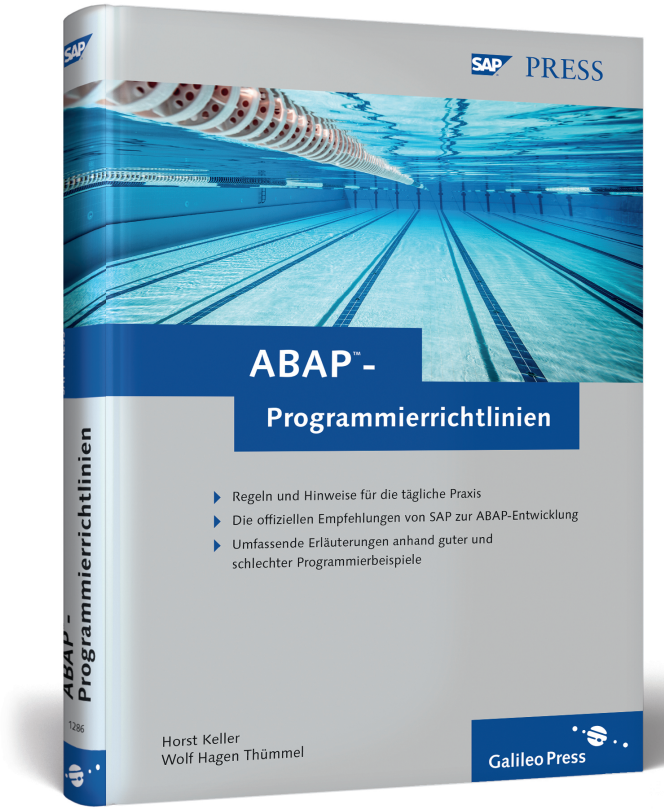
Horst Keller, Wolf Hagen Thümmel

ABAP™-Programmierrichtlinien



Galileo Press

Bonn • Boston

Auf einen Blick

Inhalt

»Ach, wenn Du erfahren wolltest, wie ich Dich liebe, so müßtest Du mir eine neue Sprache schenken.«
– Friedrich Schiller

3 ABAP-spezifische Grundregeln

Neben den in Kapitel 2, »Allgemeine Grundregeln«, aufgeführten Regeln führen wir in diesem Kapitel zusätzlich einen Satz ABAP-spezifischer Grundregeln ein, die sich aus speziellen technischen Gegebenheiten der Sprache ABAP, der ABAP-Laufzeitumgebung und ihrer Historie ergeben. Auch diese Grundregeln bestimmen viele der auf dieses Kapitel folgenden spezielleren Regeln.

3.1 ABAP Objects als Programmiermodell

Hintergrund

ABAP ist eine hybride Programmiersprache, die sowohl ein prozedurales als auch ein objektorientiertes Programmiermodell unterstützt. Das prozedurale Programmiermodell beruht auf der Modularisierung von Programmen in die klassischen Verarbeitungsblöcke, das heißt Ereignisblöcke, Dialogmodule, Funktionsbausteine und Unterprogramme. In ABAP Objects tritt die Klasse konzeptionell an die Stelle des klassischen Programms,[1] und die Modularisierung erfolgt durch deren Methoden.

Beide Modelle sind in der Weise interoperabel, dass in klassischen Verarbeitungsblöcken auf Klassen zugegriffen werden kann und innerhalb von Methoden wiederum klassische Programme und Prozeduren aufgerufen werden können. Der hybride Charakter der Sprache ist in erster Linie der Abwärtskompatibilität geschuldet, da ABAP prozedurale Wurzeln hat und sowohl ganze Programme als auch wiederverwendbare Prozeduren (in erster Linie Funktionsbausteine) mit Einführung des objektorientierten Programmiermodells Ende der 1990er-Jahre weiterhin nutzbar bleiben sollten.

1 Technisch gesehen sind Klassen nach wie vor in Programmen deklariert und implementiert.

Regel

Regel 3.1: ABAP Objects verwenden

Verwenden Sie bei der Neu- und Weiterentwicklung so weit wie möglich ABAP Objects. Klassische Verarbeitungsblöcke dürfen nur noch in Ausnahmefällen neu angelegt werden.

Details

Die Forderung nach der Trennung der Belange (siehe Regel 2.1) wird am besten durch eine weitestgehende Verwendung von ABAP Objects unterstützt. Eine detaillierte Gegenüberstellung von ABAP Objects und dem prozeduralen Programmiermodell ist nicht Gegenstand dieses Buches. Dass die objektorientierte Programmierung – und hier insbesondere ABAP Objects im Vergleich zum klassischen prozeduralen ABAP – besser geeignet ist, die Anforderungen zeitgemäßer Programmierung zu erfüllen, wurde unter anderem in einem Beitrag im SAP Professional Journal (*Not Yet Using ABAP Objects? – Eight Reasons why every ABAP Developer Should Give It a Second Look*) dargelegt. Die dort genannten acht Gründe, ABAP Objects zu verwenden, lauten zusammengefasst wie folgt:

1. **Datenkapselung**
   ABAP Objects ermöglicht eine fortgeschrittene Art der Datenkapselung. Bei der klassischen prozeduralen Programmierung wird der Zustand einer Anwendung durch den Inhalt von globalen Variablen bestimmt. In der objektorientierten Programmierung ist der Zustand in Klassen oder Objekten als Instanzen von Klassen gekapselt. Die Aufteilung der Daten in die verschiedenen Sichtbarkeitsbereiche einer Klasse – öffentlich, geschützt, paketsichtbar (ab Release 7.2) und privat – sorgt für eine klare Unterscheidung zwischen extern und intern verwendbaren Daten. Selbst ohne eine tief gehende objektorientierte Modellierung profitieren Anwendungsprogramme hinsichtlich Robustheit und Wartbarkeit von diesen Eigenschaften.
2. **Explizite Instanzierung**
   ABAP Objects ermöglicht die mehrfache Instanzierung einer Klasse über explizite Objekterzeugung mittels der Anweisung `CREATE OBJECT`. Jede Instanz einer Klasse (Objekt) hat einen eigenen Zustand, der durch die Werte ihrer Attribute festgelegt wird und über die Methoden der Klassen geändert werden kann. Eine automatische Garbage Collection sorgt dafür, dass Objekte, die nicht mehr benötigt werden, aus dem Speicher gelöscht werden. Im pro-

zeduralen Modell gibt es keine Mehrfachinstanzierung, weshalb dort mit zustandslosen Funktionen auf getrennt abgelegten Daten gearbeitet werden muss.

3. **Vererbung**
   ABAP Objects ermöglicht die Wiederverwendung von Klassen durch Vererbung, wobei Klassen mit speziellen Verhaltensweisen von allgemeineren Klassen abgeleitet werden und nur die Unterschiede neu implementiert werden müssen. Im prozeduralen Modell können vorhandene Funktionen nur genauso verwendet werden, wie sie sind, oder es müssen neue angelegt werden.

4. **Interfaces**
   In ABAP Objects können Objekte über eigenständige Interfaces angesprochen werden. Dies befreit Entwickler davon, sich um Implementierungsdetails der hinter dem Interface liegenden Klasse kümmern zu müssen. Dadurch kann der Anbieter eines Interface die dahinterliegenden Implementierungen ändern, ohne dass die Programme, die das Interface verwenden, modifiziert werden müssen. Im prozeduralen Modell gibt es kein solches Konzept eigenständiger Interfaces.

5. **Ereignisse**
   ABAP Objects erleichtert die Implementierung ereignisgetriebener Programmabläufe. Anwendungen können über einen Publish-and-Subscribe-Mechanismus lose gekoppelt werden, wobei der Auslöser eines Ereignisses nichts über eventuelle Behandler wissen muss. Dies erlaubt größere Flexibilität im Vergleich zum prozeduralen Ansatz, bei dem Programme stärker gekoppelt sind und der Programmablauf in der Regel viel starrer vorgegeben ist.

6. **Explizite orthogonale Konzepte**
   In ABAP Objects gibt es eine kleine Anzahl genau definierter fundamentaler und zueinander orthogonaler Konzepte, die es zuverlässiger und weniger fehleranfällig als das klassische ABAP machen. Im klassischen prozeduralen ABAP dominieren implizite Verhaltensweisen, in denen Programme durch implizite Ereignisse der Laufzeitumgebung und über globale Daten gesteuert werden. Die Konzepte von ABAP Objects werden dagegen in einem Programm explizit wiedergegeben. ABAP Objects ist damit im Vergleich zum klassischen prozeduralen ABAP einfacher erlern- und anwendbar.

7. **Bereinigte Syntax**
   In ABAP Objects gelten bereinigte Syntax- und Semantikregeln. Das klassische prozedurale ABAP ist eine evolutionär gewachsene Sprache mit vielen obsoleten und sich überschneidenden Konzepten. Mit Einführung von ABAP Objects bot sich mit Klassen und Methoden ein Feld für bereinigte

Syntax- und Semantikregeln, das von Anforderungen an die Abwärtskompatibilität völlig unbelastet war. Auf diese Weise konnten in ABAP Objects, das heißt innerhalb von Klassen und Methoden, die meisten obsoleten und fehleranfälligen Sprachkonstrukte syntaktisch verboten werden. Zusätzlich werden fragwürdige und potenziell fehlerhafte Zugriffe auf Daten schärfer überprüft und gegebenenfalls ebenso verboten. Die Syntaxbereinigung erzwingt in Klassen eine Verwendung der Sprache ABAP, wie sie außerhalb von Klassen nur durch Richtlinien gefordert werden kann (siehe Abschnitt 3.3, »Modernes ABAP«).

8. **Zugang zu neuen Technologien**
   ABAP Objects ist oft der einzige Weg, um mit neuen ABAP-Technologien umzugehen. Beispielsweise bieten GUI Controls, Web Dynpro ABAP, Run Time Type Services (RTTS) oder das Internet Connection Framework (ICF) ausschließlich klassenbasierte Schnittstellen an. Wenn Programme, die solche Services verwenden, weiterhin rein prozedural implementiert würden, käme es zu einer unnötigen Vermischung der Programmiermodelle mit entsprechender Erhöhung der Komplexität.

Die dringende Empfehlung zur Verwendung von ABAP Objects hat somit sowohl inhaltliche als auch formale Aspekte:

- Wie in den Punkten 1 bis 5 aufgeführt, ist das objektorientierte Programmiermodell inhaltlich besser geeignet, die Komplexität von Software durch Prinzipien wie Kapselung und Vererbung beherrschbar zu halten. Zugegebenermaßen ist gutes objektorientiertes Design keine leichte Aufgabe, und auch heute noch gibt es Entwickler mit wenig Erfahrung auf diesem Gebiet. Wer vor diesem Hintergrund immer noch mit dem Gedanken spielt, eine Neuentwicklung in klassischer prozeduraler Manier anzugehen, muss sich jedoch vergegenwärtigen, dass auch das prozedurale ereignisgesteuerte ABAP-Programmiermodell mit seinen Systemereignissen nicht leicht zu durchschauen ist.
- Die Punkte 6 bis 8 beschreiben eher formale Aspekte. Die dort aufgeführten Gründe sprechen dafür, Prozeduren heute nur noch in Form von Methoden anzulegen, selbst in Abwesenheit eines echten objektorientierten Designs. Funktionsbausteine und Unterprogramme sollen nur noch in den Ausnahmefällen angelegt werden, in denen ABAP Objects bisher keine Alternative bietet.

Hinweise und Empfehlungen zum erfolgreichen Einsatz von ABAP Objects liefert Abschnitt 5.1, »Objektorientierte Programmierung«.

