Lektion 12: Events

Ereignisse kann man nicht ausführen – man kann Sie auslösen und dann kann man nur noch reagieren! Es handelt sich also um dynamische Prozesse, die erst zur Laufzeit in ihrem konkreten Ablauf entschieden werden dürfen.

Anforderung

Reale Ereignisse sollen sowohl mit Auslösern als auch mit reagierenden Aktionen möglichst realitätsnah programmiert werden.

Lösung

Verwenden Sie dazu das Ereigniskonzept von ABAP Objects.

Bevor wir hier an die konkrete Kodierung gehen, möchte ich Ihnen zur Motivation und zum Verständnis der grundlegenden Vorgänge und notwendigen Aktivitäten eine alltägliche reale Situation schildern, die als Grundlage für das Event-Handling in ABAP Objects dienen kann.

Szenario

Es passieren aufgrund von bestimmten Aktionen Personenunfälle (Ereignisse), deren Folgen Verletzungen (Reaktion) sind und durch Ärzte versorgt werden (Behandler). Das Eintreten der Unfälle und das Bereitstellen von behandelnden Ärzten reicht nicht aus, um die Unfallopfer zu versorgen. Warum? Weder die Unfallopfer kommen zur Ambulanz (die Unfallopfer sind i.Allg. nicht mehr handlungsfähig), noch erfahren die Ärzte, dass sie etwas zu tun haben. Die Ärzte fahren auch nicht durch die Lande, um nach Unfallopfern zu suchen.

Was fehlt?

Es fehlt in unserem Szenario die Koordination zwischen eintretendem Unfallereignis und behandlungsbereiten Ärzten. Diese Koordination leistet z.B. das Rote Kreuz: Es nimmt Unfallmeldungen entgegen, reicht diese Fälle an die behandelnden Ärzte weiter und organisiert den Krankentransport. Das heißt, durch diese Koordinationsstelle wird erreicht, dass die Opfer zu den Ärzten gelangen und dort behandelt werden können. Was benötigt das Rote Kreuz dafür? Sie brauchen eine Liste der eingetretenen Unfälle und eine Liste der behandelnden Ärzte. Die Zuordnung »welcher Patient kommt zu welchem Arzt« wird z.B. über eine tabellarische Zuordnung dieser beiden Listen vorgenommen. Es ist dabei selbstredend, dass diese Listen ständig aktualisiert werden. Sie sehen dieses Szenario in Abb. 11-8 illustriert. In ABAP Objects

wird diese Koordination durch eine interne Tabelle geleistet, die über den Sprachbefehl `SET HANDLER` gefüllt wird. Mehr dazu weiter unten.

Was passiert in diesem Szenario? – Nachdem ein Ereignisauslöser ein Ereignis ausgelöst hat, wird der Ereignis-Handler benachrichtigt. In unserem Beispiel stürzt eine Skifahrerin (= Auslöser) und bricht sich dabei das Bein. Über Telefon wird das Rote Kreuz benachrichtigt. Dieses schickt einen Ambulanzwagen, der die Verletzte zum Krankenhaus fährt. Das Rote Kreuz hat die Aufgabe, die Koordination zwischen zu behandelnden Unfallopfern und behandelnden Ärzten zu managen. Dazu benötigt es eine aktuelle Liste der Unfälle und eine aktuelle Liste der behandelnden Ärzte. Für eine effektive Abwicklung muss jetzt das Rote Kreuz noch wissen, in welche Kategorie der Unfall gehört (z.B. Beinbruch) und welche diensthabenden Ärzte für welche Kategorie zuständig sind. Auf der Unfallseite stellt sich also die Frage: Wer kommt mit welchem Problem? – Und auf Behandlerseite: Wer kann was behandeln und ist verfügbar?

