Tipps und Tricks zur Erstellung von SQL*Plus Installationsscripten

Bereitstellung von Hilfsutilities für oft benötigte Mechanismen

Dr. Kurt Franke
Cellent Finance Solutions AG
Stuttgart

Schlüsselworte:

SQL*Plus, Software-Installation, SQL-Scripte

Einleitung

SQL*Plus an sich bietet nur sehr wenige Möglichkeiten zur Ablaufsteuerung von Installations-Scripten. Bei der Nutzung von SQL und PL/SQL stellt sich jedoch auch immer das Problem, die entsprechende Steuer-Information auf die SQL*Plus-Ebene zurückzubringen, damit je nach Bedingung bestimmte SQL-Scripte aufgerufen werden und ggf. auch mehrfach mit unterschiedlichen Parametern ausgeführt werden. Alternativ bietet sich auch die Möglichkeit, über das host-Kommando die Möglichkeiten des Betriebsystem-Shellprogramms zu nutzen, wobei dann jedoch nicht mehr alle Plattformen mit einer Codierungs-Variante bedient werden können. Deshalb wird sich die Anwendung solcher Mechanismen eher darauf beschränken, die sowieso OS-spezifischen Aktionen durchzuführen bzw. die entsprechenden Informationen von der Shell-Ebene ins SQL*Plus zu holen. Die Bestimmung des OS-Typs als erste Aktion vereinfacht nachfolgende Aktionen.

Hier werden diejenigen SQL*Plus-Features vorgestellt, die bei der Erstellung von Installationsscripten unbedingt notwendig sind, wie die Einstellung, Speicherung und Wiederherstellung von set-Variablen, spooling, host Kommandos, whenever Klauseln, explicites Schreiben des SQL-Buffers und insbesondere die Möglichkeit der Nutzung von define-Variablen inclusive Handling von Defaultwerten und ihrer Befüllung via Select Statement zur Rückgabe von SQL-Ergebnissen an SQL*Plus. Es werden auch mögliche Utility-Scripte vorgestellt, die immer wieder benötigte Funktionalität bereitstellen, z. B. Import von Environment-Variablen, speziell NLS_LANG, Script für Zwischendurch-Spoolen in andere Datei, Scripte für bedingten Aufruf anderer Scripte etc.

Anforderungen an Installationsscripte

Installationsscripte haben einige spezielle Anforderungen, die sonst eher weniger benötigt werden. Da einzufügende Daten in einem bestimmten Characterset-Encoding vorliegen, muss geprüft werden, ob in der Environment-Variable NLS_LANG auch der passende Zeichensatz ausgewählt ist, weil sonst

die Daten nicht korrekt in die Datenbank gelangen. Wenn dies nicht der Fall ist, muss ein sofortiger Abbruch des Installationsscripts mit einer geeigneten Fehlermeldung erfolgen. Es muss auch für einige Directories geprüft werden, ob darauf Schreibrechte vorhanden sind, um temporäre SQL-Dateien und/oder Logfiles dort anlegen zu können. Wenn auch nur für eines der zu prüfenden Directories die Schreibrechte fehlen, muss ebenfalls ein sofortiger Abbruch des Installationsscripts erfolgen.

Generell ist es wünschenswert, nur relevante Fehlermeldungen anzuzeigen. Z. B. ist eine Compile-Fehlermeldung irrelevant, wenn sie auf zum aktuellen Stand noch nicht auflösbaren Anhängigkeiten beruht und nach der kompletten Installation aller zu kompilierenden Objecte ein Re-Compile durchgeführt wird. Niemand sollte bei Durchführung einer Installation mit solchen Fehlermeldungen belästigt werden. Anderseits wird aber ein Mode benötigt, der während des Installationsablaufs für Debuggingzwecke mehr Ausgaben zuläßt, damit sich solche Scripte während Entwicklungsphasen überhaupt vernünftig testen lassen.

Wenn in einem von einem anderen SQL-Script aufgerufenen Unter-SQL-Script eine weitere temporäre SQL-Datei dynamisch erzeugt wird, dann ist die Original-Spool-Datei geschlossen. Wenn deren Name nicht global in allen beteiligten Scripten bekannt ist, kann sie im aufgerufenen Unter-SQL-Script auch nicht wieder geöffnet werden. Ein Mechnismus, der dies automatisch handelt, ohne dass globale Namen bekannt sein müssen, wäre wünschenswert.

Weiterhin kommt es öfter vor, dass ein bestimmtes Unter-SQL-Script nur beim Zutreffen bestimmter Bedingungen aufgerufen wird und gegebenenfalls ein anderes Script an dessen Stelle ausgeführt werden soll. Das ist zwar über SELECT und `spool` problemlos möglich, aber eine vereinfachte Verwendungs-Möglichkeit wäre ebenfalls wünschenswert.

SQL*Plus bietet keine Möglichkeit, ein SQL-Script abzubrechen – es werden immer alle Kommandos bis zum Scriptende ausgeführt. Wenn bestimmte Teile am Ende eines Scripts nicht immer ausgeführt werden sollen, kann dies nur dadurch erreicht werden, dass diese als separates Script abgespeichert werden, das nur bei Zutreffen einer geeigneten Bedingung ausgeführt wird. Es ist also eine gute Strategie, teilweise unabhängige Aufgabenblöcke in separate Scripte auszulagern, um die Möglichkeit zur bedingten Ausführung zu haben.

Wenn ein Abbruch des kompletten Installationsvorgangs gewünscht ist, kann mit einer gesetzten `whenever`-Klausel SQL*Plus komplett verlassen werden, um zu verhindern, dass ein nicht gewünschter Zustand in der Datenbank entsteht. Für Debuggingzwecke ist das aber höchst unbefriedigend. Die aktuellen Werte der `define`-Variablen und bind-Variablen stehen dann für eine Fehleranalyse nicht mehr zur Verfügung. Hier wäre eine Mechanismus interessant, der abhängig von einem Debugging-Flag das Verlassen von SQL*Plus unterdrückt. Das muss grundsätzlich über einen expliciten Scriptaufruf erfolgen, weil SQL*Plus keine Möglichkeit bietet, einen `whenever` -Handler – das könnte z. B. ein SQL-Script sein – für die Ausführung im Fehlerfalle zu deklarieren. Da ein solches Fehler-Exit-Script auch dann funktionieren muss, wenn keinerlei Schreibrechte vorhanden sind – um z. B. genau darüber zu informieren – ist wegen fehlender `spool`-Möglichkeit keine bedingte Ausführung von SQL*Plus-Kommandos möglich. Deswegen kann in einem solchen Script auch kein `exit` verwendet werden, sondern es muss darin mit einer `whenever`-Klausel, deren Aktion in einer `define`-Variablen steht, gearbeitet werden.

Einsetzbare Features von SQL*Plus

Hier werden einige weniger bekannte Features von SQL*Plus betrachtet, die bei der Erstellung von Installationsscripten hilfreich sind. Features, die nur für Reportingzwecke einsetzbar sind, werden hier nicht betrachtet.

Das `host`-Kommando bietet die Möglichkeit, beliebige Betriebssystem-Kommandos auszuführen, wobei Ergebnisse in ein SQL-Script geschrieben werden, das anschließend von SQL*Plus ausgeführt wird. Damit lässt sich im Prinzip alles einsetzen, was die jeweilige Betriebssystem-Shell vorsieht, allerdings um den Preis einer platform-spezifischen Implementierung. Um davon weitgehend unabhängig zu sein, sollte nur das unbedingt notwendige auf diese Art implementiert werden. Die Bestimmung des aktuellen Arbeitsverzeichnisses oder des Betriebssystem selbst oder die Werte von Environment-Variablen können nur auf diese Art erhalten werden. Man kann hier auch ein Script verwenden, das mit mehreren verschiedenen Betriebssystemen zurechtkommt, indem Code für alle unterstützten Operatingsysteme in geeigneter Reihenfolge mittels `host`-Kommando aufgerufen wird, um damit wenigstens auf Scriptebene eine gewisse OS-Unabhängigkeit zu erreichen. Solche Scripte werden üblicherweise in der Initialiserungsphase eines Installationsscripts aufgerufen, um die notwendige Information für die weitere Ablaufsteuerung zu erhalten.
Da SQL*Plus selbst keine Möglichkeit kennt, Dateien zu löschen, können auch temporär erzeugte Arbeits-Dateien nicht wieder gelöscht werden. Allenfalls bietet sich in SQL*Plus die direkte Möglichkeit, eine solche Datei mit einem neuen `spool`-Kommando zu leeren und keine neue Information vor dem Schließen zu schreiben. Für ein richtiges Löschen temporärer Dateien kann jedoch wiederum das `host`-Kommando eingesetzt werden, wobei im Hinblick auf OS-Unabhängigkeit dies über ein Script erfolgen sollte, das alle `host`-Kommandos für alle unterstützten Betriebssysteme in geeigneter Reihenfolge absetzt.