Ausnahme

Im derzeitigen Zustand (Releases 7.0 EhP2 und 7.2) fehlen in ABAP Objects noch folgende Eigenschaften, um klassische Verarbeitungsblöcke vollständig durch Methoden zu ersetzen:

- Remote Method Invocation (RMI) als Ersatz für den Remote Function Call (RFC)
- ein Ersatz für den Aufruf von Verbuchungsfunktionsbausteinen (`CALL FUNCTION IN UPDATE TASK`)
- ein Ersatz für den Aufruf von Unterprogrammen bei `COMMIT WORK` und `ROLLBACK WORK` (`PERFORM ON COMMIT/ROLLBACK`)
- objektorientierte Behandlung von klassischen Dynpros inklusive Selektionsbildern als Ersatz für Dialogtransaktionen, `CALL SCREEN` und `CALL SELECTION-SCREEN`
- dynamische Erzeugung von Klassen als Ersatz für die klassische dynamische Programmerzeugung (`GENERATE SUBROUTINE POOL`)
- direkte Unterstützung der Hintergrundverarbeitung als Ersatz für den Aufruf ausführbarer Programme (`SUBMIT VIA JOB`)

Genau für diese Fälle dürfen in neuen Programmen noch folgende klassische Verarbeitungsblöcke angelegt werden:

- Funktionsbausteine werden noch für RFC und die Verbuchung benötigt und für den Aufruf von klassischen Dynpros und Selektionsbildern empfohlen (siehe Regel 5.19).
- Unterprogramme werden noch für `PERFORM ON COMMIT/ROLLBACK` und in dynamisch generierten Subroutinen-Pools (`GENERATE SUBROUTINE POOL`) benötigt.
- Dialogmodule und Ereignisblöcke für Selektionsbildereignisse werden noch in Funktionsgruppen benötigt, die klassische Dynpros und Selektionsbilder verschalen (siehe Regel 3.2).
- Der Ereignisblock `START-OF-SELECTION` wird noch in ausführbaren Programmen benötigt, die für die Hintergrundverarbeitung vorgesehen sind.

Innerhalb eines solchen Verarbeitungsblocks soll die Ausführung dann jedoch sofort an eine geeignete Methode delegiert werden (siehe Regel 6.37, »Keine Implementierungen in Funktionsbausteinen und Unterprogrammen« und Regel 6.44, »Keine Implementierungen in Dialogmodulen und Ereignisblöcken«). Diese muss keine Methode einer globalen Klasse sein, sondern kann durchaus im Rahmen einer lokalen Klasse innerhalb des zugehörigen Rahmen-

programms angesiedelt sein. Damit auch in solchen Verarbeitungsblöcken die gleiche strengere Prüfung wie in Methoden durchgeführt wird, kann in der erweiterten Programmprüfung (siehe Abschnitt 3.4.2, »Erweiterte Programmprüfung«) die Prüfung Veraltete Anweisungen (OO-Kontext) eingeschaltet werden.

Schlechtes Beispiel

Listing 3.1 enthält eine ansatzweise Implementierung der Behandlung von verschiedenen Arten von Bankkonten in einer Funktionsgruppe und deren Verwendung in einem Programm, wobei nur die Funktion »Abheben eines Betrags« gezeigt wird. Die Funktionsbausteine der Funktionsgruppe arbeiten auf externen Daten, die hier beim Ereignis `LOAD-OF-PROGRAM` in eine globale interne Tabelle geladen werden. Die Steuerung, ob mit einem Giro- oder Sparkonto umgegangen wird, erfolgt über einen Eingabeparameter, und die unterschiedliche Behandlung wird über eine `CASE-WHEN`-Kontrollstruktur an unterschiedliche Unterprogramme delegiert, wobei keine Wiederverwendung stattfindet. Die Unterprogramme greifen auf die globale interne Tabelle zu. In einem Anwendungsprogramm wird der Funktionsbaustein zum Abheben für verschiedene Konten aufgerufen. Die Ausnahmebehandlung erfolgt klassisch mit weiteren `CASE-WHEN`-Kontrollstrukturen für die Abfrage von `sy-subrc`.

```
FUNCTION-POOL account.

DATA account_tab TYPE SORTED TABLE OF accounts
                 WITH UNIQUE KEY id.

LOAD-OF-PROGRAM.
  "fetch amount for all accounts into account_tab
  ...

...

FUNCTION withdraw.
*"----------------------------------------------------
*"  IMPORTING
*"     REFERENCE(id) TYPE accounts-id
*"     REFERENCE(kind) TYPE c DEFAULT 'C'
*"     REFERENCE(amount) TYPE accounts-amount
*"  EXCEPTIONS
*"      negative_amount
*"      unknown_account_type
*"----------------------------------------------------
  CASE kind.
    WHEN 'C'.
      PERFORM withdraw_from_checking_account
```

```
        USING id amount.
    WHEN 'S'.
      PERFORM withdraw_from_savings_account
        USING id amount.
    WHEN OTHERS.
      RAISE unknown_account_type.
  ENDCASE.
ENDFUNCTION.

FORM withdraw_from_checking_account
  USING l_id     TYPE accounts-id
        l_amount TYPE accounts-amount.
  FIELD-SYMBOLS <account> TYPE accounts.
  READ TABLE account_tab ASSIGNING <account>
    WITH TABLE KEY id = l_id.
  <account> = <account> - l_amount.
  IF <account> < 0.
    "Handle debit balance
    ...
  ENDIF.
ENDFORM.

FORM withdraw_from_savings_account
   USING l_id     TYPE accounts-id
         l_amount TYPE accounts-amount.
  FIELD-SYMBOLS <account> TYPE accounts.
  READ TABLE account_tab ASSIGNING <account>
    WITH TABLE KEY id = l_id.
  IF <account>_wa-amount > l_amount.
    <account>-amount = <account>-amount - l_amount.
  ELSE.
    RAISE negative_amount.
  ENDIF.
ENDFORM.

*********************************************************

PROGRAM bank_application.

...

CALL FUNCTION 'WITHDRAW'
  EXPORTING
    id                  = ...
    kind                = 'C'
    amount              = ...
  EXCEPTIONS
    unknown_account_type = 2
    negative_amount      = 4.
```

```
CASE sy-subrc.
  WHEN 2.
    ...
  WHEN 4.
    ...
ENDCASE.

...

CALL FUNCTION 'WITHDRAW'
  EXPORTING
    id                  = ...
    kind                = 'S'
    amount              = ...
  EXCEPTIONS
    unknown_account_type = 2
    negative_amount      = 4.
CASE sy-subrc.
  WHEN 2.
    ...
  WHEN 4.
    ...
ENDCASE.
```

Listing 3.1 Modellierung von Bankkonten in Funktionsgruppe

Gutes Beispiel

Listing 3.2 enthält eine ansatzweise Implementierung der Behandlung von verschiedenen Arten von Bankkonten in Klassen und deren Verwendung in einer Klasse, wobei wieder nur die Funktion »Abheben eines Betrags« gezeigt wird.

