATA Work.cliente;
         SET Testdata.cliente;
      RUN;
      DATA Work.cde;
         SET Testdata.cde;
      RUN;
      DATA Work.prosp;
         SET Testdata.prosp;
      RUN;

      %initTestcase       (i_object             = &l_sTestItemName.
                         ,  i_desc              = %QUOTE(&l_sDescr.)
                           );

      %analyzeCustomerLTV;

      %assertLog (i_errors     = 0
                , i_warnings   = 2
                , i_desc      = %str(Erwartete Anzahl von Fehlern und
Warnings im Log)
                  );

   %EXIT:

%MEND analyzeCustomerLTV_test;
%analyzeCustomerLTV_test;
```

Das zu testende Programm produziert unter SAS 9.4 Warnungen, da bei einigen SQL-Schritten Ziel- und Quelltabelle identisch sind. Da dies nicht sofort behoben werden konnte, wurde die Anzahl der erwarteten Warnungen zunächst angepasst, und das Thema zur späteren Bearbeitung in das Projekt-Backlog aufgenommen.

3.3 Ausführung des Testszenarios

3.3.1 Erstellung von SASUnit Startskripten

Zur Ausführung des so erstellten Testszenarios wurde einmalig ein SASUnit-Startprogramm „aa_testsuite__jl.sas“ erstellt (siehe SASUnit Examples, dort unter dem Namen run_all.sas):

```
%initSASUnit  (
 i_root        = &g_sSASBackEndRoot.
,io_target     = %SYSGET(SASUNIT_RESULTS_ROOT)
,i_overwrite   = %SYSGET(SASUNIT_OVERWRITE)
,i_project     = %SYSGET(SASUNIT_PROJECT_NAME)
,i_sasunit     = &g_sSasUnitPath.
,i_autoexec    = &g_sProgramRoot./autoexec_wrapper.sas
,i_sasautos    = &g_sTestProgramRoot.
,i_sasautos1   = &g_sPrjUtilRoot.
,i_sasautos2   = &g_sSASUtilRoot.
,i_sasautos3   = &g_sMsgFrameworkPath.
,i_sasautos4   = &g_sCallStackFrameworkPath.
,i_sasautos5   = &g_sParamFrameworkPath.
,i_sasautos6   = &g_sLockingFrameworkPath.
,i_sasautos7   = &g_sPrjProgramRoot.
,i_testdata    = &g_sDataPath.
,i_crossref      = %SYSGET(SASUNIT_CROSSREF)
,i_testcoverage  = %SYSGET(SASUNIT_COVERAGEASSESSMENT)
);

%runSASUnit (i_source = &g_sTestProgramRoot.\
                        analyzecustomerltv_test.sas);

%reportSASUnit(o_force=1
            ,  o_JUnit=1);
```

Die dort referenzierten (globalen) Makrovariablen stammen alle aus der zuvor beschriebenen Autoexec-Datei, welche als SAS-Startoption beim SASUnit-Aufruf mitgegeben wird (siehe nächstes Codebeispiel). Der Pfad zur Wrapper-Datei wird beim %initSASUnit-Aufruf angegeben (Parameter i_autoexec), da SASUnit die Tests in einer separaten SAS Sitzung ausführt, die ebenfalls initialisiert werden muss.
Der Aufruf dieses Startprogramms wird in Form eines Batch-Skriptes (Windows: .bat bzw. .cmd-Datei, Unix: .sh-Datei) konfiguriert (Beispiel: Datei zz_SASUnit_JL.cmd für Ausführung mit SAS 9.4 unter Windows):

```
REM ====== Beispiel für Windows SASUnit Startskript ========
SET SAS_VERSION=9.4
SET SAS_HOME=C:\[…]\%SAS_VERSION%
SET APP_HOME=%CD%\..\..
SET SASUNIT_ROOT=%APP_HOME%\sourcesas\sasunit
SET TEST_PROG_HOME=%APP_HOME%\sourcesastest\testprograms
SET SASUNIT_RESULTS_ROOT=%APP_HOME%\sourcesastest\testresult
```

```
SET SASUNIT_PROJECT_NAME=KSFE 2016 Testprojekt
SET USER_SUFFIX=jl

REM ========================================================
SET SYSIN_SASPGM_NAME=aa_testsuite__%USER_SUFFIX%.sas

REM Weitere System-Variablen für SASUnit
SET SASUNIT_OVERWRITE=0
SET SASUNIT_LANGUAGE=en
SET SASUNIT_HOST_OS=windows
SET SASUNIT_SAS_VERSION=%SAS_VERSION%
SET SASUNIT_COVERAGEASSESSMENT=0
SET SASUNIT_CROSSREF=0