Das `spool`-Kommando dient dazu, Ausgaben vom SQL- und SQL*Plus-Kommandos in Dateien zu schreiben. Eine wesentliche Einschränkung ist, dass immer nur eine Datei zur gleichen Zeit geöffnet und damit beschrieben werden kann. Üblicherweise erfolgt hiermit eine Protokollierung der durchgeführten Aktionen. Es bietet jedoch auch die einzige Möglichkeit, dynamische Codeteile jeglicher Art für SQL*Plus bereitzustellen (für einzelne SQL-Statements gibt es noch eine weitere Möglichkeit). Damit wird schon offensichtlich, dass die Protokollierung öfter unterbrochen werden muss, um in solche dynamisch zu erzeugenden SQL-Scripte zu schreiben, und um danach wieder fortgeführt zu werden. Ein Mechanismus zum Flushing existiert ebenfalls nicht und das Buffering beim Schreiben kann auch nicht abgeschaltetet werden. Das ist insbesondere bei einer Überwachung des Logfiles auf Fortschritte während der durchzuführenden Installation zumindest lästig. Mittels Scripting und dem Einsatz einiger `define`-Variablen zum Festhalten des Status zwischen den Aufrufen läßt sich hier wenigsten ein einfacheres Handling im Installationsscript erreichen. Wenn auch die Einschränkungen nicht aufgehoben werden können, so ist doch zumindest ein automatischer Switch zurück auf das vorherige `spool`-File möglich, wenn ein Zwischen-`spool` beendet wird, und auch ein Flush-Aufruf läßt sich damit als Script nachbilden.
Ein häufiges Problem beim Spooling von Dateien ist das Abschneiden auf die gegenwärtig eingestellte `linesize`. Deshalb sollte grundsätzlich entweder für alle Installationsscripte gemeinsam im Einstiegsscript oder explicit vor jedem spool ein ausreichender `linesize`-Wert gesetzt werden. Wenn nicht gerade Dateien mit fester Zeilenlänge herausgeschrieben werden müssen, sollten aus Performancegründen die Leerzeichen am Ende der Zeilen mit '`set trimspool on`' abgeschnitten werden.

Das `store`-Kommando sichert die aktuellen Werte der set-Variablen restore-fähig in eine Datei und ermöglicht damit eine spätere Wiederherstellung der aktuellen Einstellungen durch einfachen Aufruf dieser Datei. Dadurch ist es nicht notwendig, dass jeder Einzelschritt zuvor immer erst alle seine

erforderlichen Settings komplett herstellt, sondern es kann ausgehend von einer Standardeinstellung zuerst ein `store`, dann die erforderlichen Settings mit nachfolgenden Aktionen und abschließend ein restore durch Aufruf der erzeugten Datei erfolgen.

Die `whenever`-Klauseln „`whenever sqlerror`“ und „`whenever oserror`“ können dazu eingesetzt werden, entweder SQL*Plus bei dem betreffenden Fehler gleich mit einen geeigneten Exit-Wert zu verlassen, oder aber über die Unterscheidung nach einem COMMIT oder ROLLBACK die Fehlersituation im Script selbst zu erkennen und geeignet zu verarbeiten. Letzteres ist vorzuziehen, weil sonst nicht einmal eine passende Fehlermeldung ausgegeben werden kann, und ein entsprechendes Handling auf der Aufrufebene vorzusehen wäre, was dann wiederum OS-spezifisch wäre. Es ist möglich, den Aktionsteil der Klausel via `define`-Variable zu setzen, wodurch ein Setzen abhängig von Bedingungen ohne Zwischen-spool ermöglicht wird.

Die `define`-Variablen sind ein wichtiges Element zur Ablaufsteuerung in Installationsscripten. Leider werden diese nicht interpretiert, wenn sie als erstes Wort in einer Zeile stehen, was die Möglichkeiten für dynamisches Scripting im wesentlichen auf den Einsatz mit `spool` erzeugter Scripte reduziert.
Es ist jedoch möglich, mit dem `define`-Kommando eine Variable zu setzen, deren Name in einer bereits existierenden anderen Variablen steht. Dies ermöglicht insbesondere die Übergabe eines Variablennamens an ein Hilfsscript als Parameter, das dann diese Variable mit einem Ergebniswert befüllen kann. Wenn die Bestimmung von Ergebniswerten mittels SQL erfolgt, kann eine `define`-Variable auch durch ein SELECT-Statement befüllt werden. Dazu muss nur zuvor für den betreffenden Columnnamen über das `column`-Kommando ein „new_value“ definiert werden (funktioniert auch mit „old_value“), der auf die betreffende define-Variable zeigt. Zweckmäßigerweise wird hier meist auch ein „noprint“ eingesetzt, um verwirrende Ausgaben aus dem SELECT zu vermeiden. Wenn keine weitere Column im SELECT ist, die ausgegeben wird, erscheint keine Ergebnis-Ausgabe, auch keine Titelzeile. Das Kommando

```
column "Cur-Time" noprint new_value now
```

gefolgt von

```
SELECT sysdate AS "Cur-Time" FROM dual;
```

trägt den aktuellen Wert von sysdate mit der gegenwärtigen Default-Character-Konvertierung in die `define`-Variable `now` ein.
Ein etwas unglückliches Verhalten von `define`-Variablen ist das automatische Prompting für einen Wert, wenn eine referenzierte `define`-Variable nicht definiert ist. An dieser Stelle wünscht man sich eher die Möglichkeit eines im Script definierbaren Defaultwertes. Dies läßt sich dadurch implementieren, dass die betreffenden `define`-Variablen mit dem `define`-Kommando in eine Spooldatei geschrieben werden, dann diese Variablen auf ihren Defaultwert gesetzt werden und anschließend die gespoolte Datei ausgeführt wird. Dieser Mechanismus führt explicit nicht zu einem Prompting für eine nicht definierte Variable. Beim Ausführen der gespoolten Datei werden genau die `define`-Variablen wiederhergestellt, die vor dem Befüllen mit dem Defaultwert schon existiert haben – alle anderen behalten den zugewiesenen Defaultwert. Ein solches Handling ist insbesondere auch für die Parameter-Variablen 1, 2, 3, usw. interessant, weil damit optionale Parameter implementiert werden können.
Es gibt 2 relevante Limits in Zusammenhang mit `define`-Variablen: Die maximale Anzahl von 2048 und die maximale Länge des Werts von 240 Zeichen.
Auch wenn die maximale Anzahl groß erscheint, kann sie beim dynamischen Anlegen der Variablen in wiederholten mit SELECT erzeugten Script-Aufrufen (LOOP-Nachbildung) bei Verwendung von Prefixen zur LOOP-Kennzeichnung im Namen doch sehr schnell erreicht werden, ebenso bei Verwendung zur Array-Nachbildung mit Array-Index im Namen, beispielsweise