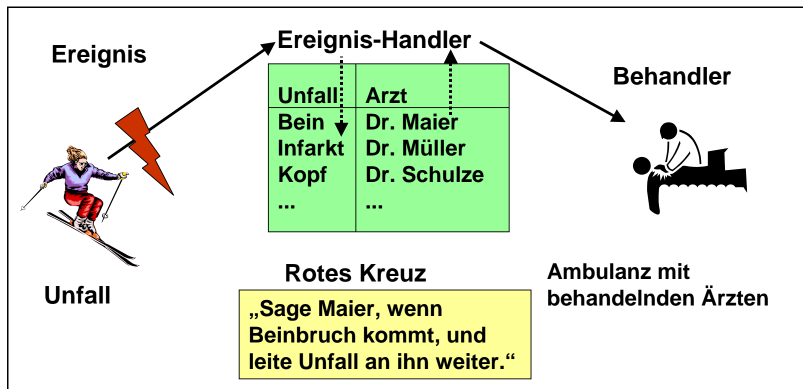


Abb. 11–8 Beispielszenario: Koordination zwischen Ereignis und Behandlung

Vom realen Szenario zum allgemeinen Vorgang

Übertragen wir dieses reale Szenario auf die Programmierumgebung, dann sind die Verletzungskategorien die verschiedenen Ereignisse, die Spezialisten in der Ambulanz die Behandlermethoden, die genau auf eine bestimmte Kategorie von Ereignis reagieren können. Auch hier die Parallele: Ein Arzt, der zwar Spezialist ist für Beinbrüche, aber entweder keinen Dienst hat oder nicht als ein solcher Spezialist beim Roten Kreuz registriert ist, kommt nicht zum Einsatz. Das heißt:

Achtung

> Ohne Registrierung passiert nichts! Das Registrieren also sorgt erst dafür, dass eine Reaktion auf ein Ereignis stattfinden kann.

In ABAP Objects ist das Konstrukt der Ereignissteuerung sehr stark und effizient ausgeprägt. Ereignisse erlauben es einem Objekt oder einer Klasse, Ereignisbehandlermethoden in anderen Objekten oder Klassen auszulösen.

> Vorteil der ereignisgesteuerten Programmierung ist vor allem, dass der Ereignisbehandler gar nicht wissen muss, von wem und wie das Ereignis ausgelöst wurde.

Ein Ereignis kann verschiedene Folgereaktionen im realen Leben auslösen. Dies ist auch bei der Programmierung möglich:

> Während beim normalen Methodenaufruf eine Methode von beliebig vielen Verwendern (Instanzen dieser Klasse) aufgerufen werden kann, können durch das Auslösen eines Ereignisses beliebig viele Ereignisbehandlermethoden aufgerufen werden.

Die Kopplung zwischen Auslöser und Behandler erfolgt erst zur Laufzeit. Beim *normalen* Methodenaufruf (d.h. dem nicht ereignisgesteuerten Aufruf) bestimmt der Aufrufer eindeutig und explizit, welche Methode er aufrufen will und die entsprechende Methode muss vorhanden sein. Bei Ereignissen hingegen bestimmt der Behandler, auf welche Ereignisse er reagieren will – und es muss nicht für jedes Ereignis eine Behandlermethode registriert sein.