REM Benennen der Werte für die SAS-Kommandozeilen-Parameter
SET AUTOEXEC=%APP_HOME%\sourcesas\autoexec_wrapper.sas
SET CONFIG=%SAS_HOME%\nls\en\SASV9.CFG
SET LOG=%SASUNIT_RESULTS_ROOT%\run_all.log
SET PRINT=%SASUNIT_RESULTS_ROOT%\run_all.lst

CD "%TEST_PROG_HOME%"
"%SAS_HOME%\sas.exe" -CONFIG "%CONFIG%" -AUTOEXEC "%AUTOEXEC%" -
sysin "%SYSIN_SASPGM_NAME%" -log "%LOG%" -print "%PRINT%" -
no$syntaxcheck -noovp -nosplash –icon

REM Behandlung des Rückgabecodes, siehe SASUnit Examples
[…]
```

Diese Vorlage kann von jedem SAS Entwickler im aktuellen Projekt verwendet werden, um ein eigenes Startprogramm während der Arbeit zu nutzen:

- Es wird jeweils eine Kopie von zz_SASUnit_JL.cmd, aa_testsuite__jl.sas erstellt. Das Namenskürzel „JL“ wird durch das Kürzel des jeweiligen Entwicklers ersetzt.
- In der .cmd-Datei wird die Systemvariable USER_SUFFIX entsprechend aktualisiert.
- In der .sas-Datei können dann während der Arbeit z.B. unterschiedliche Aufrufe von %runSASUnit eingetragen werden, je nachdem welche Testszenarien ausgeführt werden sollen. Die anderen Startskripte werden davon nicht beeinträchtigt.

3.3.2 Prüfung der Ergebnisse

Nachdem das Testszenario fehlerfrei lauffähig ist, werden Prüfungen für die erzeugten Ausgabedateien erstellt. Je nach Typ werden hierfür unterschiedliche SASUnit Assert-Makros verwendet:

- SAS-Tabellen (SAS7BDAT): %assertColumns, %assertLibrary
- Berichte (PDF, RTF etc.): %assertReport
- Grafiken (GIF, PNG etc.): %assertImage (Neu in SASUnit v1.6)
- Textdateien (CSV, JSON etc.): %assertText (Neu in SASUnit v1.6)

Für Text- und Bildvergleiche wird zusätzlich ein externes Programm benötigt, welches den eigentlichen Vergleich durchführt und das Ergebnis an SASUnit zurückmeldet. Hierfür ist aktuell das Open Source Tool ImageMagick voreingestellt, welches für Windows- und Unix-Systeme verfügbar ist (vgl. [2]). Nach der Installation wird es von SASUnit automatisch im Hintergrund aufgerufen. Andere Prüfprogramme können über das neue SASUnit-Makro %assertExternal eingebunden werden.

3.4 Schrittweise Änderung des Makros

Nachdem die Eckpfeiler des Sicherheitsnetzes verankert sind, kann das Makro nun mit mehr Sicherheit weiterbearbeitet werden.
Sofern noch nicht geschehen, sollten alle Bestandteile, die zum Betrieb des Sicherheitsnetzes nötig sind, in ein Versionskontrollsystem eingepflegt werden, da die Tests die gleiche Sorgfalt verdienen wie das zu ändernde Makro selbst.
Es gibt keine allgemeingültige Reihenfolge, in der anschließende Struktur- und Funktionsänderungen nacheinander ausgeführt werden sollten. Als Empfehlung kann folgendes formuliert werden:

- Grundlegende Strukturänderungen, welche für die Testbarkeit notwendig sind, sollten zuerst ausgeführt werden, bevor mit der Implementierung von fachlichen Änderungen begonnen wird.
- Nach jeder Funktionsänderung bzw. –erweiterung sollte die bisherige Struktur geprüft und ggf. überarbeitet werden („Refactoring“).

3.4.1 Strukturanpassungen

Wie in den vorherigen Abschnitten beschrieben, werden v.a. implizite Abhängigkeiten explizit gemacht und ausgelagert (z.B. Pfade in globale Makrovariablen, Logik in Untermakros). In der Regel werden Makroparameter eingeführt, um die verbleibenden Abhängigkeiten möglichst explizit auszudrücken. Dabei sollten Schlüsselwörter (keyword parameters) statt positionsbasierte Parameter verwendet werden, um die Lesbarkeit zu erhöhen.