my_array_element_001, my_array_element_002 usw. Da bei Erreichen dieses Limits weitere `define`-Variablen einfach nicht mehr angelegt werden (Fehler SP2-0138), ist hier ein Error-Handling nur sehr schwer möglich. Auf jeden Fall sollte sichergestellt werden, dass alle wichtigen Variablen zur Ablaufsteuerung wenigstens zuerst angelegt werden. Allgemein ist es eine gute Strategie, wenn jedes Script seine nur intern verwendeten Variablen am Ende mit `undefine` wieder entfernt. `define`-Variablen, die in mehreren Scripten verwendet werden – also quasi globale Variablen, sollten aus diesem Grunde und auch zur besseren Wartbarkeit des Codes nur in Maßen eingesetzt werden. Auch `define`-Variablen zum Sichern von Statuswerten zwischen den Aufrufen eines Scripts oder einer Gruppe von Scripten sollten nur bei echtem Bedarf eingesetzt werden.
Die Längenbegrenzung des Wertes kann z. B. bei sehr langen Datei-Pfaden zu einem Problem werden – hier kann es dann erforderlich sein, stattdessen mehrere Variable direct hintereinander einzusetzen. Auch wenn dynamisch SQL-Statements in Variablen zusammengebaut werden sollen, um sie in den SQL-Buffer zu schreiben und dort die Statements auszuführen, kann es sehr leicht zu einer Längenüberschreitung kommen. Hier kann nur sehr sorgfältige Codierung und Tests aller Grenzfälle Probleme vermeiden – ggf. muss bei kritischen Fällen ein Aufsplitten in mehrere Variablen erfolgen.

Bind-Variablen dienen dazu, aus SQL oder PL/SQL Werte zurückzuerhalten. Während bei SQL mit Einschränkungen auch durch die beschriebene Methode mit dem `column`-Kommando eine direkte Rückgabe in eine `define`-Variable möglich ist, besteht diese Möglichkeit beim Einsatz von PL/SQL nicht. Hier kann ein Rückgabewert grundsätzlich nur in eine Bind-Variable geschrieben werden. Eine Kopie des Inhaltes in eine `define`-Variable muss dann in einem zweiten Schritt mit dem SQL-Mechanismus erfolgen, indem dabei die Bind-Variable selektiert wird. Da Bind-Variablen nur in SQL und PL/SQL sowie im `print`-Kommando angesprochen werden können, ist die Rückgabe von Werten aus PL/SQL beim Erstellen von Installationsscripten auch deren hauptsächliche Verwendung.
Sie sind jedoch auch weniger in der Länge limitiert wie `define`-Variablen mit einer Maximallänge von 240, ja sogar CLOB ist als Typ möglich. Deshalb sind sie für die Datenübergabe zwischen SQL-Statements notwendig, sobald die Stringlänge von 240 überschritten wird. Auch bei wiederholter Übergabe an ein Statement aber mit unterschiedlichem Wert sind sie besser geeignet als `define`-Variablen, weil hier immer der Variablename verwendet wird und niemals eine directer Textersatz wie bei `define`-Variablen stattfindet, was in solchen Fällen insbesondere bei Parameter-Übergaben der Variablen Probleme bereitet.

SQL*Plus unterstützt einen Buffer für SQL-Kommandos, bei dem einzelne Zeilen mit SQL*Plus-Bordmitteln editiert werden können, um ein SQL-Statement in leicht abgewandelter Form einfach wiederholen zu können. Der Mechamismus als solcher wird zwar für Installationsscripte nicht benötigt, jedoch wird hiermit auch die Möglichkeit eröffnet, diesen Buffer mit jeweils einem dynamisch erzeugten SQL-Statement zu befüllen und dieses auszuführen. Dieses wird dadurch unterstützt, dass bei der Befüllung `define`-Variablen verwendet werden können. Diese stehen dann – weil bei den Buffer-Kommandos keine Interpretation erfolgt – selbst im SQL-Buffer und werden dort – anders als sonst – auch evaluiert, wenn sie als erstes Wort in einer Zeile stehen. Wenn sich der Inhalt einer im SQL-Buffer stehenden `define`-Variablen ändert, dann ändert sich auch das auszuführende Statement. Diese Methode ermöglicht es, SQL-Statements, insbesondere auch SELECT's, dynamisch aufzubauen und auszuführen, ohne sie zunächst mit dem `spool`-Kommando in eine separate Datei zu schreiben. Der SQL-Buffer selbst wird hierzu nur mit wenigen SQL*Plus-Kommandos modifiziert: „`del 1 last`“ zum kompletten Leeren sowie „`input`“ und „`append`“ zum Befüllen mit einem komplett neuen Statement. Mit dem `save`-Kommando kann der Inhalt auch in eine Datei geschrieben werden, ohne dass dadurch die aktuelle `spool`-Datei beeinflußt wird. Einzelne SQL-Statements können also zusammengebaut und aufgerufen werden, auch als separate Datei, ohne ein laufendes spool zu unterbrechen. Eine Einschränkung bei der Verwendung von Variablen im SQL-Buffer ist

deren Längenbeschränkung. Durch Verwendung mehrerer Variablen kann dies jedoch weitgehend umgangen werden, wobei das Erstellen von Statements dabei jedoch komplizierter wird.

Wie in einer Unix-Shell das ‚\'-Zeichen direct vor einem Zeilenumbruch wirkt in SQL*Plus das ‚-'-Zeichen direct vor einem Zeilenumbruch: Der Zeilenumbruch wird dadurch maskiert und ist für das Script quasi nicht vorhanden. Dies kann dazu dienen, Code lesbarer zu gestalten. Da dieses Feature aber offenbar weniger bekannt ist, führt es auch oft zu einem Fehler mit unbewußt auskommentierten Zeilen. Bei einer mit ‚--' begonnenen Kommentarzeile wirkt ein ‚-'-Zeichen direct vor dem Zeilenende nämlich ebenfalls und als Folge gehört auch die nächste Zeile zum Kommentar. Beispiel:

```
-----------------------------------------------------------------
prompt Diese Zeile gehört zum Kommentar und wird nicht ausgegeben !
```

Informationstransfer von anderen Arbeitsebenen in die SQL*Plus-Ebene und umgekehrt

Mit dem Vorhalten der Steuer-Information auf SQL*Plus-Ebene ist hier das Ablegen in `define`-Variable gemeint.

Von der Betriebsystemebene aus gibt es es grundsätzlich nur die Möglichkeit, Ausgaben in ein SQL-Script in der Form ‚define VARIABLE = "WERT"' zu schreiben und dieses Script dann in SQL*Plus auszuführen. Damit sind diese `define`-Variablen neu definiert oder geändert. Da hier nur der Weg über eine temporäre Datei zum Übertragen der Information zur Verfügung steht, besteht bei fehlenden Schreibrechten keine Möglichkeit, Information von Betriebsystemebene in SQL*Plus zu übernehmen.

Von der SQL-Ebene aus kann mit dem oben beschriebenen column-Definitions-Mechanismus direkt aus einem SQL-Statement in eine – oder auch mehrere – `define`-Variablen geschrieben werden. Alternativ kann auch wie beim `host`-Kommando die Ausgabe von SELECT-Statements in der Form ‚`define VARIABLE = "WERT"`' via `spool`-Kommando in eine SQL-Datei geschrieben werden, die dann aufgerufen wird.

Von der PL/SQL-Ebene gibt es die Möglichkeit, einen zu übergebenden Wert zunächst in eine SQL*Plus-Bind-Variable zu schreiben. Damit ist dieser Wert auf der SQL-Ebene verfügbar und kann wie eben beschrieben auf die SQL*Plus-Ebene übertragen werden. Als weitere Möglichkeit kann unter Verwendung des dbms_output-Datenbank-Packages die Information in der Form ‚`define VARIABLE = "WERT"`' via `spool`-Kommando in eine SQL-Datei geschrieben werden, die dann aufgerufen wird.

Die Informationsübergabe von der SQL*Plus-Ebene in alle anderen Ebenen ist grundsätzlich unkompliziert – hier kann einfach die entsprechende `define`-Variable eingesetzt werden.