Die verschiedenen Kontoarten werden in Unterklassen einer abstrakten Klasse für Konten implementiert. Jede Instanz eines Kontos wird in ihrem Konstruktor genau mit den Daten versorgt, die sie benötigt. Die Anwendungsklasse erzeugt je nach Bedarf Instanzen von Konten der gewünschten Art und verwendet deren Methoden polymorph über eine Oberklassenreferenzvariable. Die Ausnahmebehandlung erfolgt über klassenbasierte Ausnahmen. Es werden keine `CASE-WHEN`-Kontrollstrukturen benötigt. Wie bereits in der Beschreibung der Beispiele von Abschnitt 2.1, »Trennung der Belange«, angekündigt, entsteht hier bei der Verwendung von Klassen kein Overhead an Code mehr gegenüber der prozeduralen Programmierung.

```
CLASS cx_negative_amount DEFINITION PUBLIC
                         INHERITING FROM cx_static_check.
ENDCLASS.
```

```
CLASS cl_account DEFINITION ABSTRACT PUBLIC.
  PUBLIC SECTION.
    METHODS: constructor IMPORTING id     TYPE string,
             withdraw    IMPORTING amount TYPE i
                         RAISING   cx_negative_amount.
  PROTECTED SECTION.
    DATA amount TYPE accounts-amount.
ENDCLASS.

CLASS cl_account IMPLEMENTATION.
  METHOD constructor.
    "fetch amount for one account into attribute amount
    ...
  ENDMETHOD.
  METHOD withdraw.
    me->amount = me->amount - amount.
  ENDMETHOD.
ENDCLASS.

CLASS cl_checking_account DEFINITION PUBLIC
                          INHERITING FROM cl_account.
  PUBLIC SECTION.
    METHODS withdraw REDEFINITION.
ENDCLASS.

CLASS cl_checking_account IMPLEMENTATION.
  METHOD withdraw.
    super->withdraw( amount ).
    IF me->amount < 0.
      "Handle debit balance
      ...
    ENDIF.
  ENDMETHOD.
ENDCLASS.

CLASS cl_savings_account DEFINITION PUBLIC
                         INHERITING FROM cl_account.
  PUBLIC SECTION.
    METHODS withdraw REDEFINITION.
ENDCLASS.

CLASS cl_savings_account IMPLEMENTATION.
  METHOD withdraw.
    IF me->amount > amount.
      super->withdraw( amount ).
    ELSE.
      RAISE EXCEPTION TYPE cx_negative_amount.
    ENDIF.
```

```
  ENDMETHOD.
ENDCLASS.

*******************************************************

CLASS bank_application DEFINITION PUBLIC.
  PUBLIC SECTION.
    CLASS-METHODS main.
ENDCLASS.

CLASS bank_application IMPLEMENTATION.
  METHOD main.
    DATA: account1 TYPE REF TO cl_account,
          account2 TYPE REF TO cl_account.

  ...

    CREATE OBJECT account1 TYPE cl_checking_account
      EXPORTING
        id = ...

    CREATE OBJECT account2 TYPE cl_savings_account
      EXPORTING
        id = ...

  ...

    TRY.
        account1->withdraw( ... ).
        account2->withdraw( ... ).
      CATCH cx_negative_amount.
        ...
    ENDTRY.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.2 Modellierung von Bankkonten in Klassen

3.2 Programmtyp und Programmeigenschaften

Bereits beim Anlegen eines ABAP-Programms erfolgt über die Wahl des Programmtyps und der Programmattribute eine Weichenstellung bezüglich der späteren Robustheit und Wartbarkeit. Programmtyp und Programmattribute bestimmen unter anderem die Prüfschärfe der Syntaxprüfung. Eine weitere wichtige Eigenschaft von Programmen (wie auch von allen anderen Entwicklungsobjekten) ist deren Originalsprache.

3.2.1 Programmtyp

Hintergrund

Jedes ABAP-Programm hat einen Programmtyp, der festlegt, welche Deklarationen und Verarbeitungsblöcke ein Programm enthalten und wie es über die ABAP-Laufzeitumgebung ausgeführt werden kann. Die möglichen Programmtypen in ABAP sind:

- **Ausführbares Programm**
  Ein ausführbares Programm kann alle möglichen deklarativen Anweisungen enthalten. Alle Verarbeitungsblöcke außer Funktionsbausteinen sind möglich. Es unterstützt klassische Dynpros sowie Selektionsbilder und kann sowohl über die Anweisung `SUBMIT` als auch über Transaktionscodes ausgeführt werden. Ein ausführbares Programm wird mit dem ABAP Editor angelegt.
- **Class-Pool**
  Ein Class-Pool enthält stets deklarative Anweisungen für eine globale Klasse und kann daneben auch deklarative Anweisungen für lokale Typen, Interfaces und Klassen beinhalten. Als Verarbeitungsblöcke sind nur Methoden möglich. Er unterstützt keine klassischen Dynpros oder Selektionsbilder. Die Methoden der globalen Klasse können je nach Sichtbarkeit von außen aufgerufen und die öffentlichen Methoden der globalen Klasse auch über Transaktionscodes ausgeführt werden. Ein Class-Pool wird mit dem Class Builder angelegt.
- **Interface-Pool**
  Ein Interface-Pool kann nur die deklarativen Anweisungen für ein globales Interface enthalten. Es sind keine Verarbeitungsblöcke und keine klassischen Dynpros oder Selektionsbilder möglich. Ein Interface-Pool ist nicht aufruf- oder ausführbar und wird mit dem Class Builder angelegt.
- **Funktionsgruppe (Function-Pool)**
  Eine Funktionsgruppe kann alle Arten von deklarativen Anweisungen enthalten. Alle Verarbeitungsblöcke außer Reporting-Ereignisblöcken werden unterstützt. Sie unterstützt klassische Dynpros sowie Selektionsbilder. Ihre Funktionsbausteine können aufgerufen werden, es ist aber auch über Transaktionscodes ein Einstieg in die Dynpro-Verarbeitung der Funktionsgruppe möglich. Eine Funktionsgruppe wird mit dem Function Builder angelegt.
- **Modul-Pool**
  Ein Modul-Pool kann alle möglichen deklarativen Anweisungen enthalten. Alle Verarbeitungsblöcke außer Reporting-Ereignisblöcken und Funktionsbausteinen werden unterstützt. Er unterstützt klassische Dynpros sowie

Selektionsbilder und kann über Transaktionscodes ausgeführt werden. Ein Modul-Pool wird mit dem ABAP Editor angelegt.

- **Subroutinen-Pool**
  Ein Subroutinen-Pool kann alle möglichen deklarativen Anweisungen enthalten. Als Verarbeitungsblöcke sind der Ereignisblock `LOAD-OF-PROGRAM` sowie Unterprogramme und Methoden möglich. Er unterstützt keine klassischen Dynpros oder Selektionsbilder. Die Unterprogramme können aufgerufen werden, es ist aber auch eine Ausführung von Methoden über Transaktionscodes möglich. Ein Subroutinen-Pool wird mit dem ABAP Editor angelegt.
- **Typgruppe (Type-Pool)**
  Eine Typgruppe kann die deklarativen Anweisungen `TYPES` und `CONSTANTS` enthalten. Es sind keine Verarbeitungsblöcke und keine klassischen Dynpros oder Selektionsbilder möglich. Eine Typgruppe ist nicht aufruf- oder ausführbar. Eine Typgruppe wird mithilfe des ABAP Dictionarys angelegt.

Neben den genannten Kompilationseinheiten, das heißt Programmen, die eigenständig kompilierbar sind, gibt es auch *Include-Programme*, auf die wir in Abschnitt 4.5, »Quelltextorganisation«, gesondert eingehen.

Eine Programmausführung in ABAP bedeutet, dass ein Programm in den Speicher geladen wird und einer oder mehrere seiner Verarbeitungsblöcke ausgeführt werden. Man kann hier die eigenständige und die gerufene Programmausführung unterscheiden:

- **Eigenständige Programmausführung**
  Bei der eigenständigen Programmausführung wird das Programm entweder über einen Transaktionscode (Anweisungen `CALL TRANSACTION` und `LEAVE TO TRANSACTION`) oder bei einem ausführbaren Programm über die Anweisung `SUBMIT` gestartet. Die Anweisung `SUBMIT` gestattet auch die Ausführung in einem Hintergrundprozess.
- **Gerufene Programmausführung**
  Bei der gerufenen Programmausführung ruft ein laufendes Programm eine Prozedur (Methode, Funktionsbaustein oder Unterprogramm) eines anderen Programms auf, das bei Bedarf in den internen Modus des Aufrufers geladen wird (siehe Abschnitt 6.5.6).

Der Programmablauf im Rahmen der eigenständigen Programmausführung ist abhängig vom gewählten Programmtyp und der Art des Programmaufrufs:

- Beim Programmaufruf über eine Transaktion muss zwischen *objektorientierten* (*OO-Transaktion*) und *Dialogtransaktionen* unterschieden werden. Bei objektorientierten Transaktionen ist der Transaktionscode mit einer

Methode einer lokalen oder globalen Klasse verbunden. Der Programmablauf wird durch diese Methode bestimmt. Dialogtransaktionen sind hingegen mit einem klassischen Dynpro des Programms verknüpft. Der Programmablauf wird hier durch die zugehörige Dynpro-Ablauflogik bestimmt.

- Der Programmablauf eines über `SUBMIT` gestarteten *ausführbaren Programms* wird durch den Reporting-Prozess der ABAP-Laufzeitumgebung bestimmt. Hierbei werden die verschiedenen Reporting-Ereignisblöcke `START-OF-SELECTION`, `GET` und `END-OF-SELECTION` des Programms von der Laufzeitumgebung aufgerufen.

Der Programmtyp muss unter Beachtung der hier aufgezählten technischen Eigenschaften eines Programms und der Anforderungen an die Programmausführung geeignet gewählt werden. Nicht mehr alle der genannten Programmtypen lassen sich sinnvoll für Neuentwicklungen einsetzen.

Regel

Regel 3.2: Geeigneten Programmtyp wählen

Wählen Sie den Programmtyp wie folgt:

- Für globale Klassen und Interfaces ergibt sich automatisch der Programmtyp Class-Pool bzw. Interface-Pool.
- Für die Implementierung abgeschlossener Funktionalität, die nicht in der Klassenbibliothek erscheinen soll, kann der Programmtyp Subroutinen-Pool für lokale Klassen verwendet werden.[2]
- Bei Bedarf für Funktionsbausteine ergibt sich automatisch der Programmtyp Funktionsgruppe. Außerdem sind Funktionsgruppen zur Verschalung klassischer Dynpros oder von Selektionsbildern zu verwenden.
- Bei Bedarf für eine Ausführung im Rahmen der Hintergrundverarbeitung ergibt sich automatisch der Programmtyp ausführbares Programm.
- Es sollen keine neuen Modul-Pools und Typgruppen mehr angelegt werden.

Details

Die in Regel 3.2 aufgeführte Hierarchie zur Wahl des Programmtyps ergibt sich aus der grundlegenden Regel in Abschnitt 3.1, die die Verwendung von ABAP Objects vorschreibt. Die folgende Liste führt die Teilaspekte noch weiter aus:

2 Ab Release 7.2 kann die Verwendbarkeit globaler Klassen durch das operationale Paketkonzept auf ein Paket eingeschränkt werden, sodass diese Rolle von Subroutinen-Pools an Bedeutung verliert.

- Soll im Rahmen von ABAP Objects Funktionalität paket- oder systemweit zur Verfügung gestellt werden, erfolgt dies über globale Klassen oder Interfaces, die implizit den Programmtyp Class-Pool oder Interface-Pool haben. Der Aufruf erfolgt entweder über einen Methodenaufruf oder über eine OO-Transaktion, wenn eine eigenständige Programmausführung gewünscht ist.