Die zu Beginn erstellte Analyse des Daten- und Kontrollflusses liefert erste Ansätze, wo die Auslagerung von Logik in Untermakros sinnvoll ist.
Für jedes neu erstellte Untermakro gilt:

- Ein standardisierter Kommentarkopf sollte eingefügt werden, der die Funktion und die Schnittstelle präzise und verständlich beschreibt.
- Es sollte in der Versionsverwaltung registriert werden. Falls Subversion verwendet wird, dann sollten im Kommentarkopf SVN Schlüsselworte für Autor, Änderungsdatum und Revisionsnummer verwendet werden, so dass diese Metadaten automatisch beim Einchecken aktualisiert werden (vgl. [3]).
- Es sollte gleichzeitig ein Testszenario angelegt werden, welches das Untermakro mit seiner vollständigen Schnittstelle aufruft.
- Führt das Untermakro fachliche Berechnungen durch, dann werden die zugehörigen Tests in dessen Testszenario erstellt, nicht auf Ebene des übergeordneten

Makros. Damit helfen die Tests besser bei der Fehlersuche, und der Wartungsaufwand für das Sicherheitsnetz bleibt überschaubar.

3.4.2 Funktionsänderungen

Wenn die Struktur des Hauptprogramms stabil ist und die Abhängigkeiten nach bestem Wissen und Gewissen minimiert wurden, dann steht der geplanten Funktionsänderung nichts mehr im Weg. Wie anfangs beschrieben, geht diese Änderung Hand in Hand mit der Erstellung zusätzlicher Testfälle einher, die das geänderte Systemverhalten beschreiben.
Für die Erstellung von neuem Code lohnt sich unbedingt ein Blick in die Prinzipien der Clean Code Initiative (vgl. [6]), die hier nur auszugsweise aufgeführt werden:

- Führe keine Arbeit am Code durch, ohne ihn in einem Versionskontrollsystem zu pflegen.
- Mach es so einfach wie möglich, und nur so komplex wie nötig (KISS-Prinzip: Keep it simple, stupid).
- Vermeide doppelten Code (DRY-Prinzip: Don't Repeat Yourself)
- Und nicht zuletzt die Pfadfinderregel: Hinterlasse einen Ort (d.h. einen Programmabschnitt) immer in einem besseren Zustand als du ihn vorgefunden hast.

3.5 Programmeinsatz

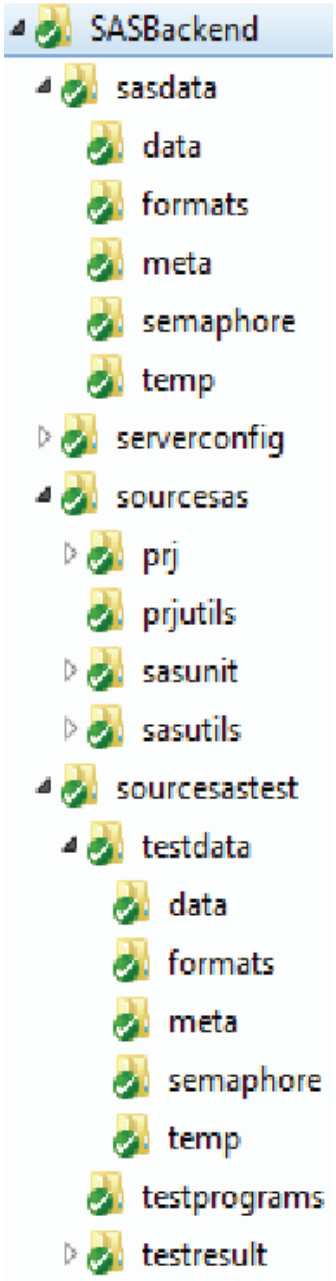
Es sollten nur solche Programmänderungen produktiv eingespielt werden, die alle automatisierten Tests bestanden haben. Falls mehrere Entwickler mit unterschiedlichen Systemen beteiligt waren, dann empfiehlt sich der Einsatz eines Continuous Integration Servers wie z.B. Jenkins, der zentral die Ausführung aller Tests in einer definierten Umgebung übernimmt (vgl. [3]).
Durch die zuvor beschriebene Autoexec-Struktur wird es möglich, die Hochstufung des getesteten Codes mit minimaler Programmanpassung durchzuführen, da lediglich die Umgebungskennung in der Wrapper-Datei umgeschaltet werden muss (vgl. 3.1.2). Dadurch wird das Risiko von Fehlern beim Programmeinsatz verringert.

4 Beispielhafte SAS Backend-Ordnerstruktur (HMS)

In diesem Abschnitt wird eine Backend-Ordnerstruktur vorgestellt, die erfolgreich für mehrere SAS Entwicklungsprojekte genutzt wurde, bei denen SASUnit fester Bestandteil war. Die Backend-Funktionen wurden jeweils über Stored Processes aufgerufen, entweder von SAS Clients oder von eigenen Oberflächen aus. Batch-Prozesse wurden (ähnlich zum SASUnit-Aufruf) über entsprechende Startskripte realisiert.
Diese Ordnerstruktur bietet Platz für allgemeine bzw. projektspezifische Hilfsmakros, Hauptmakros, Stored Process Wrapper-Makros, sowie die zugehörigen Testdaten. Konfigurationstabellen werden separat von Datentabellen in einer so genannten MetaDB

abgelegt. Formate werden über ein eigenes Makro %createFormats erzeugt und in einem separaten Ordner abgelegt.