Beispiele für Hilfs-Utilities

An dieser Stelle werden einige SQL-Scripte vorgestellt, die immer wieder benötigte Funktionalität als einfachen Aufruf bereitstellen. Es werden dabei nicht die kompletten Scripte betrachtet, sondern nur die wesentlichen Schritte. Speichern, Modifizieren und Wiederherstellen von SQL*Plus-Einstellungen (`set`-Variablen) sowie undefine von verwendetenVariablen und Parametern und Löschen von verwendeten column-Definitionen und Dateien am Ende eines Scripts werden bei den einzelnen

Utility-Scripten nicht näher beschrieben. Nichtsdestotrotz sind diese Schritte in den Scripten vorhanden und gerade bei Utility-Scripten unverzichtbar, wenn man mögliche Side-Effekte von vornherein vermeiden will. Unmodifizierte Einstellungen würden im Regelfall auch dazu führen, dass beim Ausführen von Spooldateien eine gewisse Anzahl nicht relevanter Fehlermeldungen ausgegeben würde, z. B. über nicht vorhandene Kommandos.

a) Bestimmen des lokalen Betriebssystems: `import-OS-info.sql`

Dieses Script setzt Schreibrechte im aktuellen Arbeitsverzeichnis voraus und darf deshalb erst aufgerufen werden, wenn sichergestellt ist, dass diese vorhanden sind.
Die Bestimmung des lokalen Betriebssystems muss naturgemäß mit Betriebssystemmitteln erfolgen – d. h. das Script ist nur unter den Betriebssystemen funktionsfähig, für die der erforderliche Code integriert wurde. Im vorliegenden Fall sind das die verschiedenen Versionen von MS-Windows und alle Unix/Linux-Varianten in ihren verschiedenen Versionen. Auf dem Windows-System darf ein Cygwin oder ein anderes Unix-Interface installiert sein, dessen Shell im PATH zu finden ist.
Der Typ des lokalen Betriebssystems wird durch Auswertung des aktuellen Arbeitsverzeichnisses bestimmt und kann die Werte ‚UNIX', ‚WIN' oder ‚UNKNOWN' annehmen. Damit ein Windows mit Cygwin den korrekten OS-Typ liefert, wird zuerst mit einem host-Kommando in Unix-Syntax das Arbeitsverzeichnis als ‚define import_cur_dir_env_var = \" `pwd `\" in die temporäre SQL-Datei geschrieben und danach in Windows-Syntax als ‚define import_cur_dir_env_var = %CD%' an die gleiche Datei angehängt. Damit überschreibt ein existierender Windowswert einen eventuell vorhandenen Wert aus einer Cygwin-Installation, sodass Windows korrekt erkannt wird. Um ein Anwachsen der Datei unter Windows ohne eine Cygwin-Installation bei Mehrfach-Aufrufen zu verhindern, wird zuerst die temporäre SQL-Datei mit einem host-Kommando in Windows-Syntax geleert.
Die beiden Parameter sind optional und dienen dazu, benutzerspezifische Variablennamen für die Rückgabe von LOCAL_OS_TYPE bzw. – da sowieso schon abgefragt – CURRENT_WORKING_DIR zu übergeben.

Eindeutige Namen für temporäre Dateien:

```
column "Store File Name" noprint new_value –
                           import_os_type_stored_set
column "Default Param File Name" noprint new_value –
                                  import_os_type_def_param_file
column "Variable Input File Name" noprint new_value –
                                   import_os_type_input_var

SELECT 'import-OS_type-stored-set_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Store File Name",
       'import-OS_type-defparm_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Default Param File Name",
       'import-OS_type-input-var_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.variable' AS
         "Variable Input File Name"
FROM dual;

store set &import_os_type_stored_set
```

Parameter Default Werte:

```
spool &import_os_type_def_param_file

define 1

define 2

spool off

define 1 = 'LOCAL_OS_TYPE'

define 2 = 'CURRENT_WORKING_DIR'

@&import_os_type_def_param_file
```

Information von Betriebssystemebene holen:

```
define import_cur_dir_env_var = ''

host cmd /c type nul >&import_os_type_input_var 2>nul

host sh -c 'echo define import_cur_dir_env_var = \"`pwd`\" -
                          >&import_os_type_input_var 2>/dev/null'

host cmd /c echo define import_cur_dir_env_var = %CD% -
                          >>&import_os_type_input_var 2>nul

-- load the written definition into SQL*Plus
@&import_os_type_input_var
```

Weitere Berechnung und Schreiben der Werte in die vorgegebenen `define`-Variablen

```
define &2 = &import_cur_dir_env_var

column "Local OS Type" noprint new_value &1

SELECT decode(substr('&import_cur_dir_env_var',1,1),
              '/','UNIX',
              decode(substr('&import_cur_dir_env_var',2,1),
                     ':','WIN',
                     '--- UNKNOWN ---'
                    )
             ) AS "Local OS Type"
 FROM dual
;
```

b) Import von Environment Variablen (NLS_LANG): `import-NLS_LANG-setting.sql`

Die wichtigste Environment-Variable für Installations-Scripte ist wohl NLS_LANG, weil damit festgelegt wird, welcher Characterset in SQL*Plus verwendet wird. Insbesondere beim Einfügen von Daten in Initialisierungs-Scripten ist es wichtig, dass der vordefinierte Characterset zu den in den INSERT-Statements vorliegenden Daten passt. Da diese Einstellung ausserhalb von SQL*Plus erfolgt, sollte unbedingt überprüft werden, ob der vorliegende Wert geeignet ist, um sicherzustellen, dass alle während der Installation eingefügten Daten auch korrekt in der Datenbank abgelegt werden. Dazu wird die Environment-Variable NLS_LANG ins SQL*Plus hereingeholt und kann dann gegen eine Vorgabe geprüft werden. Das hier vorgestellte Script kann allgemein als Beispiel zum Importieren von benötigten Environment-Variablen gelten. Dieses Script setzt Schreibrechte im aktuellen Arbeitsverzeichnis voraus und darf deshalb erst aufgerufen werden, wenn sichergestellt ist, dass diese vorhanden sind. Die beiden Parameter sind optional und dienen dazu, benutzerspezifische Variablennamen für die Rückgabe von NLS_LANG bzw. NLS_CHARACTER_SET zu übergeben.
Die Abfrage von Environment-Variablen ist nicht in SQL*Plus selbst implementiert und kann deshalb nur mit Betriebssystemmitteln erfolgen – d. h. das Script ist nur unter den Betriebssystemen funktionsfähig, für die der erforderliche Code integriert wurde. Im vorliegenden Fall sind das die verschiedenen Versionen von MS-Windows und alle Unix/Linux-Varianten in ihren verschiedenen Versionen. Auf dem Windows-System darf ein Cygwin oder ein anderes Unix-Interface installiert sein, dessen Shell im PATH zu finden ist.

Eindeutige Namen für temporäre Dateien:

```
column "Store File Name" noprint new_value –
                           import_nls_lang_stored_set
column "Default Param File Name" noprint new_value –
                                   import_nls_lang_def_param_file
column "Variable Input File Name" noprint new_value –
                                    import_nls_lang_input_var

SELECT 'import-NLS_LANG-stored-set_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
           "Store File Name",
       'import-NLS_LANG-defparm_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
           "Default Param File Name",
       'import-NLS_LANG-input-var_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.variable' AS
           "Variable Input File Name"
FROM dual;

store set &import_nls_lang_stored_set
```

Parameter Default Werte:

```
spool &import_nls_lang_def_param_file

define 1

define 2

spool off
```

```
define 1 = 'NLS_LANG'

define 2 = 'NLS_CHARACTER_SET'

@&import_nls_lang_def_param_file
```

Werte von Betriebssystemebene holen:

```
define import_nls_lang_env_var = ''

host cmd /c type nul >&import_nls_lang_input_var 2>nul

host sh -c 'echo define import_nls_lang_env_var = \"$NLS_LANG\" -
                          >&import_nls_lang_input_var 2>/dev/null'

host cmd /c echo define import_nls_lang_env_var = %NLS_LANG% -
                          >>&import_nls_lang_input_var 2>nul

-- load the written definition into SQL*Plus
@&import_nls_lang_input_var
```