- Zur Implementierung abgeschlossener Funktionalität, die nicht über einen Methodenaufruf, sondern über einen Transaktionscode aufgerufen werden soll und die darüber hinaus weder eine Parameterübergabe benötigt noch eine Benutzeroberfläche aufweist, kann der Programmtyp Subroutinen-Pool verwendet werden. Die Implementierung soll ausschließlich über lokale Klassen und der Programmaufruf über eine OO-Transaktion erfolgen. Subroutinen-Pools waren, wie die Bezeichnung nahelegt, ursprünglich einmal für Unterprogramme vorgesehen, die aus anderen Programmen aufgerufen werden. Da Unterprogramme und insbesondere deren externer Aufruf im Rahmen der vorliegenden Programmierrichtlinien für obsolet erklärt werden, entfällt dieser Verwendungszweck für Subroutinen-Pools. Stattdessen werden Subroutinen-Pools hier als unabhängige Container für lokale Klassen vorgeschlagen, da sie ansonsten kaum von impliziten Prozessen der ABAP-Laufzeitumgebung beeinflusst werden.
- Remotefähige Funktionsbausteine (Remote-enabled Function Module, RFM), die Funktionalität über die RFC-Schnittstelle entweder server- oder systemübergreifend zur Verfügung stellen oder der Parallelisierung dienen, können nur in einer Funktionsgruppe angelegt werden. Die Implementierung der eigentlichen Funktionalität soll aber in einer Klasse erfolgen, beispielsweise in einer lokalen Klasse innerhalb der Funktionsgruppe (siehe Regel 6.37).
- Für Verbuchungsfunktionsbausteine, die im Rahmen der Verbuchung mit `CALL FUNCTION IN UPDATE TASK` aufgerufen werden, gilt das Gleiche wie für remotefähige Funktionsbausteine.
- Programme mit einer klassischen Dynpro-Oberfläche oder Selektionsbildern (soweit diese noch erforderlich sein sollten; siehe Regel 5.18, »Web Dynpro ABAP verwenden«) sollen ebenfalls in Form einer Funktionsgruppe angelegt werden, die lediglich die Benutzeroberfläche implementiert, jedoch keine eigene Anwendungslogik enthält (siehe Regel 2.1, »SoC-Prinzip befolgen« und Regel 5.19, »Klassische Dynpros und Selektionsbilder kapseln«). Dieser Programmtyp ist deshalb geeignet, weil er sowohl klassische Dynpros als auch eine externe funktionale Schnittstelle in Form von Funktionsbausteinen enthalten kann. Die von der Dynpro-Ablauflogik aufgerufe-

nen Dialogmodule der Funktionsgruppe sollten im Wesentlichen nur Methodenaufrufe enthalten, beispielsweise für Methoden lokaler Klassen.

- Ein ausführbares Programm besteht aus einer Reihe von Ereignisblöcken, die beim Eintreten der verschiedenen Reporting-Ereignisse ausgeführt werden. Diese Form der Ereignissteuerung ist im Wesentlichen obsolet und soll nicht mehr verwendet werden. Ausführbare Programme sollen nur noch dort zum Einsatz kommen, wo dies technisch notwendig ist, im Wesentlichen demnach für die Hintergrundverarbeitung. Auch in diesem Fall soll die eigentliche Implementierung in Methoden erfolgen, beispielsweise über Methoden einer lokalen Klasse innerhalb des ausführbaren Programms. Der Ereignisblock des Einstiegsereignisses `START-OF-SELECTION` soll lediglich aus einem Methodenaufruf bestehen (siehe Regel 6.44), und andere Ereignisblöcke sollten nicht mehr vorkommen.
- Der Modul-Pool war der Programmtyp, der traditionsgemäß bei der klassischen Dialogprogrammierung mit Dynpros zum Einsatz kam. Wie in Abschnitt 2.1 aufgezeigt, wird durch Modul-Pools das Konzept der Trennung der Belange nicht ausreichend unterstützt. Aus diesem Grund sollen keine neuen Modul-Pools mehr angelegt werden. Stattdessen sollen klassische Dynpros, soweit diese noch verwendet werden müssen, in Funktionsgruppen verschalt werden.
- Der Programmtyp Typgruppe wurde anfangs als Notlösung dafür eingeführt, dass im ABAP Dictionary zeitweise noch keine Typen für interne Tabellen definiert werden konnten. Ebenso verhielt es sich mit der globalen Ablage von Konstanten. Beide Lücken sind inzwischen geschlossen. Im ABAP Dictionary können beliebige Typen definiert werden, und in globalen Klassen und Interfaces ist es möglich, sowohl Typen als auch Konstanten zur paket- oder systemweiten Verwendung anzulegen. Aus diesem Grund ist der Programmtyp Typgruppe obsolet, und es sollen keine neuen Typgruppen mehr angelegt werden (siehe Abschnitt 6.1.2, »Deklaration von Datentypen und Konstanten«).

Anmerkung

In den Fällen, in denen noch mit anderen Programmtypen als Class- und Interface-Pools gearbeitet wird, sollte in der erweiterten Programmprüfung (siehe Abschnitt 3.4.2) die Prüfung Veraltete Anweisungen (OO-Kontext) eingeschaltet werden, um auch für die Programmteile, die nicht in lokalen Klassen implementiert sind, die gleiche strengere Syntaxprüfung wie innerhalb von Klassen durchzuführen.

3.2.2 Programmattribute

Hintergrund

Jedes ABAP-Programm hat neben weiteren, weniger wichtigen Eigenschaften einen Satz von Programmattributen, die bestimmte Aspekte des Programmverhaltens und der Syntaxprüfschärfe steuern. Diese sind:

- **Unicode-Prüfungen aktiv**
  zur Erstellung eines Unicode-Programms
- **Festpunktarithmetik**
  für die Berücksichtigung des Dezimaltrennzeichens in Operationen mit gepackten Zahlen
- **Logische Datenbank**
  zur Verknüpfung eines ausführbaren Programms mit einer logischen Datenbank

Die Programmeigenschaften werden beim Anlegen eines Programms im entsprechenden Werkzeug (Class Builder, Function Builder, ABAP Editor) festgelegt und können auch nachträglich noch geändert werden.

Regel

Regel 3.3: Standardeinstellungen für Programmattribute übernehmen

Setzen Sie die Programmattribute für neue Programme wie folgt:

- Unicode-Prüfungen aktiv eingeschaltet
- Festpunktarithmetik eingeschaltet
- keine Zuordnung zu einer logischen Datenbank

Diese Einstellungen entsprechen den Vorschlagswerten beim Anlegen eines neuen Programms, die daher ohne Änderungen übernommen werden sollen. Einmal gesetzte Programmattribute sollten nachträglich nicht mehr abgeändert werden.

Details

Verschiedene Verhaltensweisen oder Prüfschärfen werden nur noch aus Kompatibilitätsgründen angeboten, um bestehende Programme weiterhin kompilier- und ausführbar zu halten. Neue Programme sollen in keinem Fall von veralteten Einstellungen Gebrauch machen.

- Beim Anlegen eines neuen Programms ist das Attribut Unicode-Prüfungen aktiv bereits als Standardeinstellung gesetzt. Dieses Attribut darf niemals zurückgesetzt werden. Nur mit eingeschalteten Unicode-Prüfungen kann

sichergestellt werden, dass das Programm sowohl in Unicode-Systemen als auch in Nicht-Unicode-Systemen lauffähig ist und jeweils die gleichen Ergebnisse liefert.[3] Bei der Vorbereitung eines Nicht-Unicode-Systems zur Umstellung auf Unicode müssen alle noch vorhandenen Nicht-Unicode-Programme in Unicode-Programme umgesetzt werden. Die Aktivierung der Unicode-Prüfungen bringt dem Entwickler ausschließlich Vorteile, beispielsweise in Form einer strengeren statischen Typprüfung und einer strikten Trennung von Byte- und Zeichenkettenverarbeitung.

- Beim Anlegen eines neuen Programms ist das Attribut Festpunktarithmetik bereits als Standardeinstellung gesetzt. Auch dieses Attribut darf niemals zurückgesetzt werden. Bei ausgeschalteter Festpunktarithmetik wird die Stellung des Dezimaltrennzeichens von gepackten Zahlen (Typ `p`) nur bei der Ausgabe auf dem klassischen Dynpro oder bei der Formatierung mittels `WRITE TO` berücksichtigt, nicht jedoch bei Berechnungen. Ein solches Verhalten wird heute nur in den seltensten Fällen den Erwartungen des Entwicklers entsprechen. Soll mit gepackten Zahlen ohne Nachkommastellen gerechnet werden, ist dies über den Zusatz `DECIMALS 0` bei der Deklaration anzugeben.
- Beim Anlegen eines neuen ausführbaren Programms ist das Attribut Logische Datenbank leer. Durch dieses Attribut werden ausführbare Programme einer logischen Datenbank[4] zugeordnet, wodurch das Selektionsbild und der Programmablauf des Programms mit dem Selektionsbild und dem Ablauf der logischen Datenbank kombiniert werden. Logische Datenbanken sollen nicht mehr verwendet werden, da sie auf der programmübergreifenden Nutzung globaler Daten, einem impliziten Unterprogrammaufruf und der Reporting-Ereignissteuerung beruhen und damit modernen Konzepten zuwiderlaufen. Der Zugriff auf bestehende logische Datenbanken kann bei Bedarf über den Funktionsbaustein `LDB_PROCESS` erfolgen, der beispielsweise aus einer Methode heraus aufgerufen werden kann. Neue

3 Ein Programm mit eingeschalteten Unicode-Prüfungen wird als *Unicode-Programm* bezeichnet. Als *Unicode-System* bezeichnet man ein SAP-System, in dem die Zeichendarstellung im Unicode-Format (ISO/IEC 10646) erfolgt (derzeit UTF-16 mit plattformabhängiger Bytereihenfolge). Auf einem Unicode-System können nur Unicode-Programme, Unicode-Programme können aber auch auf Nicht-Unicode-Systemen ausgeführt werden. Die von SAP ausgelieferten Programme sind in der Regel Unicode-Programme.

4 Eine logische Datenbank ist ein spezielles Entwicklungsobjekt, das im *Logical Database Builder* bearbeitet wird und anderen ABAP-Programmen Daten aus den Knoten einer hierarchischen Baumstruktur zur Verfügung stellt. Eine logische Datenbank verfügt über eine hierarchische Struktur, ein in ABAP geschriebenes Datenbankprogramm und ein eigenes Standardselektionsbild.

logische Datenbanken sollen nicht mehr angelegt werden. Stattdessen soll ein entsprechender Service über eine globale Klasse angeboten werden.

Da eine nachträgliche Änderung von Programmeigenschaften potenziell mit Umstellungsaufwand verbunden ist, sollten die richtigen Eigenschaften von Anfang an eingestellt und nicht mehr geändert werden. Insbesondere bei Attributen, die die Syntaxprüfung beeinflussen (derzeit die Unicode-Prüfung) sollte man sich immer gleich für die größtmögliche Prüfschärfe entscheiden, um bei späteren eventuell angeordneten Umstellungen bestens vorbereitet zu sein.

Im Folgenden gehen wir davon aus, dass nur noch mit eingeschalteter Unicode-Prüfung und Festpunktarithmetik und ohne logische Datenbanken gearbeitet wird. Für veraltete oder problematische Sprachkonstrukte, die nur noch bei ausgeschalteten Unicode-Prüfungen verfügbar sind, wird in diesen Richtlinien daher keine spezielle Regel mehr erstellt. Wir erwähnen sie nur kurz im Rahmen der Liste der obsoleten Sprachelemente (siehe Anhang A).

Schlechtes Beispiel

Abbildung 3.1 zeigt ein ABAP-Programm, bei dem in den Programmeigenschaften das Attribut UNICODE-PRÜFUNGEN AKTIV entgegen der Empfehlung von Regel 3.3 nicht ausgewählt ist.

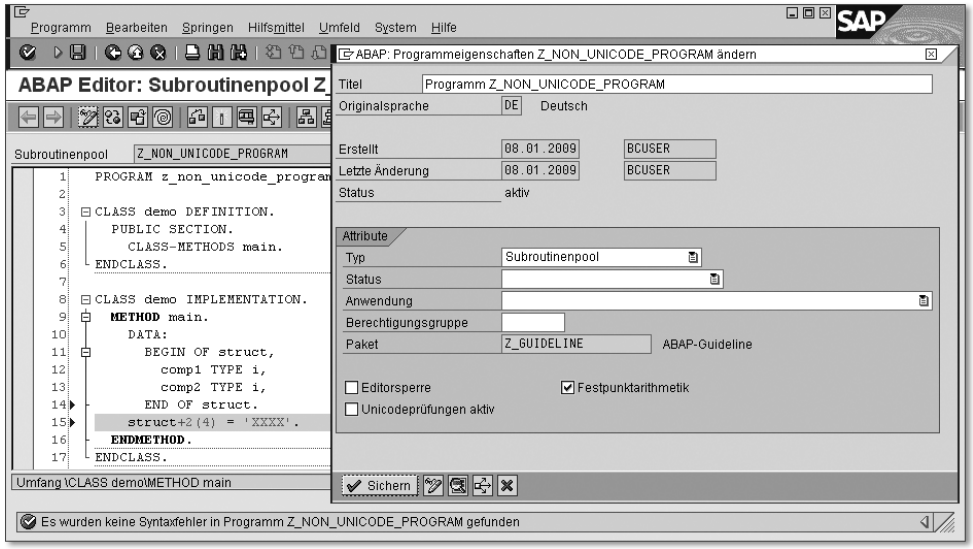


Abbildung 3.1 Erlaubter Teilfeldzugriff auf eine Struktur in einem Nicht-Unicode-Programm

In dem Nicht-Unicode-Programm aus Abbildung 3.1 ist es straflos möglich, einen schreibenden Teilfeldzugriff über zwei numerische Komponenten einer