Globale Makrovariablen werden, sofern sie umgebungsspezifisch gefüllt sind, in der Autoexec-Datei deklariert und initialisiert. Handelt es sich um fachliche, umgebungsneutrale Konstanten, dann werden sie im projektspezifischen Hilfsmakro %createConstants definiert.

Tabelle 3: Ordnerstruktur für SAS Backend



- **/sasdata**
  - gleiche Struktur wie unter **/testdata** (s.u.)
- **/serverconfig**
  - Nur falls SAS Plattform Projekt:
  - Application Server Kontext Konfiguration
- **/sourcesas**
  - Ablage von autoexec.sas, autoexec_wrapper.sas
  - **/prj**
    - Projektspezifische Hauptmakros
    - Bilden die Anwendungsfälle auf oberster Ebene ab
  - **/prjutils**
    - Projektspezifische Hilfsmakros
    - z.B. createconstants.sas, createformats.sas
  - **/sasunit**
    - Komplettes SASUnit-Framework
  - **/sasutils**
    - Allgemeine Hilfsmakros
    - z.B. Fehlerbehandlung, Parameterprüfung
- **/sourcesastest**
  - **/testdata**
    - **/data**
      - Statische Testdaten und Referenzergebnisse
    - **/formats**
      - Formatkatalog(e)
    - **/meta**
      - Steuertabellen und sonstige Meta-Tabellen
    - **/semaphore**
      - Platzhalter falls eigenes Locking-Framework genutzt wird
    - **/temp**
      - Nur für temporäre Zwischenergebnisse (z.B. Sicherung der Work-Tabellen)
  - **/testprograms**
    - Testszenarien, SASUnit-Startskripte
  - **/testresult**
    - SASUnit-Ausgabe

5 Fazit

Der schrittweise Aufbau eines Sicherheitsnetzes aus automatisierten Tests ist möglich, auch wenn es sich um bestehende SAS Programme handelt, die nicht testgetrieben entwickelt wurden. Dadurch wird die Angst reduziert, bei Codeänderungen bestehende Funktionalität zu beeinträchtigen, da die Auswirkungen durch das ständige Ausführen aller Tests bereits während der Entwicklung sichtbar werden.
Der Schlüssel beim nachträglichen Aufbau von Tests liegt darin, die vielen impliziten Abhängigkeiten zwischen den Programmteilen zu reduzieren, und durch wenige explizite zu ersetzen. Dabei kann eine standardisierte Ordnerstruktur helfen, in der alle Belange ihren Platz haben, so dass sie nicht in einem einzelnen SAS Programm untergebracht werden müssen. Die beschriebene zweistufige Autoexec-Initialisierung hilft zusätzlich dabei, die geänderten Programme mit minimaler manueller Anpassung in den Betrieb zu überführen.
Auf ein Versionskontrollsystem sollte nicht verzichtet werden, da es nicht nur den produktiven Code absichert, sondern auch das Sicherheitsnetz selbst.

Literatur

[1] M. Feathers: Effektives Arbeiten mit Legacy Code. Refactoring und Testen bestehender Software, mitp, Heidelberg 2011.

[2] ImageMagick Projekt-Homepage (Open Source), http://www.imagemagick.org, abgerufen am 17.02.2016.

[3] J. Lang: Qualitätssicherung leicht gemacht: Open Source Tools sinnvoll einsetzen und verzahnen, Tutorium auf der KSFE 2015 in Hannover, https://www.analytical-software.de/fileadmin/doc/papers/KSFE2015_Lang_Qualitaetssicherung-Toolkette_Folien1.pdf, abgerufen am 17.02.2016.

[4] SASUnit Projekt-Homepage (Open Source), https://sourceforge.net/projects/sasunit/, abgerufen am 15.02.2016.

[5] TortoiseSVN Projekt-Homepage (Open Source), https://tortoisesvn.net/, abgerufen am 15.02.2016.

[6] R. Westphal, S. Lieser: Clean Code Developer. Eine Initiative für mehr Professionalität in der Softwareentwicklung, http://clean-code-developer.de/, abgerufen am 15.02.2016.