Weitere Berechnung und Schreiben der Werte in die vorgegebenen `define`-Variablen

```
define &1 = &import_nls_lang_env_var

column "NLS Character Set" noprint new_value &2

SELECT nvl(substr('&import_nls_lang_env_var',
                  instr('&import_nls_lang_env_var','.')+1),'-----')
         AS "NLS Character Set"
 FROM dual;
```

c) Directory-Schreibrechte prüfen: `dir-write-check.sql`

Durch Aufruf dieses Scripts ist eine Überprüfung möglich, ob in einem zu übergebenden Directory Schreibrechte existieren. Ggf. kann dann rechtzeitig ein Abbruch des Installationsscripts mit einer aussagekräftigen Fehlermeldung erfolgen. Das Script benötigt 2 Parameter. Im ersten wird das zu prüfende Verzeichnis übergeben – ein expliciter Leerstring prüft das aktuelle Arbeitsverzeichnis.
Ein nicht leerer Wert muss grundsätzlich mit dem für das aktuelle Betriebssystem gültigen Pfad-Separator abgeschlossen werden. Dadurch wird vermieden, dass betriebssystem-spezifische Werte im Script verwaltet werden müssen und das Script ist somit für alle von SQL*Plus unterstützten Betriebssysteme einsetzbar.
Im zweiten Parameter wird eine Variable übergeben, die das Prüfungsergebnis aufnimmt – 1 für Schreibrecht, 0 für kein Schreibrecht.
Das Script verwendet für eigene Zwecke keine Dateien und benötigt deshalb auch keine Schreibrechte auf irgendeinem Directory.
Die Prüfung erfolgt durch Anlegen und Beschreiben einer neuen Testdatei mit sekundengenauem Datum im Namen. Die zuvor gesetzte Klausel ‘WHENEVER OSERROR ROLLBACK’ sorgt dafür,
dass eine nicht mit COMMIT abgeschlossene Änderung eines extra angelegten Datensatzes in PLAN_TABLE zurückgerollt wird. Das nachfolgende SELECT auf diesen Satz liefert dann das

Ergebnis: Schreibrecht, wenn das letzte Update sichtbar ist, und kein Schreibrecht, wenn das letzte Update zurückgerollt wurde. Die GLOBAL TEMPORARY TABLE PLAN_TABLE wird eingesetzt, weil hier jeder DB-User Schreibrechte hat, auch wenn er keine eigenen Tabellen anlegen darf.

Variablen Initialisierung, Bestimmung Testdateiname, Erzeugung Prüf-Datensatz in PLAN_TABLE

```
whenever oserror continue none

define dir_to_check = '&1'

define def_var = '&2'

define state_value_key = 'SQL*Plus-dir_write-check'

define ext = '.out'

column "Current Date" noprint new_value now
SELECT to_char(sysdate,'YYYYMMDDHH24MISS') AS "Current Date"
 FROM dual;

define testfile = 'write_check.&now&ext'

column "Position Value Column" noprint new_value &def_var

-- global temporary plan_table is writable by each user
INSERT INTO plan_table
  (statement_id,
   remarks,position)
VALUES('&state_value_key',
       nvl('&dir_to_check','___current_working_directory___'),0);
;

COMMIT;
```

Änderung an Prüf-Datensatz, Transaction darf nicht abgeschlossen werden

```
UPDATE plan_table
 SET position = 1
 WHERE statement_id = '&state_value_key'
   AND remarks =
         nvl('&dir_to_check','___current_working_directory___')
;

whenever oserror continue rollback;
```

eigentlicher Schreibtest:

```
spool &dir_to_check&testfile
SELECT 'test write to new file in ' ||
       decode('&dir_to_check',
              NULL, 'current working directory',
              'directory "&dir_to_check"') AS "write_check"
```

```
 FROM dual;
spool off
```

Prüfung des Testergebnisses, Übertragen in Rückgabe-Variable

```
whenever oserror continue none

SELECT position AS "Position Value Column"
 FROM plan_table
 WHERE statement_id = '&state_value_key'
   AND remarks =
         nvl('&dir_to_check','___current_working_directory___')
;
```

Entfernen des Prüf-Datensatzes

```
DELETE FROM plan_table
 WHERE statement_id = '&state_value_key'
   AND remarks = nvl('&dir_to_check','___current_working_directory___')
;

COMMIT;
```

d) Löschen von Dateien: `delete-file.sql`

Wie oben bereits beschrieben, kann das Löschen von Dateien nur unter Einsatz von Betriebssystemmitteln erfolgen. Das heißt, dass jegliche derartige Lösung immer nur die Betriebssysteme abdeckt, für die sie entwickelt wurde. Im vorliegenden Fall sind das die verschiedenen Versionen von MS-Windows und alle Unix/Linux-Varianten in ihren verschiedenen Versionen. Auf dem Windows-System darf ein Cygwin oder ein anderes Unix-Interface installiert sein, dessen Shell im PATH zu finden ist.

```
host cmd /c del "&&1" 2>nul
host sh -c 'rm -f nul "&&1" 2>/dev/null'
```

Zunächst wird mit Windows-Syntax versucht, die übergebene Datei zu löschen, wobei Fehlermeldungen auf die nul-Device umgeleitet werden. Auf allen echten Unix-Systemen führt dies dazu, dass eine Datei `nul` im aktuellen Arbeitsverzeichnis angelegt wird. Deshalb wird im nachfolgenden Aufruf mit Unix-Syntax zusätzlich zum Löschversuch der übergebenen Datei auch noch die Datei `nul` im aktuellen Arbeitsverzeichnis gelöscht, wobei wiederum eine Umleitung der Fehlermeldungen auf die null-Device erfolgt. Da die Unix-Syntax für die null-Device keinem zulässigen Dateinamen unter Windows entspricht, wird hier unter Windows auch keine Datei angelegt. Dieses Verhalten war maßgeblich für die Reihenfolge der Löschversuche – zuerst Windows und danach Unix – zur Vermeidung des Entstehens von Artefakten durch das Script unter bestimmten Betriebssystemen.

e) Automatisches Schachteln beim Spooling: `lspool.sql` und `lspool-off.sql`

Das Script-Paar `lspool.sql` und `lspool-off.sql` ermöglicht ein geschachteltes Spooling. Normalerweise muss nach einem Zwischen-Spool in eine andere Datei der Name der vorherigen Spooldatei in jedem Unter-Script mit Zwischen-Spool bekannt sein, damit diese ursprüngliche Spooldatei für weiteres Spooling erneut im Append-Mode geöffnet werden kann. Dies bedeutet entweder die Verwendung globaler Variablen für die Namen von geöffneten Spool-Dateien, oder die Übergabe des jeweils aktuellen Spooldatei-Namens an alle Unterscripte.
Die durchgängige Verwendung des Scriptpaaares für alle Spool-Vorgänge vermeidet dies durch script-interne Verwaltung der Spooldatei-Namen auf allen Schachtelungsebenen. Die `define`-Variablen mit den Namen ___LSPOOL_LEVEL und ___LSPOOL_FILE_LEVEL_# mit '#' als Integerwert werden dazu für diesen Mechanismus für exclusive Nutzung benötigt und dürfen deshalb nicht anderweitig verwendet werden, wenn dieser Mechanismus eingesetzt wird.
Das Script `lspool.sql` erhält als Parameter einen Dateinamen und als zweiten Parameter wie das `spool`-Kommando optional einen der Mode-Werte CREATE, REPLACE oder APPEND. wobei REPLACE der Defaultwert ist. Der besondere Wert OFF als Dateiname ist explicit veboten. `lspool.sql` eröffnet grundsätzlich eine neue Schachtelungsebene – falls noch keine existiert, die erste – inkrementiert dazu die `define`-Variable ___LSPOOL_LEVEL und merkt sich den neuen Spoolfile-Namen in der Variablen ___LSPOOL_FILE_LEVEL_&___LSPOOL_LEVEL mit dem schon inkrementierten Level. Eine zuvor bereits geöffnete Spooldatei wird geschlossen und die neue im angegebenen Mode geöffnet. Die damit erreichbare Schachtelungstiefe ist nur durch SQL*Plus-Limits begrenzt, im speziellen durch die maximal mögliche Anzahl von `define`-Variablen.
Das parameterlose Script `lspool-off.sql` schließt die aktuelle Spooldatei, entfernt die `define`-Variable mit dessen Namen durch Aufruf von `undefine` und dekrementiert die `define`-Variable ___LSPOOL_LEVEL. Anschließend wird des Datei des dekrementierten Levels – falls dieser größer als 0 ist – im APPEND-Mode erneut geöffnet, so dass ab sofort die Ausgaben wieder in die vorhergehende Spooldatei gelenkt werden.

