8 Ergänzende Techniken zur Qualitätssicherung

> »Es ist das Wesen meiner Kampfkunst, dass du lernst, am Gegner vor dir auch das kleinste Anzeichen dessen wahrzunehmen, was er beabsichtigt, dass du es erkennst, solange es noch nicht ausgeführt ist.«
>
> Miyamoto Musashi

Dieses Kapitel beschäftigt sich mit weiteren Techniken, die Ihre Software noch qualitativ hochwertiger machen können. Diese Techniken unterstützen den Softwareentwickler dabei, früher Fehlerzustände zu erkennen. Die hier vorgestellten Techniken sind keine Testtechniken, sondern Methoden der Qualitätssicherung von Softwareprogrammen.

Übersicht über das Kapitel:

- Aktivierbare Checkpoints
- `ASSERTION`s und `BREAK-POINT`-Haltepunkte
- Checkpoint-Groups und deren Einsatz

Die Techniken, die wir uns jetzt anschauen wollen, gehören zu den Kontrolltechniken beim Erstellen von Software. In SAP stehen Ihnen diverse Möglichkeiten zur Verfügung, Kontrollmechanismen einzusetzen. Diese erlauben es Ihnen, stabilere Software auszuliefern, da Sie an den für Sie wichtigen Programmstellen die Kontrollmechanismen einsetzen können.

Auf den folgenden Seiten werden wir uns mit Assertions, Breakpoints und Checkpoint-Groups beschäftigen und wie Sie diese in Ihren Programmen einsetzen können.

8.1 Assertions

Assertions bezeichnen Zusicherungen in Programmen. Eine Assertion wird über die Anweisung ASSERT als bedingter Checkpoint definiert. Assertions sind entweder immer aktiv oder durch Zuordnung zu einer Checkpoint-Gruppe aktivierbar. Eine Assertion ist nichts anderes als eine Zusicherung, dass ein bestimmter Zustand, d.h. ein Inhalt einer Variablen, im Programm auch vorliegt. Die Idee dieses Konstruktes ist es, dass bestimmte Vorbedingungen (siehe ⇨ Design by Contract) eingehalten sind, bevor eine weitere Verarbeitung erfolgen soll.

The Three Amigos: Mr. Good, Mr. Bad and Mr. Pragmatic

Mr. Bad: Wieso soll ich denn ASSERT-Anweisungen benutzen? Es gibt doch die Möglichkeit, mit Ausnahmen den Programmfluss zu steuern?

Mr. Good: Die Motivation der Assertions ist es, die Robustheit von Programmen zu erhöhen, indem sie auf die Korrektheit des Zustands, also beispielsweise eines Variableninhalts, prüfen. Das Augenmerk ist hierbei darauf gerichtet, dass ein Programm so geprüft wird, wie die Anforderungen dies vorgesehen haben.

Mr. Bad: Heißt das etwa, dass die Anforderungen so geprüft werden, wie diese gestellt worden sind, und dass keine weitere Verarbeitung stattfindet, wenn der erwartete Zustand (gemäß Anforderung) nicht vorliegt?

Mr. Pragmatic: Ja, genau so ist es. Die Assertions folgen dem Designprinzip Design by Contract. Und die Anforderungen bzw. deren Spezifizierung über die Vorbedingungen werden hier als Vertrag zwischen den Objekten gesehen und beenden eine Verarbeitung, wenn der Vertrag (also Zustand) nicht eingehalten wird.

Die Anweisung ASSERT werden wir nun in unserer Beispielanwendung einsetzen. Nehmen wir an, dass die Anforderung explizit vorsieht, dass die Selektionsparameter der Reports ZAU_DISPLAY_FLIGHT_BONUS auf gar keinen Fall leer sein dürfen. Außerdem sieht die Anforderung vor, dass ggf. der Report durch andere Reports mit dem ABAP-Befehl SUBMIT aufgerufen werden soll. Der Auftraggeber wünscht sich in diesem Fall, diesen Zustand als nie eintreffend zu sehen – entsprechend robust sollte das Programm erstellt sein.