Struktur hinweg auszuführen, wobei – horribile dictu – ein implizites Casting (siehe Abschnitt 6.2.8) des Teilbereichs auf den Typ c stattfindet. Das Ergebnis in den Komponenten ist abhängig von Ausrichtungslücken, der internen Darstellung numerischer Werte (Bytereihenfolge) sowie der verwendeten Codepage und damit extrem plattformabhängig. Ein produktives Programm darf keinesfalls solchen Code enthalten. Es führt in der Regel zu fehlerhaften Daten oder zu schwer nachvollziehbaren Laufzeitfehlern.

Gutes Beispiel

Abbildung 3.2 zeigt ein ABAP-Programm, bei dem in den Programmeigenschaften nach Regel 3.3 das Attribut Unicode-Prüfungen aktiv ausgewählt ist.

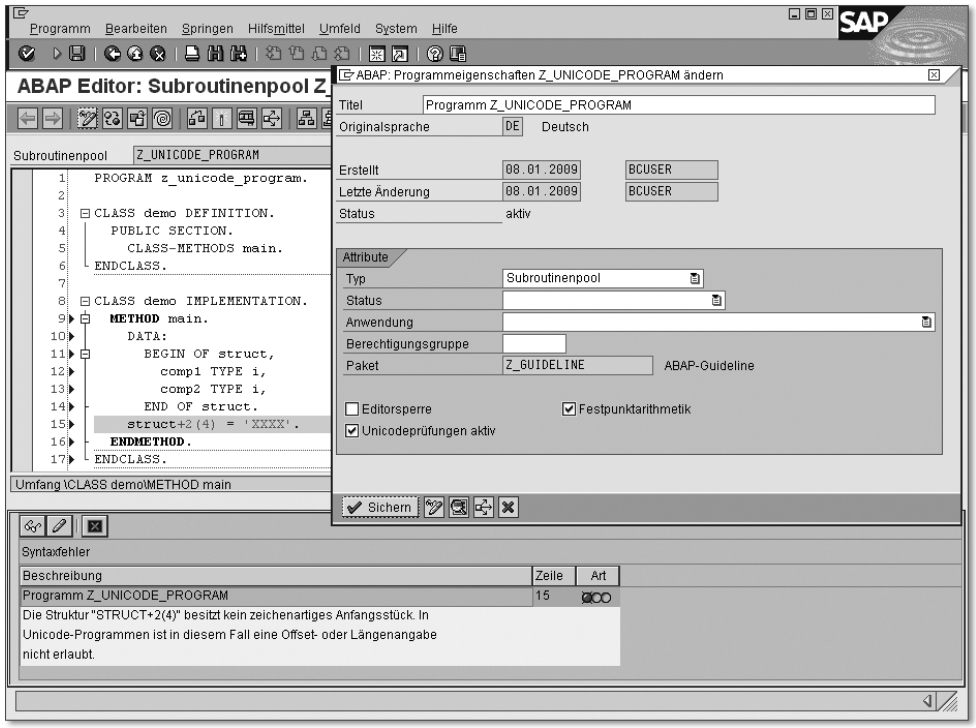


Abbildung 3.2 Syntaxfehler bei Teilfeldzugriff auf eine Struktur in einem Unicode-Programm

Im Unicode-Programm aus Abbildung 3.2 führt der Code aus Abbildung 3.1 zu einem Syntaxfehler. Unerwünschte Teilfeldzugriffe sind wie andere unerwünschte Zugriffe auf Strukturen oder andere Teile des Arbeitsspeichers verboten. Falls statisch erkennbar, führen diese wie im hier gezeigten Beispiel zu einem Syntaxfehler. Anderenfalls kommt es während der Programmausführung zu einem Laufzeitfehler mit einem aussagekräftigen Kurzdump.

3.2.3 Originalsprache

Hintergrund

Beim Anlegen eines neuen Repository-Objektes (beispielsweise eines Programms, einer Klasse oder einer Datenbanktabelle im ABAP Dictionary) muss seine Originalsprache festgelegt werden. Dies passiert implizit durch die aktuelle Anmeldesprache. Alle während der Entwicklung angelegten übersetzungsfähigen Texte des Entwicklungsobjektes, wie zum Beispiel beschreibende Kurz- und Langtexte, die Textelemente eines Programms und auch die Dokumentation von Datentypen oder Schnittstellen, werden der angegebenen Originalsprache zugeordnet. Die Erstellung der Texte in anderen Sprachen erfolgt durch einen von der Entwicklung losgelösten Übersetzungsvorgang aus der Originalsprache in die Zielsprachen.

Derzeit gibt es keine technische Unterstützung für die projektweite Ersetzung einer einmal gewählten Originalsprache durch eine andere Sprache.

Regel

Regel 3.4: Originalsprache auf Projektebene festlegen

Legen Sie vor Beginn der Implementierung eine sorgfältig ausgewählte Originalsprache auf Projektebene für die Repository-Objekte fest. Entwickler dürfen ihre Entwicklungsobjekte nur in der für das jeweilige Projekt (oder in Ausnahmefällen für ein Teilprojekt) festgelegten Originalsprache anlegen.

Details

Bei der Festlegung der Originalsprache soll wie folgt vorgegangen werden:

- Bei einsprachiger Besetzung aller an einem Projekt beteiligten Entwicklungsgruppen ist die Originalsprache aller Entwicklungsobjekte die Muttersprache aller beteiligten Entwickler (einsprachige Entwicklung).
- Bei mehrsprachiger Besetzung der Entwicklungsgruppen
  - ist die Originalsprache aller Entwicklungsobjekte entweder eine von allen Beteiligten verstandene Sprache – in der Regel Englisch – (einsprachige Entwicklung)
  - oder richtet sich die Originalsprache von Entwicklungsobjekten in Teilen des Projektes nach der Muttersprache der hauptsächlich daran arbeitenden Entwickler (mehrsprachige Entwicklung).

Einsprachige Entwicklungsgruppen stellen sozusagen den Idealfall dar, sind heutzutage aber nicht immer zu realisieren. Die beiden möglichen Einstellungen für mehrsprachige Entwicklergruppen – einsprachige und mehrsprachige Entwicklung – erfüllen zwei unterschiedliche Anforderungen, die sich aber widersprechen:

- Bei der Anmeldung an einem System in einer anderen Sprache als der Originalsprache lässt sich im Allgemeinen nicht sinnvoll mit einem in der Entwicklung befindlichen oder neu entwickelten Produkt arbeiten, bis eine Übersetzung der relevanten Texte in die jeweilige Zielsprache vorliegt. Die Übersetzung erfolgt in der Regel in einem nachgelagerten Übersetzungssystem und muss in das Entwicklungssystem zurücktransportiert werden. Aus diesem Grund ist eine effiziente Entwicklung, insbesondere in international besetzten Entwicklungsgruppen (die eventuell auch noch über mehrere Standorte verteilt sind), nur dann möglich, wenn zu Beginn projektweit eine einheitliche Originalsprache festgelegt wird, die es allen am Entwicklungs- und Validierungsprozess beteiligten Personen erlaubt, das Produkt zumindest testweise zu verwenden. Bei einsprachiger Entwicklung in mehrsprachigen Entwicklungsgruppen müssen daher einige, wenn nicht gar alle Entwickler eines Projektes Texte in einer Sprache anlegen, die nicht ihre Muttersprache ist.
- Für die sprachliche und stilistische Überprüfung von Oberflächentexten und Dokumentationen, die von Entwicklern in anderen Sprachen als ihrer Muttersprache angelegt werden, gibt es in der Regel keine Unterstützung in Form von Werkzeugen oder definierten Abläufen. Daher wäre es wünschenswert, dass die an der Entwicklung von Benutzerdialogen und Dokumentationen beteiligten Entwickler idealerweise in ihrer Muttersprache arbeiten und diese Texte dann von geschulten Übersetzern anhand von vorgegebener Terminologie in deren Muttersprache übersetzt werden.

Der zweite Punkt ist der Grund, warum nicht Englisch als allumfassende einheitliche Originalsprache für alle Entwicklungsprojekte gefordert wird, sondern dass einsprachige Entwicklungsgruppen durchaus in ihrer Muttersprache mit eventueller nachgelagerter Übersetzung arbeiten sollten.

Bei mehrsprachigen Entwicklungsgruppen kommt es letztendlich auf den konkreten Fall an, welche Originalsprache für jedes Entwicklungsobjekt festgelegt wird. In der Regel wiegt der erste Punkt schwerer, sodass bei internationaler Entwicklung eine einsprachige Entwicklung durchgeführt werden muss, um die Entwicklungsressourcen für ein Projekt möglichst effektiv zu nutzen. In Einzelfällen kann es bei Teilprojekten, in denen besonders viel Text angelegt

werden muss, durchaus auch sinnvoll sein, die Originalsprache gemäß der Muttersprache der Entwickler festzulegen.[5]

Bei mehrsprachigen Projekten sollten betriebswirtschaftlich zusammengehörige Funktionen sprachenrein entwickelt werden, zumindest auf Paketebene. Auch Tabelleninhalte sollten in einer einheitlichen Sprache angelegt werden.

Hinweis

Da die Originalsprache beim Anlegen eines Repository-Objekts durch die Anmeldesprache festgelegt wird, muss für das Anlegen und Bearbeiten von Repository-Objekten ganz bewusst die Entscheidung für eine Anmeldesprache getroffen werden.

Anmerkung

Unabhängig davon, ob eine ein- oder mehrsprachige Entwicklung innerhalb eines Projektes durchgeführt wird, muss vor Entwicklungsbeginn immer eine einheitliche *Terminologie* für alle im Projekt angelegten Texte erstellt und diese durchgängig befolgt werden. Bei einer mehrsprachigen Entwicklung sollte die Übersetzung der Terminologiebegriffe in die verwendeten Sprachen möglichst vor Beginn der Entwicklung vorgenommen werden, damit sie von den Entwicklern verwendet werden können. Zudem müssen immer die existierenden Standards für Oberflächentexte und Dokumentation befolgt werden (siehe Abschnitt 2.3, »Korrektheit und Qualität«).

3.3 Modernes ABAP

Hintergrund

ABAP ist eine lebendige Programmiersprache, die kontinuierlich weiterentwickelt wird. Seit der Einführung von ABAP vor etwa 30 Jahren entstehen laufend neue ABAP-Programme, während parallel dazu an der Sprache ABAP selbst gearbeitet wird. Weiterentwicklungen an der Sprache ABAP sind entweder Erweiterungen der vorhandenen Spracheigenschaften, um neue Funktionalität einzuführen, oder der Ersatz vorhandener Funktionalität durch fortgeschrittenere Konzepte. Der Ersatz vorhandener durch neue Sprachelemente macht die vorhandenen in der Regel überflüssig bzw. obsolet. Das prominen-

5 Dies betrifft insbesondere die SAP-eigene Entwicklung, bei der nach wie vor größere Anteile von deutschsprachigen Entwicklern ausgeführt werden.

teste Beispiel einer Weiterentwicklung der Sprache ABAP ist nach wie vor die Einführung von ABAP Objects zu Release 4.6.

SAP hat sich bezüglich der Sprache ABAP einer Politik der strikten Abwärtskompatibilität verschrieben. Das bedeutet zum einen, dass ein beispielsweise zu R/3-Release 3.0 geschriebenes ABAP-Programm auf einem AS ABAP in Release 7.2 unverändert ausgeführt werden kann, zumindest solange es sich um ein Nicht-Unicode-System handelt. Auf der anderen Seite bedeutet es aber auch:

- Ein erfahrener Entwickler wurde bisher durch fast nichts gezwungen, alte Gewohnheiten abzulegen und sich mit neuen Konzepten zu beschäftigen. Die einzige Ausnahme stellt die Umstellung auf Unicode-Systeme dar, für die ABAP-Programme in Unicode-Programme mit leicht veränderten Syntaxregeln umgewandelt werden müssen.
- ABAP-Einsteiger werden durch die Vielfalt der Möglichkeiten verwirrt, die es gibt, um ein und dasselbe zu tun. Wenn dann im Zweifelsfall ältere Programme als Vorlagen dienen, kommen oft statt der neuen weiterhin die obsoleten Konzepte zum Einsatz.

Um diesen Problemen abzuhelfen, gibt es folgende einfache Regel.

Regel

Regel 3.5: Keine obsoleten Sprachmittel verwenden

Verwenden Sie für Neuentwicklungen keine obsoleten Sprachmittel. Auch für bestehende Programme wird eine inkrementelle Umstellung auf neuere Konzepte empfohlen, wie sie zur Verfügung stehen.

Details

Neuere Sprachelemente sind immer die besseren Sprachelemente. Obsolete Sprachmittel werden nur aus Gründen der Abwärtskompatibilität weiterhin angeboten. Eine Anweisung oder ein Anweisungszusatz wird erst dann für obsolet erklärt, wenn eine leistungsfähigere Alternative existiert oder das Sprachelement als fehlerträchtig (in dem Sinne, dass es zu unsicherer und nicht robuster Programmierung einlädt) erkannt wurde. Aus diesem Grund ist eine sichere und robuste Programmierung nicht mit dem Einsatz obsoleter Sprachelemente zu vereinbaren. Damit verbietet sich die Verwendung solcher obsoleter Sprachmittel im Rahmen der Neuentwicklung.

Bei der Verwendung von ABAP Objects ist ein Großteil der obsoleten Anweisungen und Zusätze bereits syntaktisch verboten. Unter anderem aus diesem Grund wird die Verwendung von ABAP Objects unbedingt empfohlen (siehe Regel 3.1). Außerhalb von ABAP Objects, das heißt in den Fällen, die nach Abschnitt 3.1, »ABAP Objects als Programmiermodell«, noch erlaubt sind, muss selbst Sorge dafür getragen werden, dass keine obsoleten Sprachelemente zum Einsatz kommen. Hierfür liefert Anhang A, »Obsolete Sprachkonstrukte«, eine Übersicht der obsoleten Anweisungen und Anweisungszusätze.

Schlechtes Beispiel

Listing 3.3 zeigt die Lösung einer Aufgabe unter Verwendung obsoleter Sprachmittel. Eine Prozedur soll in einem Text `text` alle Vorkommen einer Unterfolge `substring` durch eine neue Zeichenfolge `new` ersetzen, falls die Unterfolge nicht am Ende eines Wortes steht.