Durch die Verwendung der lspool-Scripte ergibt sich automatisch eine vereinfachte Möglichkeit für ein Flushing der aktuellen `spool`-Datei:
Flush-Script für die aktuelle spool-Datei: `lspool-flush.sql`

Der Aufruf dieser Datei erfolgt ohne Parameter, weil die Namen der spool-Dateien vom lspool-Mechanismus verwaltet werden und damit bekannt sind. Dieses Script ist nahezu identisch zum Script `lspool-off.sql` – lediglich das Dekrementieren des lspool-Levels findet in diesem Script nicht statt. Damit wird die aktuelle `spool`-Datei einfach geschlossen und erneut geöffnet, womit automatisch ein Flushing verbunden ist. Der Mehrwert dieses Scripts liegt darin, dass durch den parameterlosen Aufruf ein Flushing auf in Unter-Scripten erfolgen kann, in denen der Name der aktuellen `spool`-Datei nicht bekannt zu sein braucht. Voraussetzung ist lediglich, dass durchgängig der lspool-Mechanismus verwendet wird. Da für eine Installation alles ausser den hier beschriebenen Hilfs-Scripten spezifisch codiert wird, stellt das kein Problem dar. Insbesondere setzt der Einsatz des lspool-Mechanismus eine durchgängige Verwendung voraus, wenn das gewünschte Verhalten erreicht werden soll.

lspool.sql :

```
column "LSpool Level" noprint new_value ___LSPOOL_LEVEL

define lspool__init_chk = 'lspool__init.check'

-- first stop the previous spool if any
```

```
spool off

spool &lspool__init_chk

define ___LSPOOL_LEVEL

define 2

spool off

define ___LSPOOL_LEVEL = 0

define 2 = 'REPLACE'

@&lspool__init_chk

-- increment the ___LSPOOL_LEVEL first to get the correct file names
SELECT ltrim(to_char(to_number('&___LSPOOL_LEVEL') + 1)) AS "LSpool Level"
 FROM dual
;

-- store the name of the new spool file
define ___LSPOOL_FILE_LEVEL_&___LSPOOL_LEVEL = "&1"

spool "&1" &2
```

lspool-off.sql :

```
column "LSpool Level" noprint new_value ___LSPOOL_LEVEL

column "LSpool Prev Spool Cmd" noprint new_value lspool__prev_spool_cmd

define lspool__init_chk = 'lspool__init.check'

define lspool__eval = 'lspool__eval.output'

define lspool__ampersand = '&'

define lspool__file_level_var_prefix = "___LSPOOL_FILE_LEVEL_"

define lspool__prev_spool_cmd = ''

-- first stop the previous spool if any
spool off

spool &lspool__init_chk

define ___LSPOOL_LEVEL

spool off

define ___LSPOOL_LEVEL = 0

@&lspool__init_chk
```

```
-- undefine the name for the current level
undefine ___LSPOOL_FILE_LEVEL_&___LSPOOL_LEVEL

-- decrement ___LSPOOL_LEVEL first to get the correct previous file names
SELECT CASE WHEN to_number('&___LSPOOL_LEVEL') - 1 < 0
         THEN '0'
         ELSE ltrim(to_char(to_number('&___LSPOOL_LEVEL') - 1))
         END AS "LSpool Level"
 FROM dual
;

SELECT
    'spool ' ||
    '"&lspool__ampersand&lspool__file_level_var_prefix&___LSPOOL_LEVEL"' ||
    ' append' AS "LSpool Prev Spool Cmd"
 FROM dual
 WHERE to_number('&___LSPOOL_LEVEL') > 0
;

spool &lspool__eval

prompt &lspool__prev_spool_cmd

spool off

@&lspool__eval
```

f) Bedingte Script-Aufrufe: `ifstart.sql` und `ifelsestart.sql`

Natürlich lassen sich bedingte Script-Starts immer dadurch realisieren, das das Startkommando über ein SELECT-Statement mit der Bedingung als WHERE-Klausel in eine Spooldatei geschrieben und diese dann aufgerufen wird. Wenn die WHERE-Bedingung keinen Treffer liefert, ist die Spooldatei einfach leer. Das erscheint aber für einen Scriptaufruf abhängig von einer Bedingung doch etwas umständlich.
Die beiden Scripte ermöglichen ein vereinfachtes Handling in solchen Fällen. `ifstart.sql` erhält als ersten Parameter eine Bedingung und als zweiten Parameter das bedingt aufzurufende Script. Optional können bis zu 8 weitere Parameter übergeben werden, die an das aufzurufende Script durchgereicht werden. `ifelsestart.sql` erhält als ersten Parameter eine Bedingung, als zweiten Parameter das bedingt aufzurufende Script und als dritten Parameter das Script, dass auszuführen ist, wenn die Bedingung nicht zutrifft. Optional können bis zu 8 weitere Parameter übergeben werden, die an das aufzurufende Script durchgereicht werden. Wichtig ist hier, dass unabhängig davon, ob das Script für zutreffende oder das für nicht zutreffende Bedingung aufgerufen wird, die Parameter identisch durchgereicht werden. Beide Scripte müssen also mit den gleichen Parametern umgehen können, wobei jeweils bestimmte Parameter auch ignoriert werden können.
Der bedingte Scriptaufruf erfolgt hier nicht durch Spooling und anschließenden Aufruf der Spooldatei, sondern direct über eine `define`-Variable nach dem ‚@'-Zeichen. Daraus ergibt sich insbesondere für `ifstart.sql`, dass immer ein existierendes Script aufgerufen werden muss, wenn eine Fehlermeldung vermieden werden soll, also auch dann, wenn die Bedingung nicht zutrifft. Insbesondere der Fall einer leeren Variablen für den Scriptnamen muss unbedingt vermieden werden, weil SQL*Plus sonst versucht, ein Script mit dem Namen des aktullen Scripts in SQLPATH zu finden und zu starten, was zu unerwarteten Ergebnissen führen kann. Deshalb wird hierfür ein leeres Script

(nur mit Kommentaren versehen) eingesetzt, das in gleichen Verzeichnis wie das Script `ifstart.sql` erwartet wird.

Leer-Script: `null.sql`

Dieses leere Script dient in erster Lienie als weiteres Hilfscript für `ifstart.sql`, kann aber natürlich auch sonst überall eingesetzt werden, wo ein Scriptaufruf erforderlich ist, der keine Wirkung haben darf.

ifstart.sql :

Eindeutige Namen für temporäre Dateien:

```
column "Store File Name" noprint new_value ifstart_stored_set
column "Default Param File Name" noprint new_value ifstart_def_param_file

SELECT 'ifstart-stored-set_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Store File Name",
       'ifstart-defparm_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Default Param File Name"
FROM dual;

store set &ifstart_stored_set
```

Parameter Default Werte:

```
spool &ifstart_def_param_file

define 1
define 2
define 3
define 4
define 5
define 6
define 7
define 8
define 9
define 10

spool off

define 1 = '1 = 0'

define 2 = ""

define 3 = ""
define 4 = ""
define 5 = ""
define 6 = ""
define 7 = ""
define 8 = ""
define 9 = ""
```

```
define 10 = ""

@&ifstart_def_param_file
```