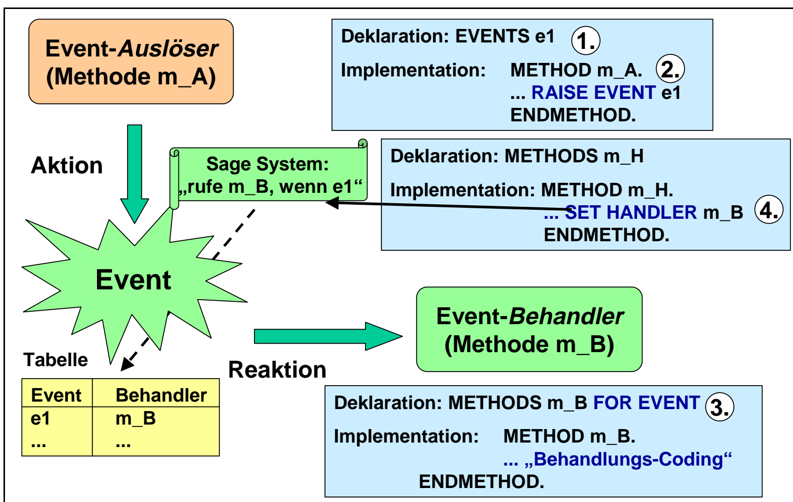


Abb. 11–9
Ereignistechnik

Übersicht

Tabelle 11-2 zeigt die fünf notwendigen Aktionen seitens der Programmierung von Events. Abb. 11-9 stellt die Zusammenhänge dar und gibt an, was wo programmiert werden muss.

Ereignisse haben eine ähnliche Parameterschnittstelle wie Methoden, jedoch keine Eingabe-, sondern nur Ausgabeparameter. Diese Parameter können vom Auslöser (Anweisung RAISE EVENT) an die Ereignisbehandlermethoden übergeben werden, die sie als Eingabeparameter übernehmen.

Die Zuordnung zwischen Auslöser und Behandler erfolgt dynamisch zur Laufzeit eines Programms mit der Anweisung SET HANDLER. Auslöser und Behandler können Objekte und Klassen sein, je nachdem ob es sich um Instanz- oder statische Ereignisse bzw. Ereignisbehandlermethoden handelt. Löst ein Auslöser ein Ereignis aus, werden in allen registrierten Behandlern die entsprechenden Ereignisbehandlermethoden ausgeführt.

Überblick

Tabelle 11-2 zeigt Ihnen im Überblick, welche Schritte in welcher Reihenfolge bei der Programmierung von Ereignissen durchzuführen sind.

Tab. 11–2 *Schritte zur Event-Programmierung: Überblick*

1. Schritt	Event e1 definieren (EVENTS: e1)
2. Schritt	Event e1 auslösen in einer Methode m_A (RAISE EVENT e1)
3. Schritt	Event e1 behandeln (Behandlermethode m_B als eine solche kennzeichnen durch FOR EVENT e1) und implementieren
4. Schritt	Zuordnen von Event e1 zum Behandler durch Registrierung in Methode m_H (SET HANDLER m_B FOR e1)
5. Schritt	Aufruf von Event-Auslöser m_A im Hauptprogramm (CALL METHOD m_A): reagieren!

Auf Ereignisse können nur die Methoden reagieren, die als solche deklariert sind (durch den Zusatz FOR EVENT: statisch) und die aktuell registriert sind (durch SET HANDLER: dynamisch).

Im Einzelnen sieht das folgendermaßen aus:

1. Schritt: *Definition eines Ereignisses*

Die Definition von Ereignissen ist analog zu der von Methoden (im Definitionsteil einer Klasse bzw. eines Interfaces):

```
CLASS <cl> DEFINITION.
  ...
  EVENTS: <e1>
     [EXPORTING VALUE(<exp_i>) TYPE [OPTIONAL] ... , ...].
     ...
ENDCLASS.
```

Wenn Sie wollen, dass der Auslöser eines Ereignisses einen Wert an die Behandlermethode übergeben soll, können Sie dies durch den EXPORTING-Zusatz bei der EVENT-Deklaration mit EVENTS erreichen. Die Übergabe kann dabei nur mit Call-by-Value erfolgen.

Statische Ereignisse können Sie mit der Anweisung CLASS-EVENTS mit analoger Syntax wie bei Instanzereignissen vereinbaren.