Diese Anforderung werden wir nicht mit einer Fehlernachricht überprüfen, sondern mit der ASSERT-Anweisung, um dadurch zu gewährleisten, dass das Programm nicht weiter ausgeführt wird. Die Anweisung ASSERT bewirkt eigentlich viel mehr: Ist das Ergebnis der Überprüfung zutreffend, dann wird das Programm normal fortgesetzt. Wenn das Ergebnis der Überprüfung nicht zutreffend ist, dann bricht das Programm an dieser Stelle mit dem nicht abfangbaren Laufzeitfehler ASSERTION_FAILED ab.

Gehen Sie in den Report ZAU_DISPLAY_FLIGHT_BONUS und ändern Sie das Programmcoding wie folgt ab:

Listing 8–1

```
report  zau_display_flight_bonus.
...
*---------------------------------------------------------*
* B E G I N N   D E R   V E R A R B E I T U N G
*---------------------------------------------------------*
start-of-selection.

  assert id ZAU_STARTER CONDITION:
    s_car[]   is not initial,
    s_date[]  is not INITIAL,
    s_city[]  is not INITIAL,
    s_citto[] is not INITIAL.

  zcl_flight_application=>initialize(
      it_carrid   = s_car[]
      it_fldate   = s_date[]
      it_cityfrom = s_city[]
      it_cityto   = s_citto[] ).
```

Den Report haben wir nun um die ASSERT-Anweisung ergänzt. Wir wollen mit der Anweisung überprüfen, ob die Selektionstabellen auch wirklich gefüllt sind. Wenn dieser Zustand nicht eingehalten wird, dann erfolgt an dieser Stelle der oben genannte Laufzeitfehler.

Die vereinfachte Syntax des ASSERT-Befehls sieht wie folgt aus:

```
ASSERT [ [ID group ] CONDITION ] log_exp.
```

Das Kürzel `group` enthält die Checkpoint-Gruppe und das Kürzel `log_exp` enthält den Ausdruck, der auf seinen Zustand hin geprüft wird.

Sichern Sie Ihre Änderungen und prüfen Sie die Syntax. Der Syntax-Prüfer liefert folgenden Fehler:

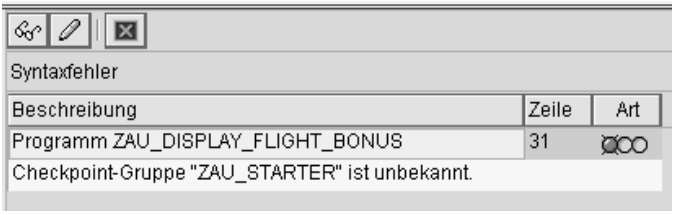

Abb. 8–1 *Syntaxfehler beim Programm ZAU_DISPLAY_FLIGHT_BONUS*

Da wir den Befehl ASSERT mit dem Zusatz ID ZAU_STARTER benutzt haben, liefert die Syntax-Prüfung einen Fehler, da wir die Checkpoint-Gruppe noch nicht angelegt haben.

Anlegen der Checkpoint-Gruppe

Eine Checkpoint-Gruppe dient zur Gruppierung und Aktivierung von ⇨ aktivierbaren Checkpoints. Das Anlegen von Checkpoint-Gruppen und die Pflege der Aktivierungseinstellungen erfolgen über die Transaktion SAAB. Die Aktivierungseinstellungen einer Checkpoint-Gruppe gelten für alle Checkpoints, die der Gruppe zugeordnet sind. Im Grunde genommen bieten Checkpoint-Gruppen nichts anderes, als Kontrollmechanismen in Programmen außerhalb des Programms zu aktivieren und deaktivieren und auch Protokollierungsmöglichkeiten zu den Kontrollpunkten zu nutzen.