Zusammenbau der Start-Kommandozeile und Aufruf des Scripts:

```
-- as default call null.sql in same directory as this script to avoid
-- searching for a script with name of the actual one and calling it
define ifstart_command_line = "@null.sql"

column "Command Line" noprint new_value ifstart_command_line

SELECT '&2 ''&3'' ''&4'' ''&5'' ''&6'' ''&7'' ''&8'' ''&9'' ''&10'' ' AS
         "Command Line"
 FROM dual
 WHERE &1
   AND ltrim('&2') IS NOT NULL
;

-- make sure the parameters are cleared before calling script
undefine 1 2 3 4 5 6 7 8 9 10

@&ifstart_stored_set

@&ifstart_command_line
```

ifelsestart.sql :

Eindeutige Namen für temporäre Dateien:

```
column "Store File Name" noprint new_value ifelsestart_stored_set
column "Default Param File Name" noprint -
                        new_value ifelsestart_def_param_file

SELECT 'ifelsestart-stored-set_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Store File Name",
       'ifelsestart-defparm_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Default Param File Name"
FROM dual;

store set &ifelsestart_stored_set
```

Parameter Default Werte:

```
spool &ifelsestart_def_param_file

define 1
define 2
define 3
define 4
```

```
define 5
define 6
define 7
define 8
define 9
define 10
define 11

spool off

define 1 = 'NULL = NULL'

define 2 = ""

define 3 = ""
define 4 = ""
define 5 = ""
define 6 = ""
define 7 = ""
define 8 = ""
define 9 = ""
define 10 = ""
define 11 = ""

@&ifelsestart_def_param_file
```

Zusammenbau der Start-Kommandozeile und Aufruf des Scripts:

```
-- as default call null.sql in same directory as this script to avoid
-- searching for a script with name of the actual one and calling it
define ifelsestart_command_line = "@null.sql"

column "Command Line" noprint new_value ifelsestart_command_line

SELECT '&2 ''&4'' ''&5'' ''&6'' ''&7'' ''&8'' ''&9'' ''&10'' ''&11'' ' AS
          "Command Line"
 FROM dual
 WHERE &1
   AND ltrim('&2') IS NOT NULL
UNION ALL
SELECT '&3 ''&4'' ''&5'' ''&6'' ''&7'' ''&8'' ''&9'' ''&10'' ''&11'' ' AS
          "Command Line"
 FROM dual
 WHERE NOT (&1)
   AND ltrim('&3') IS NOT NULL
;

-- make sure the parameters are cleared before calling script
undefine 1 2 3 4 5 6 7 8 9 10 11

@&ifelsestart_stored_set

@&ifelsestart_command_line
```

g) Abbrechen von SQL*Plus bei Fehlern: `ifstart-fatal-fin.sql` und `fatal-fin.sql`

Das Script `fatal-fin.sql` wird verwendet, wenn SQL*Plus wegen einer Fehlerbedingung beendet werden muss. Das Script `ifstart-fatal-fin.sql` dient lediglich dazu, den Aufruf nur abhängig vom Zutreffen einer Bedingung auszuführen. Die Bedingung im ersten Parameter muss TRUE sein, damit `fatal-fin.sql` aufgerufen wird. Eine leere Bedigung wird als FALSE bewertet. Eine Bedingung mit dem Wert FALSE führt wie in `ifstart.sql` zum Aufruf des Scripts `null.sql`.
Für diese beiden Scripte gelten verschärfte Anforderungen. Insbesondere müssen beide auch funktionsfähig sein, wenn Rechte zum Anlegen von Dateien nicht vorhanden sind und auch dann, wenn keine Tabellen im aktuellen Datenbankschema angelegt und mit Steuer-Information beschrieben werden dürfen. Dies ist deshalb erforderlich, weil diese Scripte auch beim Fehlschlagen der ersten Tests zu den notwendigen Einstellungen und Rechten aufrufbar sein müssen. Nur deshalb existiert hierfür auch ein spezielles Vorschalt-Script zum bedingten Aufruf, weil das Standard-Script `ifstart.sql` selbst Dateien schreibt.
Als Konsequenz verändert `ifstart-fatal-fin.sql` so wenig Einstellungen wie möglich, weil ja wegen der fehlenden Speichermöglichkeit keine Wiederherstellung der vorgefundenen Einstellungeen möglich ist. Ansonsten ist die Funktionalität analog zu `ifstart.sql` – es werden jedoch ausser der Bedingung nur 2 weitere feste Parameter für das Script `fatal-fin.sql` erwartet. Ein Default-Handling für die Parameter findet nicht statt. Das erst zur Laufzeit bekannte Select-Statement wird über eine Variable direkt in den SQL-Buffer geschrieben und ausgeführt. Da `fatal-fin.sql` selbst nur dann aufgerufen wird, wenn wirklich eine Beendigung von SQL*Plus erfolgen soll, kann dieses Script jedoch ohne Nachwirkungen alle Einstellungen nach Bedarf ändern. Der erste Parameter muss ein Integerwert im Bereich 1..126 sein – jeder andere Wert, auch Zeichenketten, werden als 126 betrachtet. Dieser Integerwert wird dann als Exit-Wert für SQL*Plus verwendet. Der zweite Parameter enthält eine beliebige Fehlermeldung, die vor dem Verlassen von SQL*Plus ausgegeben wird.
Der Aufruf dieses Scripts stellt sicher, dass ein Installationsvorgang beendet wird, wenn eine Fortsetzung nicht sinnvoll ist und/oder Probleme bereiten kann. Während der Entwicklung und bei späteren Problemanalysen ist es jedoch sehr hinderlich, wenn SQL*Plus in solch einem Falle verlassen wird, weil dann keine Prüfung der `define`-Variablen usw. mehr erfolgen kann. Deshalb gibt es für Debugging-Zwecke einen speziellen Mode, der nicht für den Einsatz beim installierenden Kunden gedacht ist, in dem zwar die Datenbank-Session abgebrochen wird, das Verlassen von SQL*Plus jedoch unterbleibt. Wegen der möglicherweise fehlenden Rechte zum Schreiben von Dateien und damit nicht möglichem Default-Handling von `define`-Variablen wird für das Schalten in diesen Mode ganz bewußt ein spezieller Wert der immer existierenden und in Installationsscripten sonst nicht benötigten `define`-Variable '_EDITOR' verwendet. Wenn hierin der Wert 'CONTINUE AFTER FATAL FIN' vorgefunden wird, unterbleibt das Verlassen von SQL*Plus und eine Meldung dazu wird ausgegeben. Erreicht wird dies dadurch, dass beim Aufruf der Klausel `whenever sqlerror &whenever_clause` in der Variablen der Wert '`CONTINUE NONE`' statt '`EXIT &1`' steht – damit führt das Raise des Fehlers ORA-00028 nur noch zum Abbruch der aktuellen Datenbank-Session.

ifstart-fatal-fin.sql :

```
define ifstart_fatal_fin_call = "null.sql"
define ifstart_fatal_fin_call_build = ""

column "Command Line" noprint new_value ifstart_fatal_fin_call
```

```
column "Command Line Build" noprint new_value -
                               ifstart_fatal_fin_call_build

SELECT 'SELECT "Command Line"' || chr(10) ||
       ' FROM' || chr(10) ||
       ' (' || chr(10) ||
       '  SELECT ''fatal-fin.sql ' || nvl('&2','126') || ' "' ||
              nvl('&3','Message Parameter is empty') || '"'' AS
                 "Command Line"' || chr(10) ||
       '   FROM dual' || chr(10) ||
       '   WHERE ' || nvl('&1','1 = 0') || chr(10) ||
       '  UNION ALL' || chr(10) ||
       '  SELECT ''null.sql'' AS "Command Line"' || chr(10) ||
       '   FROM dual' || chr(10) ||
       ' )' || chr(10) ||
       ' WHERE rownum = 1' AS "Command Line Build"
 FROM dual
;

del 1 last

append &ifstart_fatal_fin_call_build

/

del 1 last

@@&ifstart_fatal_fin_call
```

fatal-fin.sql :