2. Schritt: Auslösen eines Ereignisses

Bestimmte Methoden fungieren als *Auslöser* für Ereignisse. Darauf können dann andere Methoden al*s Behandler reagieren* d.h., sie werden dann zum Ereigniszeitpunkt ausgeführt. Das Auslösen eines Ereignisses geschieht in den Methoden der gleichen Klasse mit der Anweisung RAISE EVENT. Zu einem Ereignis <e1> der Klasse <cl> können dann bei der Methodendeklaration in anderen Klassen oder in der gleichen Klasse mit dem Zusatz FOR EVENT <e1> OF <cl> Ereignisbehandlermethoden deklariert werden (durch das Deklarieren allein wird noch nichts aufgerufen!).

```
CLASS <cl> IMPLEMENTATION.
  METHOD <meth>.
    ...
    RAISE EVENT <e1>
      [EXPORTING ... <expi> = <imi> ...].
  ENDMETHOD.
  ...
ENDCLASS.
```

Hinweis

Instanzereignisse und statische Ereignisse werden ähnlich unterschieden wie Instanzmethoden und statische Methoden.

3. Schritt: Reagieren auf ein Ereignis

Anschließend können wir nun auf das Ereignis *reagieren* (d.h., wir behandeln das Ereignis durch den Aufruf einer Methode). Dies kann in Methoden beliebiger anderer Klassen erfolgen. Das Reagieren auf ein Ereignis geschieht durch das Behandeln mit einer so genannten *Ereignisbehandlermethode*. Ereignisbehandlermethoden sind spezielle Methoden, die nicht nur mit der Anweisung CALL METHOD aufgerufen werden können, sondern auch über Ereignisse auslösbar sind (in diesem Fall ist also kein CALL METHOD erforderlich). Dazu werden sie durch den Sprachzusatz FOR EVENT <e1> OF <cl> als eine Behandlermethode deklariert. Die Deklaration einer Ereignisbehandlermethode erfolgt über

```
METHODS <meth> FOR EVENT <e1> OF [<cl>|<if1>]
  IMPORTING ... <expi> = < imi> ...
```

für eine Instanzmethode und mit CLASS-METHODS entsprechend für eine statische Methode. Dabei ist <e1> ein in der Klasse <cl> bzw. im Interface <if1> deklariertes Ereignis. Zusätzlich muss diese Behandlerme-

thode zur *Laufzeit* für das Ereignis *registriert* werden (siehe 4. Schritt weiter unten).

Die Angabe FOR EVENT ermöglicht neben der normalen Nutzung über den direkten expliziten Methodenaufruf (über CALL METHOD) die zusätzliche Nutzung über den impliziten Systemaufruf bei Vorhandensein eines Ereignisses <e1> (Deklaration: EVENTS). Dabei muss das Eintreten des Ereignisses ausgelöst werden (über RAISE EVENT), und das Ereignis muss registriert (SET HANDLER) sein.

Jede Klasse kann Ereignisbehandlermethoden für Ereignisse anderer Klassen oder Interfaces enthalten. Ereignisbehandlermethoden für Ereignisse der gleichen Klasse sind natürlich auch möglich.

Für die Schnittstelle einer Ereignisbehandlermethode gelten folgende Besonderheiten:

- Die Schnittstelle darf nur aus IMPORTING-Parametern bestehen.
- Jeder IMPORTING-Parameter der Ereignisbehandlermethode muss ein EXPORTING-Parameter des Ereignisses <e1> sein (kurz: eine *surjektive* Abbildung).
- Die Eigenschaften der Parameter werden bei der Deklaration des Ereignisses <e1> mit der Anweisung EVENTS festgelegt und von der Ereignisbehandlermethode übernommen.

Die Ereignisbehandlermethode muss aber nicht alle Parameter übernehmen, die mit der Anweisung RAISE EVENT übergeben werden.

Die Deklaration einer Ereignisbehandlermethode in einer Klasse bedeutet, dass die Instanzen der Klasse bzw. die Klasse selbst, prinzipiell in der Lage sind, ein Ereignis <e1>, das in einer Methode ausgelöst wird, zu behandeln.