Machen Sie in dem Report ZAU_DISPLAY_FLIGHT_BONUS einen Doppelklick auf die Anweisung ZAU_STARTER. Durch den Navigationsversuch erkennt das SAP-System, dass die Checkpoint-Gruppe noch nicht existiert:

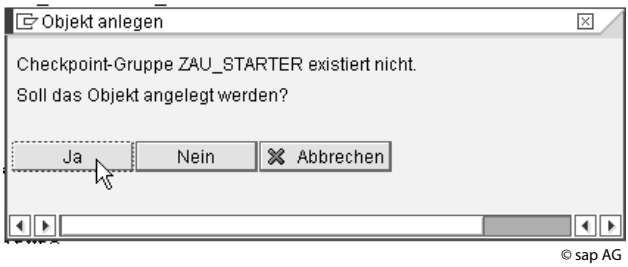

Abb. 8–2 *Checkpoint-Gruppe ZAU_STARTER anlegen*

Bestätigen Sie den Dialog mit Drücken des *Ja*-Buttons.

Geben Sie den folgenden Beschreibungstext ein und fahren Sie mit dem Drücken des *Weiter*-Buttons fort:

Abb. 8–3 *Beschreibung der Checkpoint-Gruppe hinzufügen*

In den folgenden Dialog geben Sie das Paket ZAUNIT ein und bestätigen daraufhin den bekannten Transportauftrag durch Drücken des *Weiter*-Buttons.

Letztendlich erhalten Sie einen Bestätigungsdialog, dass alles angelegt wurde:

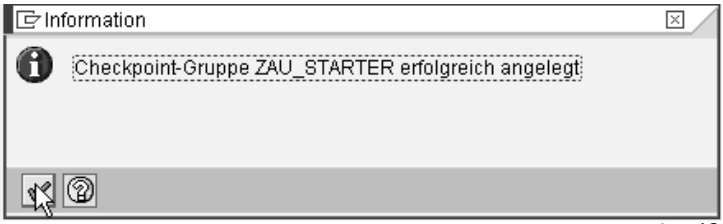

Abb. 8–4 *Informationsdialog zur Anlage der Checkpoint-Gruppe*

Bestätigen Sie auch diesen Dialog mit dem *Weiter*-Button.

Sie sind nun im *Ändern*-Modus der Checkpoint-Gruppe `ZAU_STARTER`. Per Default sind die aktivierbaren Checkpoints auf inaktiv gesetzt. Wir werden nun im Abschnitt *Assertions* die Checkbox auf *abbrechen* setzen.

Damit haben wir die Checkpoint-Gruppe `ZAU_STARTER` angelegt und die Checkpoint Assertion aktiviert. Weiterhin haben wir eingestellt, dass bei Fehlschlagen der `ASSERT`-Anweisung der Report sofort abbrechen soll. Die Einstellungen der Checkpoint-Gruppe sehen wie folgt aus:

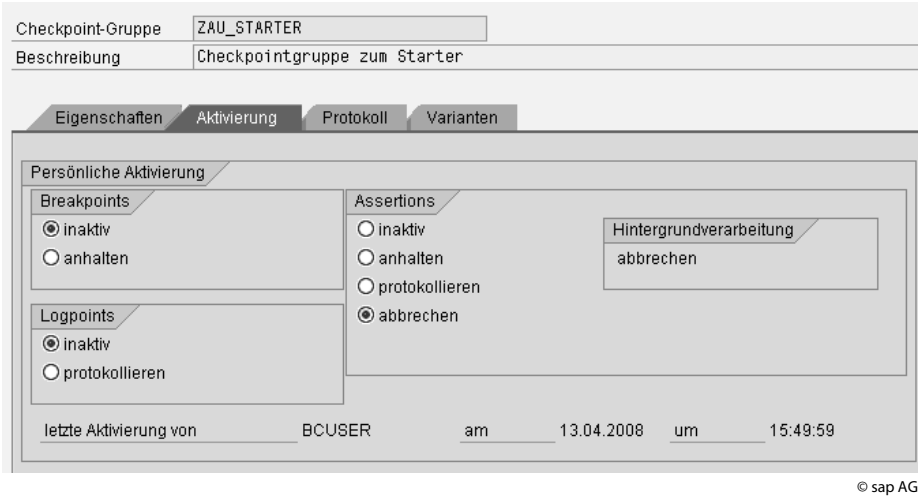

Abb. 8–5 *Einstellungsmaske der Checkpoint-Gruppe ZAU_STARTER*

Sichern Sie Ihre Änderungen und gehen Sie in den Report `ZAU_DISPLAY_FLIGHT_BONUS` zurück. Prüfen Sie erneut die Syntax und aktivieren Sie dann den Report.