```
FORM bad_example USING    substring TYPE csequence
                          new       TYPE csequence
                CHANGING text      TYPE csequence.
  DATA: pattern TYPE string,
        subrc   TYPE sy-subrc.
  CONCATENATE '*' substring INTO pattern.
  SEARCH text FOR pattern.
  IF sy-subrc <> 0.
    CLEAR subrc.
    WHILE subrc = 0.
      REPLACE substring WITH new INTO text.
      subrc = sy-subrc.
    ENDWHILE.
  ENDIF.
ENDFORM.
```

Listing 3.3 Verwendung obsoleter Sprachmittel

In Listing 3.3 sind, abgesehen von der Modularisierung mit `FORM-ENDFORM`, die Anweisung `SEARCH` und die verwendete Variante von `REPLACE` ab Release 7.0 obsolet. Darüber hinaus steht ab den Releases 7.0 EhP2 und 7.2 ein Zeichenkettenoperator `&&` als Ersatz für `CONCATENATE` zur Verfügung.

Gutes Beispiel

Listing 3.4 führt die gleiche Aufgabe wie Listing 3.3 unter Verwendung der neuesten zur Verfügung stehenden Sprachelemente aus.

```
METHOD good_example.
  FIND REGEX substring && `\b` IN text.
  IF sy-subrc <> 0.
    REPLACE ALL OCCURRENCES OF substring IN text WITH new.
  ENDIF.
ENDMETHOD.
```

Listing 3.4 Verwendung moderner Sprachmittel

Das Unterprogramm wird durch eine Methode ersetzt. Durch Verwendung von `FIND` in Zusammenhang mit einem regulären Ausdruck, der über den Zeichenkettenoperator `&&` zusammengesetzt wird, ist keine Hilfsvariable mehr nötig. Die `WHILE`-Schleife wird durch `REPLACE ALL OCCURRENCES` ersetzt, wobei eine weitere Hilfsvariable entfällt und der Kontrollfluss in die ABAP-Laufzeitumgebung verschoben wird. Letzteres erhöht die Ausführungsgeschwindigkeit und ist auch zur Erfüllung von Regel 4.22 zur Beschränkung der maximalen Schachtelungstiefe hilfreich.

Anmerkung

Im Zusammenhang mit Regel 3.5, »Keine obsoleten Sprachmittel verwenden«, stellt sich die Frage, wie es mit der Koexistenz alter und neuer Konzepte innerhalb einer Programmeinheit aussieht. Es gibt nur eine Stelle, an der dies syntaktisch klar geregelt ist, nämlich die Verwendung des klassischen und des klassenbasierten Ausnahmekonzeptes (siehe Abschnitt 5.2.2) in Verarbeitungsblöcken. Anderenfalls können obsolete Sprachelemente in einem Programmteil direkt neben neuen Sprachelementen stehen. Unsere Empfehlung hierzu ist, die Verwendung innerhalb eines Kontextes möglichst einheitlich zu gestalten, das heißt nicht verschiedene Anweisungen, wie zum Beispiel `FIND` und `SEARCH`, nebeneinander zum gleichen Zweck einzusetzen.

Dies soll aber nicht bedeuten, dass bei Erweiterungen an bestehenden Prozeduren aus Gründen der Einheitlichkeit weiterhin obsolete Sprachelemente verwendet werden sollen, nur weil sie dort bereits vorhanden sind. Vielmehr sollte man die Gelegenheit ergreifen und gleich die gesamte Prozedur auf die entsprechenden neuen Sprachelemente umstellen. Durch die Abdeckung der zu ändernden Prozeduren mit Modultests kann sichergestellt werden, dass es bei einer solchen Umstellung nicht zu Überraschungen kommt.

3.4 Prüfungen auf Korrektheit

In Abschnitt 2.3, »Korrektheit und Qualität«, wurde bereits allgemein auf die Korrektheit und Qualität von Programmen eingegangen, und die für deren Überprüfung vorhandenen Werkzeuge wurden kurz vorgestellt. Der vorliegende Abschnitt beschäftigt sich nochmals speziell mit der syntaktischen Korrektheit von ABAP-Programmen, die mit der Syntaxprüfung und der erweiterten Programmprüfung kontrolliert wird, sowie mit der Standardprüfung des Code Inspectors und dem neuen ABAP-Testcockpit.

3.4.1 Syntaxprüfung

Hintergrund

Die Syntaxprüfung liefert Syntaxfehler und Syntaxwarnungen.

- Sobald ein Syntaxfehler auftritt, wird die Prüfung beendet und eine entsprechende Fehlermeldung angezeigt. In vielen Fällen wird eine Korrektur vorgeschlagen, die übernommen werden kann. Ein Programm mit Syntaxfehlern ist zwar aktivierbar, kann aber nicht generiert und damit nicht ausgeführt werden. Syntaxfehler werden von der erweiterten Programmprüfung als fatale Fehler gemeldet. Syntaxfehler müssen unbedingt behoben werden.
- Tritt eine Syntaxwarnung auf, wird die Syntaxprüfung nicht beendet, und das Programm ist im Prinzip ausführbar. Die Syntaxwarnungen werden nach einer Ausführung der Syntaxprüfung im ABAP Editor und auch von der erweiterten Programmprüfung (siehe Abschnitt 3.4.2) angezeigt.[6] Bei der Aktivierung eines Programms werden Syntaxwarnungen nur dann ausgegeben, wenn es gleichzeitig auch Syntaxfehler gibt.

  Die von der Syntaxprüfung gemeldeten Warnungen sind in drei Prioritäten unterteilt, die aber nur von der erweiterten Programmprüfung angezeigt werden:

  - **Priorität 1**
    Fehler, die erkennbar zu einem Programmabbruch bei der Ausführung des ABAP-Programms führen werden. Zudem alle Konstrukte, die keinesfalls verwendet werden sollen, da sie auf Programmierfehler hindeuten und höchstwahrscheinlich zu falschem Verhalten führen.

6 Natürlich zeigen Testwerkzeuge, die die Prüfungen der erweiterten Programmprüfung umfassen, wie der Code Inspector und das SAP-interne ABAP-Testcockpit (seit den Releases 7.0 EhP2 und 7.2), die Syntaxwarnungen ebenfalls an.

- **Priorität 2**
  Alle Konstrukte, die nicht unbedingt zu Fehlverhalten führen, aber zum Beispiel obsolet sind und durch aktuelle Konstrukte ersetzt werden sollen. Fehler der Priorität 2 können in zukünftigen Releases zu Fehlern der Priorität 1 oder zu Syntaxfehlern werden.
- **Priorität 3**
  Fasst alle Fehler zusammen, deren Behebung zwar wünschenswert, aber nicht unbedingt für das aktuelle Release notwendig ist. Eine Verschärfung der Priorität in kommenden Releases ist jedoch nicht ausgeschlossen.

Die Prüfschärfe der ABAP-Syntaxprüfung wird durch die beim Anlegen eines Programms getroffenen Entscheidungen bestimmt (siehe Abschnitt 3.2, »Programmtyp und Programmeigenschaften«). So können Programmkonstrukte, die außerhalb von Klassen oder in Nicht-Unicode-Programmen nur zu Syntaxwarnungen führen, in Klassen oder in Unicode-Programmen echte Syntaxfehler darstellen. Seit den Releases 7.0 EhP2 und 7.2 können ausgesuchte Syntaxwarnungen durch sogenannte Pragmas[7] unterdrückt werden.

Mit der Einführung des operationalen Paketkonzeptes ab Release 7.2 überprüft die Syntaxprüfung auch Paketverletzungen. Dabei hängt es von der beim betreffenden Paket eingestellten Kapselungsstärke ab, ob es zu einem Syntaxfehler oder lediglich zu einer Syntaxwarnung kommt.

Regel

Regel 3.6: Syntaxwarnungen beachten [✓]

Nehmen Sie alle Warnungen der ABAP-Syntaxprüfung ernst. In einem fertiggestellten Programm dürfen keine Syntaxwarnungen mehr auftreten.

Details

Die Ursachen von Syntaxwarnungen müssen immer korrigiert werden, da sie im Allgemeinen zu unvorhersagbaren Fehlern führen. Solche Warnungen werden von SAP häufig in einem späteren Release des AS ABAP zu Fehlern heraufgestuft. In diesem Fall ist dann ein zunächst nur mit Syntaxwarnungen behaftetes Programm nach einem Upgrade syntaktisch falsch und nicht mehr benutzbar. Genauso verhält es sich bei der Umstellung von Nicht-Unicode-Pro-

7 Ein Pragma ist eine Programmdirektive, die den Programmablauf nicht beeinflusst, sondern Auswirkung auf bestimmte Überprüfungen hat.

grammen auf Unicode-Programme oder bei der Migration älterer Programmteile nach ABAP Objects.

Bezüglich der Paketprüfung stellt die konsequente Verwendung des bereits vor Release 7.2 zur Verfügung stehenden Paketkonzeptes (Auswahl von PAKETPRÜFUNG ALS SERVER im Package Builder) bzw. die Einstellung einer schwachen Kapselung ab Release 7.2 einen ersten Schritt auf dem Weg zur echten Kapselung dar. Sie ermöglicht den Verwendern von Entwicklungsobjekten eine Anpassung ihrer Verwendungsstellen, noch bevor es zu harten Syntaxfehlern kommt. Aus diesem Grund müssen sowohl vor als auch nach Release 7.2 insbesondere alle Warnungen der Paketprüfung ernst genommen und behoben werden, damit das Programm auch nach einer verschärften Kapselung der verwendeten Pakete syntaktisch korrekt bleibt.

Schlechtes Beispiel

Abbildung 3.3 zeigt einen Ausschnitt eines Nicht-Unicode-Programms in einem Nicht-Unicode-System, in dem eine `VALUE`-Angabe zu einer Syntaxwarnung führt, weil ein nicht typgerechter Startwert für eine Struktur gesetzt wird.

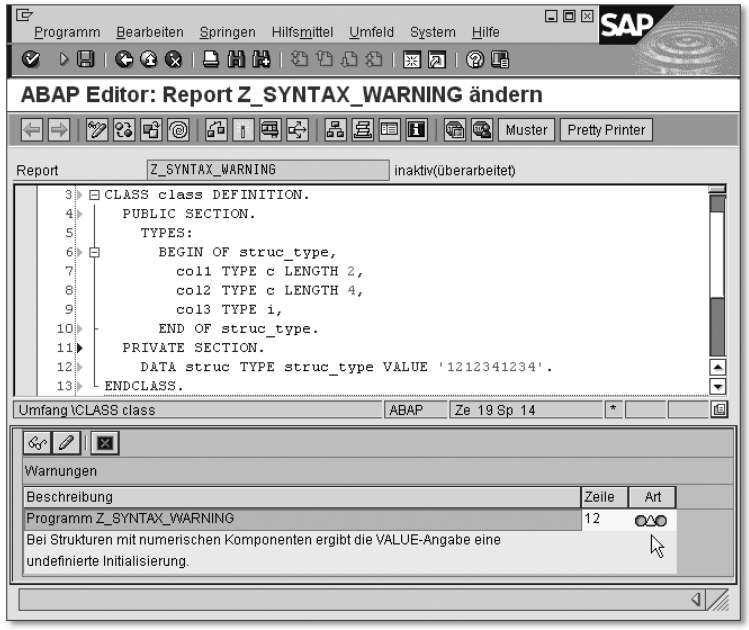


Abbildung 3.3 Programm mit Syntaxwarnung

Anmerkung

In einem Unicode-Programm – das heißt einem Programm, für das die Programmeigenschaft UNICODE-PRÜFUNGEN AKTIV gesetzt ist – führt die Anweisung, die in Abbildung 3.3 nur zu einer Warnung führt, zu einem Syntaxfehler.

Gutes Beispiel

Abbildung 3.4 zeigt das korrigierte Programm aus Abbildung 3.3. Die Komponenten der Struktur werden im Instanzkonstruktor typgerecht mit Startwerten versorgt. Das Programm ist frei von Syntaxwarnungen und auch als Unicode-Programm korrekt.

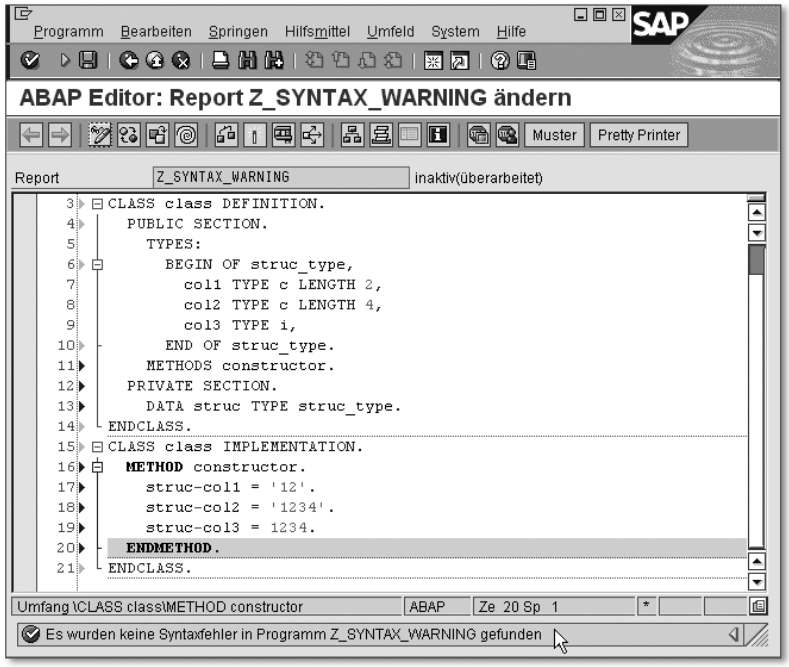


Abbildung 3.4 Korrektes Programm ohne Syntaxwarnung

3.4.2 Erweiterte Programmprüfung

Hintergrund

Die erweiterte Programmprüfung kann für aktivierte Programme entweder aus der ABAP Workbench heraus oder über die Transaktion SLIN aufgerufen werden. Sie führt statische Prüfungen durch, die für die normale Syntaxprüfung zu aufwendig sind. Es können entweder einzelne oder mehrere Teiltests oder eine Standardprüfung durchgeführt werden, die die wichtigsten Teiltests umfasst.

Die erweiterte Programmprüfung gibt Fehler, Warnungen und Meldungen aus. Von der Standardprüfung werden die Fehler und Warnungen gemeldet, die besonders kritisch sind.[8] Darüber hinaus werden immer auch die Fehler und Warnungen der Syntaxprüfung angezeigt.

Seit den Releases 7.0 EhP2 und 7.2 kann im Einstiegsbild der erweiterten Programmprüfung auch eine Prüfung von Programmierrichtlinien ausgewählt werden, die die Einhaltung einiger der in diesem Buch vorgestellten Regeln, die statisch verifiziert werden können, überprüft.

Die Meldungen der erweiterten Programmprüfung, die in speziellen Sonderfällen unzutreffend sind, können über Pseudokommentare und seit den Releases 7.0 EhP2 und 7.2 über Pragmas ausgeblendet werden. Meldungen, die direkt von der normalen Syntaxprüfung kommen, ließen sich vor den Releases 7.0 EhP2 und 7.2 nicht ausblenden.

Regel

[✓] **Regel 3.7: Erweiterte Programmprüfung verwenden**

Verwenden Sie die erweiterte Programmprüfung, und nehmen Sie ihre Ergebnisse ernst. Für ein fertiggestelltes Programm dürfen keine Meldungen der Standardprüfung mehr auftreten.

Details

Die von der erweiterten Programmprüfung ausgegebenen Fehler, Warnungen und Meldungen sind genauso wichtig wie die Syntaxfehler und Syntaxwarnungen der Syntaxprüfung (siehe Abschnitt 3.4.1). Ein von der erweiterten Programmprüfung gemeldeter Fehler kann zum Beispiel darauf hinweisen, dass ein Programm bei der Ausführung sicher zu einem Laufzeitfehler führt. Warnungen und Meldungen weisen in der Regel auf die fragwürdige Verwendung von Sprachelementen hin, die aller Voraussicht nach zu unerwartetem Programmverhalten führt.

In den seltenen Fällen, in denen ein von der erweiterten Programmprüfung gemeldetes Prüfergebnis unberechtigt ist, muss dies durch einen geeigneten Pseudokommentar oder ein Pragma (seit den Releases 7.0 EhP2 und 7.2) dokumentiert werden (der geeignete Pseudokommentar bzw. das Pragma wird in

8 Die Einstufung eines einzelnen Ergebnisses als Fehler, Warnung oder Meldung kann variieren, je nachdem, ob eine Standardprüfung oder aber explizit ausgewählte Einzelprüfungen durchgeführt werden.

der Meldung jeweils genannt). Dadurch wird diese Meldung der erweiterten Programmprüfung unterdrückt. Idealerweise sollte in weniger offensichtlichen Situationen ein zusätzlicher Kommentar erläutern, warum an dieser Stelle die Meldung nicht zutreffend ist.

Hinweis

Die erweiterte Programmprüfung ist eine wertvolle Hilfe beim Schreiben korrekter ABAP-Programme. Dieser Vorteil darf nicht durch Verwendung unspezifischer Pseudokommentare oder Pragmas zunichte gemacht werden. Insbesondere sollte die Anweisung