Parameter 1 Handing:

```
var exit_val NUMBER

DECLARE
  exit_val INTEGER;
BEGIN
  BEGIN
    exit_val := round(to_number('&1'));
  EXCEPTION
    WHEN OTHERS THEN
      exit_val := 126;
  END;
  IF exit_val < 1 OR exit_val > 126
  THEN
    exit_val := 126;
  END IF;
  :exit_val := exit_val;
END;
/

column "Exit Value" noprint new_value 1

SELECT :exit_val AS "Exit Value"
```

```
 FROM dual
;
```

Setzen WHENEVER SQLERROR Klausel anhängig von `define`-Variable _EDITOR :

```
column "Whenever Clause" noprint new_value whenever_clause
column "PreClose Message" noprint new_value preclose_message

SELECT 'CONTINUE NONE' AS "Whenever Clause",
       'The Oracle Session will be terminated now' AS
           "PreClose Message"
 FROM dual
 WHERE upper('&_EDITOR') = 'CONTINUE AFTER FATAL FIN'
UNION ALL
SELECT 'EXIT &1' AS "Whenever Clause",
       'The Oracle Session and SQL*Plus will be terminated now' ||
           ' with exit value ' || ltrim(to_char('&1')) AS
           "PreClose Message"
 FROM dual
 WHERE upper('&_EDITOR') != 'CONTINUE AFTER FATAL FIN'
;

whenever sqlerror &whenever_clause
```

Meldung ausgeben und Session beenden, im Normalfall auch SQL*Plus :

```
prompt
prompt
prompt ============================================================
prompt &2
prompt ============================================================
prompt
prompt ############################################################
prompt ###  &preclose_message
prompt ############################################################
prompt

DECLARE
  session_kill EXCEPTION;
  PRAGMA EXCEPTION_INIT(session_kill,-28);
BEGIN
  RAISE session_kill;
END;
/

-- never reach this position when exiting on sqlerror

prompt
prompt ############################################################
prompt ###  Leaving SQL*Plus is prohibited  !!!                  ###
prompt ############################################################
prompt #
prompt # Don't use this Session for new connection and further work
prompt # other than checking variables for script debugging reason
```

```
prompt #
prompt ############################################################
prompt
```

h) Aufrufe von Objectscripten mit Ausgabe-Steuerung: `DB-install.sql` mit Hilfsscript eval-verbose_output_level.sql

Bei jeder Installation in eine Datenbank werden üblicherweise eine größere Anzahl von Srcipten aufgerufen, die die jeweiligen Objecte in der Datenbank anlegen oder ändern. Als Entwickler oder Installationstester hat man da gerne Information über Erfolg oder Misserfolg, wie sie durch die `show error` Klausel nach dem DDL-Statement ausgegeben werden. Andererseits werden dadurch auch viele Fehlermeldungen angezeigt, die nur durch aktuell nicht auflösbare Abhängigkeiten verursacht durch eine nicht optimale Aufruf-Reihenfolge für die Datenbank-Objecte oder gar geschlossene Abhängigkeitsketten der Db-Objecte. Solche unnötigen Fehlermeldungen sollten während einer Installation bei einem Kunden unbedingt vermieden werden.
Beide Forderungen sind vereinbar, wenn die `show error` Klausel grundsätzlich in den Objectscripten vorhanden ist, deren Aufruf aber über ein zusätzliches Script mit Ausgabesteuerung erfolgt. `DB-install.sql` mit genau einem Parameter für den Objectscript-Namen verwendet dazu die 3 `define`-Variablen verbose_1_output, verbose_2_output und verbose_3_output, die jewiels die Werte ON oder OFF aufweisen müssen. verbose_1_output bestimmt den `termout`-Wert während des Ausgebens der Installationsmeldung, verbose_2_output den `termout`-Wert während der Ausführung des Objectscripts, und verbose_3_output den `echo`-Wert während der Ausführung des Objectscripts.
Das Hilfscript `eval-verbose_output_level.sql` dienst lediglich dazu, vor dem Aufruf irgendwelcher Objectscripte ausgehend von der `define`-Variablen verbose_output__level und abhängig von deren Wert die beschriebenen `define`-Variablen mit den ON-OFF-Werten zu setzen – die Variable verbose_output__level wird dabei gelöscht, sodass bei der nächsten Installation wieder das Defaultverhalten gilt.

DB-install.sql :

```
set termout &verbose_1_output echo off

prompt installing &&1

set termout &verbose_2_output echo &verbose_3_output

@&1

set termout off echo off
```

eval-verbose_output_level.sql :

Eindeutige Namen für temporäre Dateien:

```
define verbose_output__stored_set = -
```

```
                "verbose_output_stored-set_abcdefghijklmn.sql"
define verbose_output__check = -
                'verbose_output_level_check_abcdefghijklmn.sql'

column "Verbose Output Store File Name" noprint -
                new_value verbose_output__stored_set
column "Verbose Output Check File Name" noprint -
                new_value verbose_output__check
column "Verbose Output Level" noprint new_value verbose_output__level
column "Verbose Output 1 Level" noprint new_value verbose_1_output
column "Verbose Output 2 Level" noprint new_value verbose_2_output
column "Verbose Output 3 Level" noprint new_value verbose_3_output

SELECT 'verbose_output_stored-set_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Verbose Output Store File Name",
       'verbose_output_level_check_' ||
       to_char(systimestamp,'YYYYMMDDHH24MISSFF') || '.sql' AS
         "Verbose Output Check File Name"
FROM dual;

store set &verbose_output__stored_set
```

Variablen Default Wert:

```
spool &verbose_output__check

define verbose_output__level

spool off

define verbose_output__level = 0

@&verbose_output__check
```

Umsetzung unerlaubte Variablenwerte auf Defaultwerte:

```
var testvar NUMBER

DECLARE
  n NUMBER;
  not_a_number EXCEPTION;
  PRAGMA EXCEPTION_INIT(not_a_number,-1722);
BEGIN
  BEGIN
    n := to_number('&verbose_output__level');
  EXCEPTION
    WHEN not_a_number THEN
      n := 0;
  END;
  IF n < 0 THEN n := 0; END IF;
  IF n > 3 THEN n := 3; END IF;
  n := round(n);
  :testvar := n;
```

```
END;
/
```

Setzen der Variablen mit ON-OFF-Werten abhängig vom Wert der Hauptvariablen:

```
SELECT CASE WHEN :testvar >= 1 THEN 'ON' ELSE 'OFF' END AS
         "Verbose Output 1 Level",
       CASE WHEN :testvar >= 2 THEN 'ON' ELSE 'OFF' END AS
         "Verbose Output 2 Level",
       CASE WHEN :testvar >= 3 THEN 'ON' ELSE 'OFF' END AS
         "Verbose Output 3 Level"
FROM dual;

-- always undefine verbose_output_level
-- because it may didn't exist before calling this script
undefine verbose_output__level
```

Zusammenfassung

Es fällt auf, dass die Übertragung von Werten auf die SQL*Plus-Ebene einen beträchtlichen Code-Anteil in den Scripten ausmacht. Dies gilt auch für die in den Beispiel-Scripten nicht dargestellten Code-Fragmente zum Sichern, Modifzieren und Wiederherstellen von SQL*Plus-Einstellungen sowie das Löschen von im Script erzeugten `define`-Variablen und `column`-Definitionen.
Gerade diese Eigenschaft läßt den Einsatz solcher Scripte um so dringlicher erscheinen, weil gerade die Codierung zur Wiederherstellung eines bestimmten Zustands wie vor dem Scriptaufruf besonders fehlerträchtig ist. Solche Fehler fallen nicht im vorliegenden Script bei der aktuellen Ausführung auf, sondern, falls sie sich auswirken, im späteren Ablauf der Installation, was das Debugging um so schwieriger macht. Derartige Hilfsutilities bieten also nicht nur Komfort für den Entwickler, sondern führen auch zu einer verbesserten Qualität und Wartbarkeit der Installationsscripte.

Kontaktadresse:

Dr. Kurt Franke
Cellent Finance Solutions AG
Calwer Straße 33
D-70173 Stuttgart

Telefon: +49 (0) 711-222992-676
Fax: +49 (0) 711-222992-899
E-Mail Kurt.Franke@cellent-fs.de
Internet: www.cellent-fs.de