Jetzt ist es an der Zeit, dass wir den Report und die darin enthaltenen Assertions prüfen. Führen Sie den Report durch Drücken des Buttons *Ausführen* aus:

Abb. 8–6 *Ausführen des Reports ZAU_DISPLAY_FLIGHT_BONUS*

Der Selektionsbildschirm des Reports wird aufgerufen. Da wir an dieser Stelle die `ASSERTION`s überprüfen möchten, werden wir nun den Report ohne Eingabe von Daten ausführen. Nach Ausführen des Reports erhalten Sie den erwarteten Laufzeitfehler:

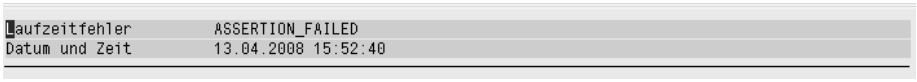

Abb. 8–7 *Laufzeitfehler des Reports ZAU_DISPLAY_FLIGHT_BONUS*

Wenn Sie zum Ende des Laufzeitfehler-Protokolls scrollen, sehen Sie auch die Abbruchstelle im Report:

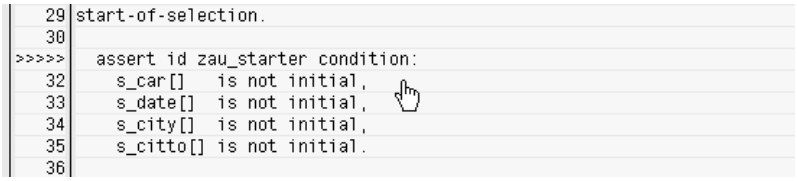

Abb. 8–8 *Abbruchstelle im Laufzeitfehler-Protokoll*

Mittels der Aktivierung und der Deaktivierung der Assertions können Sie ggf. Programme, deren Überprüfung zeitaufwändig wäre, da beispielsweise viele Zeilen an Programmcoding zu debuggen sind, einfacher analysieren. Wenn Sie eine Fehlersituation in dem Programm erwarten, dann aktivieren Sie Ihre angelegten Checkpoints, dadurch können Sie den Teil des Programms eingrenzen und Fehlerzustände eher entdecken.

8.2 BREAK-POINT

Break-Points sind Haltepunkte in einem ABAP-Programm, bei deren Erreichen in der Dialogverarbeitung in den ABAP Debugger verzweigt wird. Mit der Anweisung `BREAK-POINT` und dem Zusatz `ID` wir eine Zuordnung zur Checkpoint-Gruppe gemacht. Das bietet den Vorteil, dass die externe Aktivierung und Deaktivierung der Anweisung durch die Checkpoint-Gruppe erfolgt.

Der Einsatz der `BREAK-POINT`s findet eine große Verbreitung unter den Entwicklern. Die Anwendungen, die entwickelt werden, sind in einem großen Kontext integriert. Sie wollen den gesamten Kontext nicht debuggen, sondern an bestimmte Stellen springen, wenn Sie eine fehlerhaften Programmzustand haben. Auch Ihre eigene Anwendung kann diverse Schichten und Entwicklungskomponenten besitzen, sodass Sie gerne die Möglichkeit nutzen, nur an bestimmten Programmstellen den `BREAK-POINT` zu aktivieren.