```
SET EXTENDED CHECK OFF.
```

niemals verwendet werden, die alle Meldungen der erweiterten Programmprüfung für einen gesamten Quelltextabschnitt unterdrückt.

Wird ein ABAP-Programm einem Code-Review unterzogen, sollten die Ergebnisse der erweiterten Programmprüfung zur Beurteilung der Qualität mit herangezogen werden.

Schlechtes Beispiel

Abbildung 3.5 zeigt das Ergebnis einer Teilprüfung der erweiterten Programmprüfung, die seit den Releases 7.0 EhP2 und 7.2 ausgeführt wird. Sie macht auf eine äußerst fragwürdige Abfrage des Inhalts von `sy-subrc` aufmerksam (siehe Abschnitt 6.3.4, »Rückgabewert«).

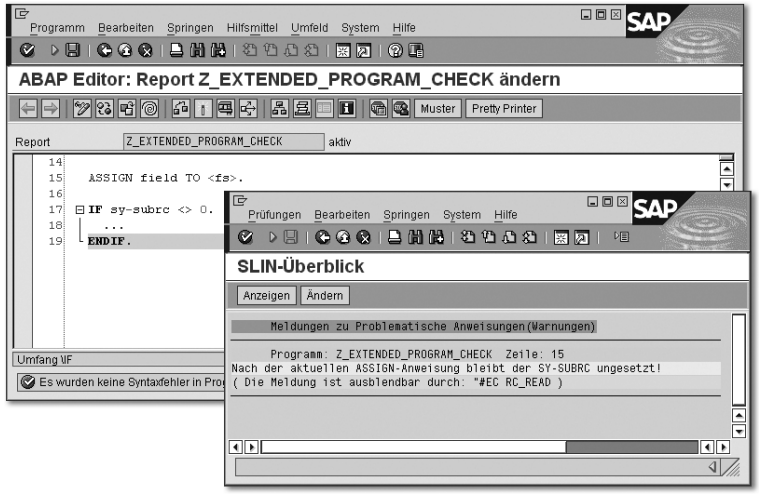


Abbildung 3.5 Warnung der erweiterten Programmprüfung

Der Programmabschnitt zeigt einen typischen Fehler in einem syntaktisch korrekten Programm. Der Entwickler nimmt fälschlicherweise an, dass die statische Form der Anweisung `ASSIGN` das Systemfeld `sy-subrc` setzt, was aber nicht der Fall ist. Dies hat zur Folge, dass er sich zum einen in der falschen Sicherheit wiegt, sein Programm abgesichert zu haben, und zum anderen ein falsches Programmverhalten auftritt, wenn `sy-subrc` von vorhergehenden Anweisungen her einen Wert ungleich null hat. Der große Vorteil der erweiterten Programmprüfung ist daher, dass nicht nur einzelne Anweisungen auf syntaktische Korrektheit, sondern ganze Programmabschnitte auf semantische Fehler hin untersucht werden.

Gutes Beispiel

Abbildung 3.6 zeigt die korrigierte Fassung des Programms aus Abbildung 3.5. Statt der falschen Abfrage von `sy-subrc` wird der in der Dokumentation empfohlene logische Ausdruck `IS ASSIGNED` verwendet. Die Meldung der erweiterten Programmprüfung wäre zwar auch durch einen Pseudokommentar `"#EC RC_READ` oder ein Pragma `##SUBRC_READ` (seit den Releases 7.0 EhP2 und 7.2) ausblendbar, aber das wird in einem solchen Fall wie hier gerade nicht empfohlen, da die erweiterte Programmprüfung auf ein echtes Problem hinweist.

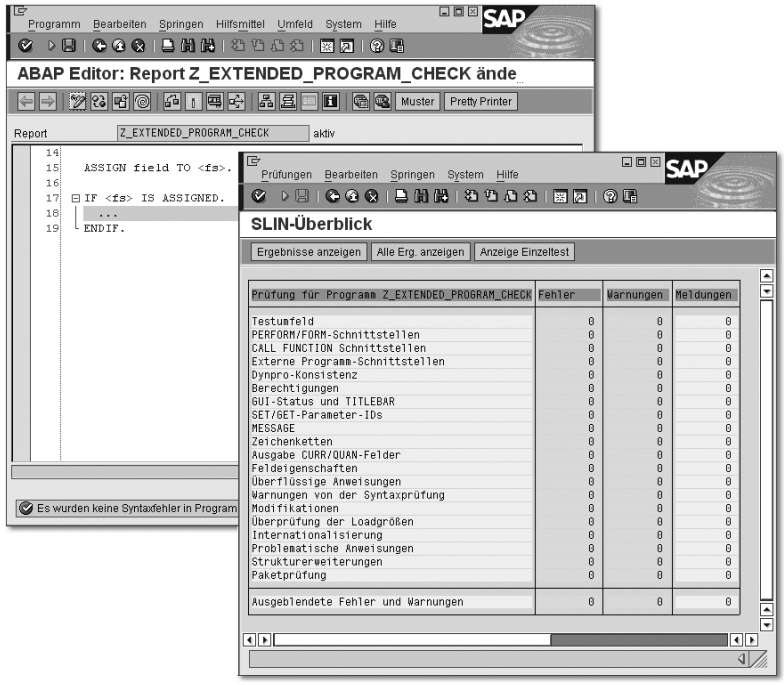


Abbildung 3.6 Erweitere Programmprüfung ohne Meldung

3.4.3 Code Inspector

Hintergrund

Der Code Inspector ist ein Werkzeug zur statischen Überprüfung von Repository-Objekten bezüglich Performance, Sicherheit, Syntax und der Einhaltung von Namenskonventionen. Der volle Funktionsumfang des Code Inspectors kann über die Transaktion SCI verwendet werden, um komplexe statische Prüfungen sowie regelmäßige Massentests für große Mengen von Entwicklungsobjekten auszuführen.

Der Code Inspector kann auch aus der ABAP Workbench heraus aufgerufen werden, um eine Standardmenge an Prüfungen für ihr aktuelles Objekt durchzuführen, wie zum Beispiel über den Menüpfad Programm • Prüfen • Code Inspector des ABAP Editors. Die hierbei verwendete Standardprüfvariante enthält die meisten Prüfungen der erweiterten Programmprüfung (siehe Abschnitt 3.4.2) sowie einige weitere Sicherheits- und Performanceprüfungen. Weiterhin kann der Code Inspector in die Freigabe von Transporten eingebunden werden.

Wie bei der erweiterten Programmprüfung sind auch die Ergebnisse des Code Inspectors in die drei Kategorien Fehler, Warnungen und einfache Meldungen unterteilt und können mit speziellen Pseudokommentaren ausgeblendet werden.

Regel

Regel 3.8: Standardprüfvariante des Code Inspectors verwenden [✓]

Führen Sie die Standardprüfvariante des Code Inspectors vor der Freigabe eines Programms aus, und beseitigen Sie sämtliche Fehlermeldungen.

Details

Wird Regel 3.7, »Erweiterte Programmprüfung verwenden«, beachtet, meldet die Standardprüfvariante des Code Inspectors nur noch Meldungen von Prüfungen, die über die erweiterte Programmprüfung hinausgehen. Dies sind im Wesentlichen Meldungen über eventuelle Performance- oder Sicherheitsrisiken in Programmen. Beispiele sind Meldungen über ungünstige `WHERE`-Bedingungen beim `SELECT`, die Wertübergabe statt der Referenzübergabe von Parametern oder unsichere Programmaufrufe.

Diese Probleme sind, verglichen mit den Meldungen der erweiterten Programmprüfung, nicht immer so einfach an der Ursache zu korrigieren, bei-

spielsweise weil es keine andere Möglichkeit für eine Selektion gibt oder die Übersichtlichkeit oder Robustheit eines Konstruktes als wichtiger als ein eventueller kleiner Performanceverlust angesehen wird.

In solchen Fällen können die Meldungen mit den passenden Pseudokommentaren unterdrückt werden. Ein solcher Pseudokommentar drückt für den Leser des Programms klar aus, dass der Programmautor die entsprechenden Überprüfungen durchgeführt hat und er die Meldung bewusst und aus guten Gründen unterdrückt. Letzteres kann, wo notwendig, durch zusätzliche normale Kommentare erhärtet werden (siehe Abschnitt 4.3).