Für die Beispielanwendung werden wir in der globalen Klasse `ZCL_FLIGHTS_BONUS` in jeder Verarbeitungsmethode einen aktivierbaren `BREAK-POINT` integrieren. Diese Checkpoint-Gruppen werden wir wie folgt organisieren:

- `ZAU_SELECT`
  Checkpoint-Gruppe zur Methode `SELECTION`
- `ZAU_REDUCE`
  Checkpoint-Gruppe zur Methode `REDUCE`
- `ZAU_MODIFY`
  Checkpoint-Gruppe zur Methode `MODIFY`
- `ZAU_UPLOAD`
  Checkpoint-Gruppe zur Methode `UPLOAD`
- `ZAU_DISPLAY`
  Checkpoint-Gruppe zur Methode `DISPLAY`

Gehen Sie nun in die Transaktion `SAAB` und legen Sie die obigen Checkpoint-Gruppen im Paket `ZAUNIT` und dem bekannten Transportauftrag an. Beachten Sie bitte bei der Anlage, dass Sie im Abschnitt *Breakpoints* die Checkbox auf *anhalten* setzen:

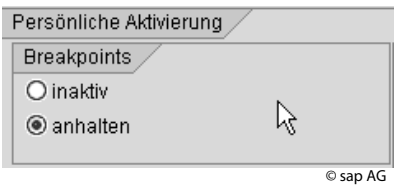

Abb. 8–9 *Breakpoint-Aktivierung in der Checkpoint-Gruppe*

Nachdem Sie alle Checkpoint-Gruppen angelegt und aktiviert haben, gehen Sie in die Klasse ZCL_FLIGHTS_BONUS, und zwar in den Methoden-Editor der Methode SELECTION. Fügen Sie die folgende Anweisung hinzu:

Listing 8–2

```
method selection.

  break-point id zau_select.
...
endmethod.
```

Gehen Sie nun in die Methoden REDUCE, MODIFY, UPLOAD und DISPLAY und fügen Sie die BREAK-POINT-Anweisung mit der zugehörigen ID hinzu.

Prüfen Sie die Syntax und aktivieren Sie die Programmänderungen.

Für die Checkpoint-Gruppen der BREAK-POINTs werden wir jetzt eine Aktivierungsvariante für Checkpoint-Gruppen anlegen. Aktivierungsvarianten dienen zum Speichern und Wiederverwenden komplexer Aktivierungen. Eine Aktivierungsvariante besteht aus einer Liste von Checkpoint-Gruppen und/oder ausführbaren Programmtypen (wie z.B. Programmen, Klassen usw.). Sie können für alle in der Aktivierungsvariante enthaltenen Objekte eine Aktivierungseinstellung speichern und diese nach Bedarf aktivieren und deaktivieren.

Wir legen die Checkpoint-Aktivierungsvariante ZAU_ALL an. Dort werden wir alle Checkpoint-Gruppen, die wir bisher erstellt haben, speichern. Gehen Sie in die Transaktion SAAB und legen Sie die globale Variante ZAU_ALL an:

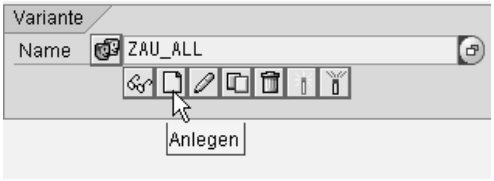

Abb. 8–10 *Checkpoint-Variante ZAU_ALL anlegen*

Geben Sie die folgende Beschreibung ein und drücken Sie den *Weiter*-Button:

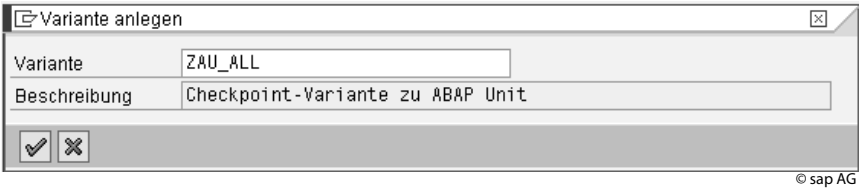

Abb. 8–11 *Beschreibung zur Checkpoint-Variante ZAU_ALL eingeben*

Tragen Sie im folgenden Dialog das Paket ZAUNIT ein und bestätigen Sie die weiteren Dialoge.