Schlechtes Beispiel

Abbildung 3.7 zeigt das Ergebnis eines Code-Inspector-Laufs für eine Beispielklasse. Es werden Warnungen ausgegeben, weil eine interne Tabelle per Wertübergabe zurückgegeben und in der `SELECT`-Anweisung ein Inner Join für Datenbanktabellen mit eingeschalteter SAP-Pufferung verwendet wird.

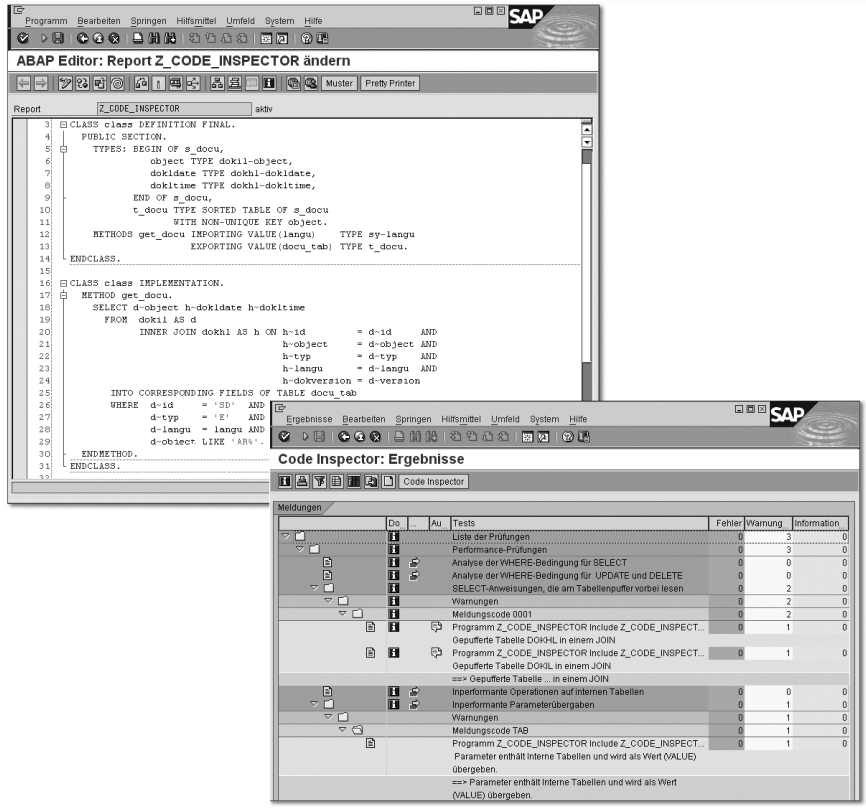


Abbildung 3.7 Warnungen des Code Inspectors

Gutes Beispiel

Abbildung 3.8 zeigt die korrigierte Fassung des Programms aus Abbildung 3.7, für die der Code Inspector keine Meldungen mehr ausgibt.

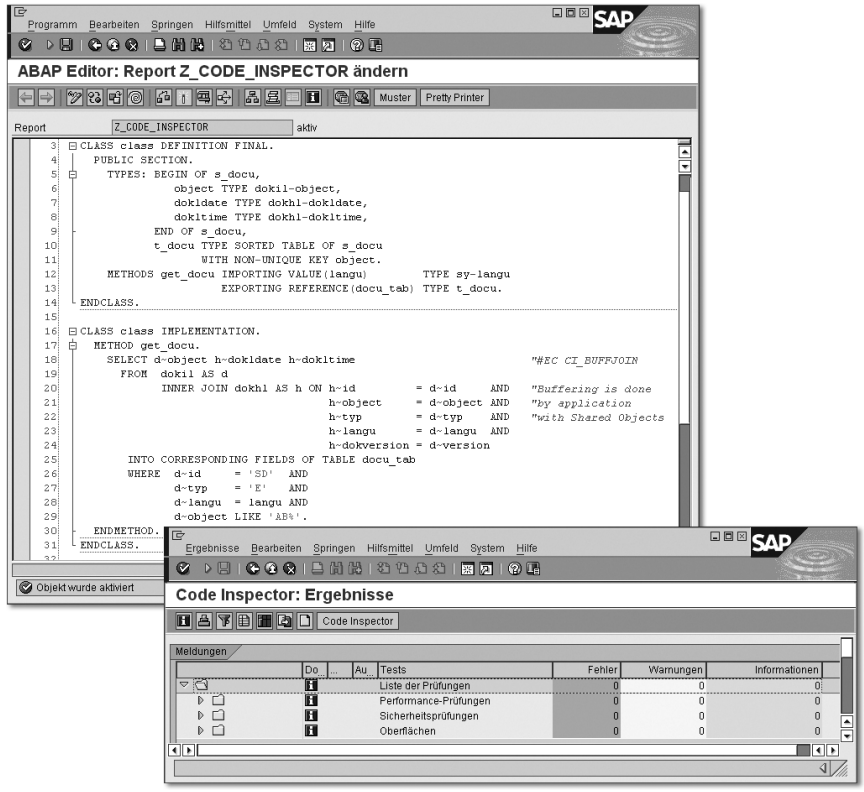


Abbildung 3.8 Code-Inspektion ohne Meldungen

Die Wertübergabe der internen Tabelle wurde durch eine Referenzübergabe ersetzt. Bei der Übergabe des elementaren Parameters `langu` wurde die Wertübergabe aus Gründen der Robustheit belassen. In der verwendeten Standardprüfung hatte sie auch keine Warnung erzeugt. Wenn der Code Inspector in einem solchen Fall eine Warnung anzeigt, kann sie mit dem Pseudokommentar `"#EC CI_VALPAR` ausgeblendet werden.

Der Inner Join der `SELECT`-Anweisung umgeht die SAP-Pufferung, was bei einem häufigen Aufruf der Methode zu Performanceproblemen führen würde. Wenn wir für das gezeigte Beispiel aber annehmen, dass die Methode Teil einer größeren Anwendung ist, in der selbst für eine Pufferung der ausgewählten Daten über Shared Objects gesorgt wird, ist die Verwendung des Inner Joins anderen weniger performanten Konstrukten, wie zum Beispiel einer

geschachtelten SELECT-Schleife, vorzuziehen. Deshalb wird die Warnung des Code Inspectors über den Pseudokommentar "#EC CI_BUFFJOIN ausgeblendet und die Gründe hierfür über einen normalen Kommentar erläutert.

3.4.4 ABAP-Testcockpit

Hintergrund

Seit den Releases 7.0 EhP2 und 7.2 ist mit dem ABAP-Testcockpit (ATC) zur SAP-internen Verwendung ein Framework in die ABAP Workbench integriert, das den entwicklungsnahen Umgang mit den notwendigen Tests erheblich erleichtert. Das ATC erlaubt die Ausführung und Ergebnisanzeige verschiedener Tests für Entwicklungsobjekte, wie beispielsweise:

- erweiterte Programmprüfungen
- statische Performanctests
- Modultests mit ABAP Unit
- statische Bedienbarkeitstests
- Paketprüfungen

Während der Code Inspector nur über die in Abschnitt 3.4.3 aufgeführte Standardprüfung in die Entwicklungsumgebung integriert ist und ansonsten nur über eine eigene Transaktion bedienbar ist, ist das ATC vollständig in den Object Navigator und den Transport Organizer integriert und steht dort für entwicklungsbegleitende Tests zur Verfügung. Qualitätsmanagern erlaubt das ATC die Durchführung von Massentests. Das ABAP-Testcockpit steht vorerst aber nur bei SAP selbst und eventuell bei SAP-Partnern für die Entwicklung von SAP-Programmen zur Verfügung.

Regel

[✓] **Regel 3.9: ABAP-Testcockpit richtig konfigurieren und verwenden**

Ist das ABAP-Testcockpit in Ihrem System verfügbar, stellen Sie vor einer Transportfreigabe sicher, dass ein ATC-Lauf über alle beteiligten Entwicklungsobjekte hinweg keine Meldungen mehr anzeigt. Hierzu sollte die ATC-Prüfung direkt in die Transportfreigabe eingebunden werden.

Details

Mit dem ATC steht erstmals ein Werkzeug zur Verfügung, das von SAP-Entwicklern und im Rahmen einer zentralen Qualitätssicherung gleichermaßen

verwendet werden kann. Überprüft ein Entwickler beispielsweise alle Entwicklungsobjekte eines Paketes im Entwicklungssystem mit der gleichen ATC-Konfiguration, wie ein Qualitätsmanager es im Rahmen eines Massenlaufs in einem Konsolidierungssystem tut, kann er alle Meldungen im Vorfeld verhindern, ohne auf Rückmeldungen vom Qualitätsmanager warten zu müssen.

Ist das ATC vorhanden und richtig konfiguriert, schließt Regel 3.9 die vorhergehenden Regeln 3.6, 3.7 und 3.8 mit ein.

Ausnahme

Das ATC steht derzeit noch nicht für Entwicklungen in Kundensystemen zur Verfügung.

Schlechtes Beispiel

Abbildung 3.9 zeigt das Ergebnis der Ausführung eines ATC-Laufs im Transport Organizer. Der geprüfte Transportauftrag enthält noch fehlerhafte Objekte.

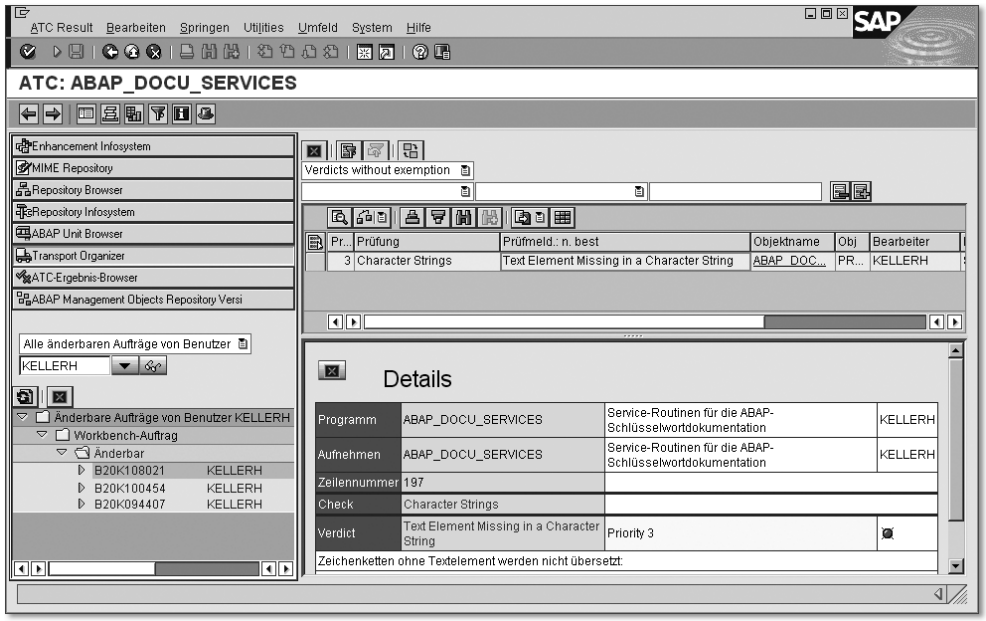


Abbildung 3.9 Warnung des ABAP-Testcockpits

Gutes Beispiel

Abbildung 3.10 zeigt das Ergebnis eines ATC-Laufs im Transport Organizer nach Behebung des Fehlers aus Abbildung 3.9. Jetzt kann der Transport freigegeben werden.

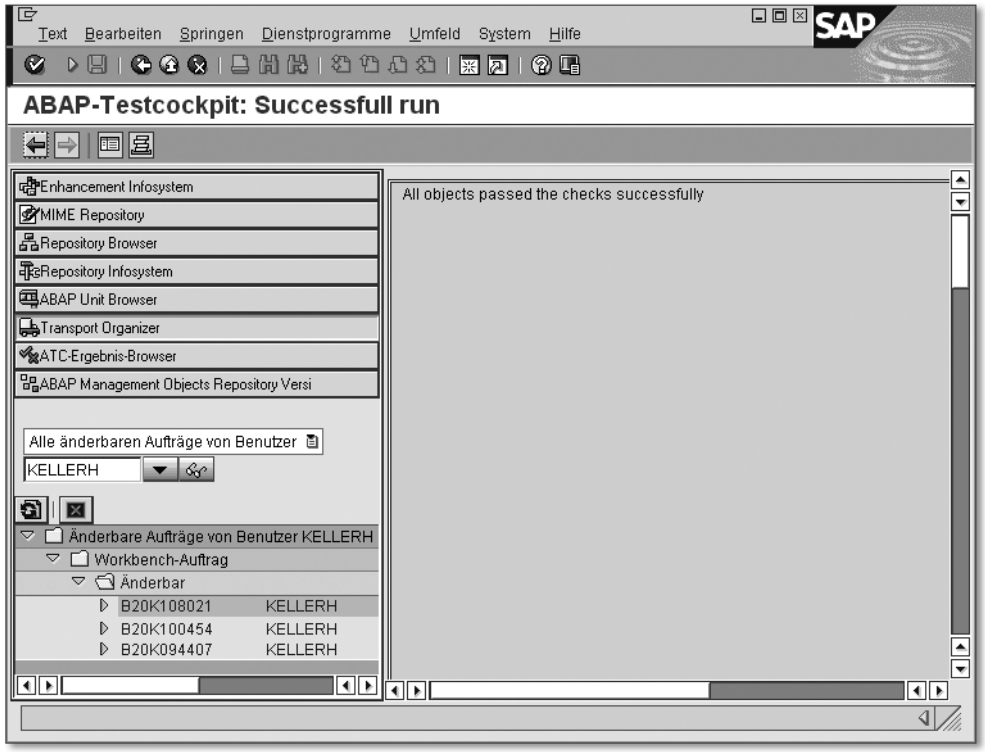


Abbildung 3.10 ABAP-Testcockpit ohne Meldungen

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z