Tragen Sie die bisherigen Checkpoint-Gruppen ein und sichern Sie Ihre Einstellungen:

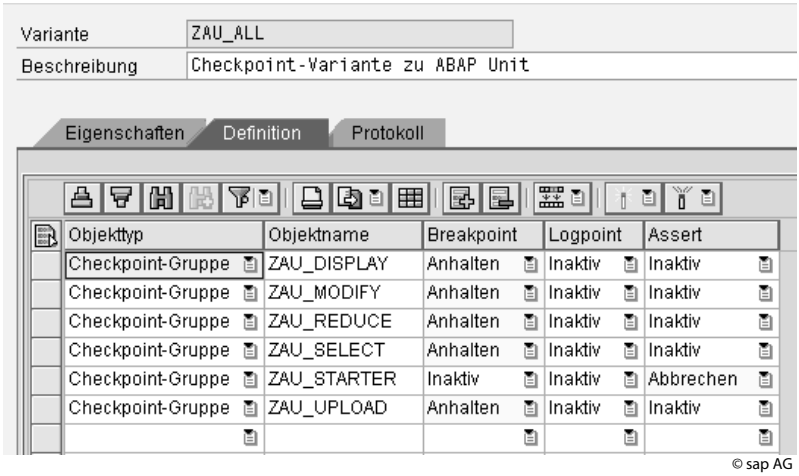

Abb. 8–12 *Checkpoint-Gruppen zur Checkpoint-Variante eingeben*

Nun können Sie über Aktivierung bzw. Deaktivierung der Varianten die gesamten Kontrollpunkte auf *Aktiv* oder *Inaktiv* setzen.

Aktivieren und Deaktivieren der Checkpoints beim Debuggen

Damit die vorgestellten Kontrollmechanismen dem Entwickler helfen, schneller die potenzielle Fehlerquelle zu finden, gibt es beim neuen Debugger die Möglichkeit, die Checkpoints zu aktivieren oder deaktivieren. Wenn Sie sich beim Debuggen befinden, wechseln Sie in den Tabreiter *Breakpoints/Watchpoints*:

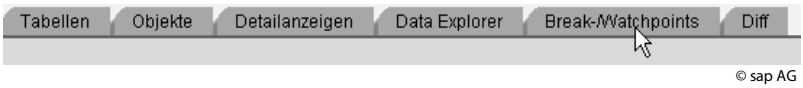

Abb. 8–13 *Zum Tabreiter Break-/Watchpoints wechseln*

Daraufhin erhalten Sie weitere Tabreiter angezeigt. Wechseln Sie in den Tabreiter *Checkpoint-Aktivierungen*:

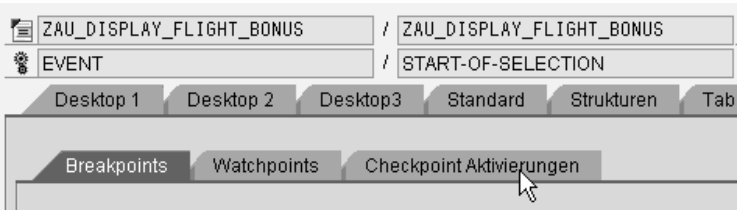

Abb. 8–14 *Tabreiter Checkpoint Aktivierungen*

Sie erhalten nun alle Checkpoint-Gruppen angezeigt, die im aktuellen Programm vorhanden sind:

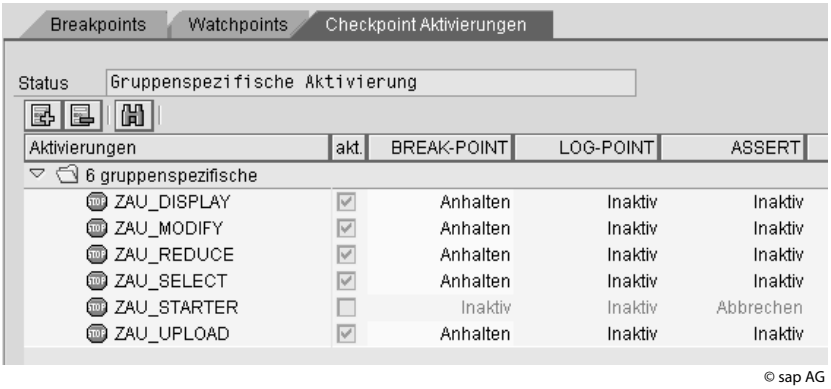

Abb. 8–15 *Übersicht der gruppenspezifischen Checkpoints*

Wenn Sie zum Eintrag `ZAU_DISPLAY` das Kontextmenü aufrufen (rechte Maustaste), können Sie die Aktivierung/Deaktivierung der Checkpoints für den aktuellen Programmlauf ändern:

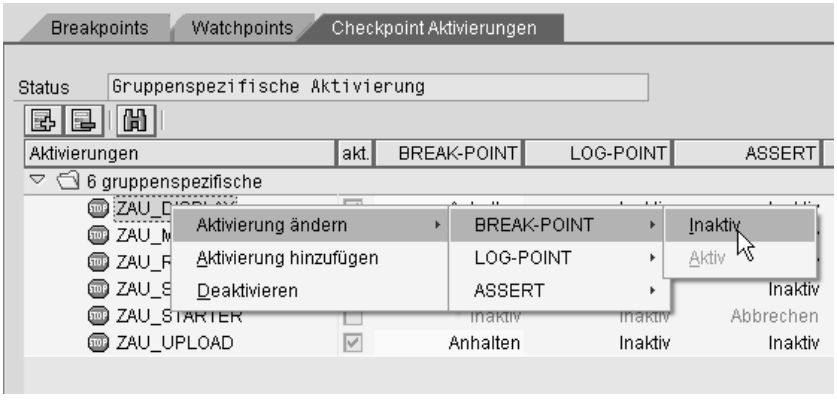

Abb. 8–16 *Checkpoint-Gruppen aktivieren*

Das Ergebnis der Änderung sehen Sie dann sofort im Debugger:

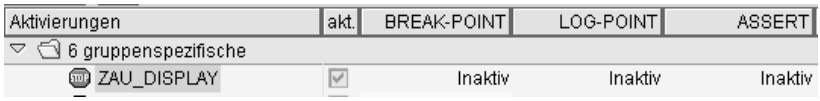

Abb. 8–17 *Ergebnis der Aktivierung*

Somit haben Sie die Möglichkeit, beim Debuggen flexibel die jeweiligen Checkpoint-Gruppen anzupassen, um eine bessere Analyse Ihres Laufzeitobjektes vorzunehmen.

8.3 Zusammenfassung

Dieses Kapitel hat Ihnen Techniken wie die aktivierbaren Checkpoints gezeigt, die im Zusammenhang mit den Checkpoint-Gruppen weitere Möglichkeiten bieten, Ihre Programme besser zu analysieren und auch qualitativ besser zu entwickeln, da Sie folgende Möglichkeiten zur Verfügung haben:

- `ASSERTIONs`, die basierend auf dem Design-by-Contract-Prinzip die Prüfung von erwarteten Zuständen ermöglichen.
- `BREAK-POINTs` sind Haltepunkte, die mittels der Checkpoint-Gruppen aktivierbar und deaktivierbar sind und somit Ihen eine Möglichkeit bieten, gezielt an den relevanten Programmstellen in den Debugging-Modus zu springen.
- Checkpoint-Gruppen ermöglichen Ihnen, die Checkpoints wie `ASSERTIONs` und `BREAK-POINTs` zu aktivieren und zu deaktivieren. Es ist eine externe Möglickeit, die Kontrollmechanismen in Ihren Programmen zu steuern.