Erläuterungen zu Abb. 11-10

Methode m_B1 wird ausgeführt, falls Ereignis e1 eintrifft. Methode m_B2 wird nicht ausgeführt, auch wenn Ereignis e2 eintritt, denn sie ist nicht registriert worden! Methode m_B3 ist zwar registriert, da aber Ereignis e3 nicht eintritt (es wird nicht ausgelöst, da Methode m_A4 nicht entsprechend durchgeführt wird), wird auch m_B3 nicht ausgeführt.

Eine reagierende Methode kann nur ***einem*** Ereignis zugeordnet werden, aber ***ein*** Ereignis kann von ***n*** agierenden Methoden ausgelöst werden.

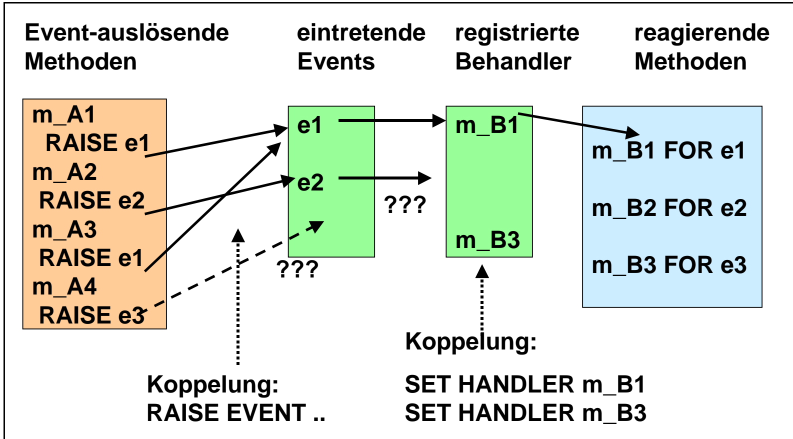


Abb. 11–10
Interne Tabelle zur Koordination zwischen Auslöser und Reagierer

Tipp

Wenn Sie wollen, dass mit einer Methode n Events ausgelöst werden können, dann »schachteln« Sie: Rufen Sie diese behandelnden Methoden innerhalb einer offiziellen Behandlermethode auf.

Die Reihenfolge, in welcher die Behandlermethoden ausgeführt werden, richtet sich nach ihrer Registrierung. Da die Registrierung der Behandlermethoden dynamisch erfolgt, sollte man sich bei der synchronen Ereignisbehandlung aber auf keine Reihenfolge verlassen, sondern von der Annahme ausgehen, dass alle Behandlermethoden gleichzeitig ausgeführt werden.

4. Schritt: Ereignisbehandlermethoden registrieren

Damit eine Ereignisbehandlermethode auf ein ausgelöstes Ereignis reagiert, d.h. zum Ereigniszeitpunkt ausgeführt wird, muss zur Laufzeit festgelegt werden, auf welche Auslöser sie reagieren soll. Dies geschieht mit der Anweisung

```
SET HANDLER... <han_i>... [FOR ALL INSTANCES|<ref>] OF <cl> ... .
```

Diese Anweisung verknüpft eine Liste von Behandlermethoden mit Auslösermethoden.

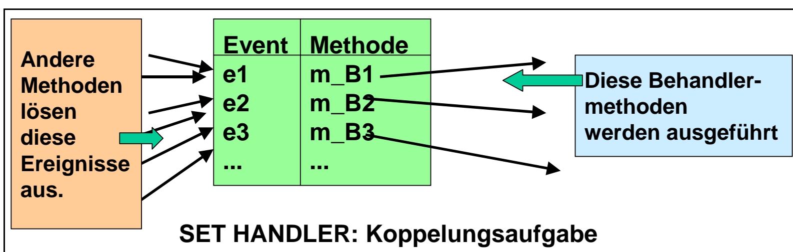


Abb. 11–11
Koppelungsaufgabe: Wirkung von SET HANDLER

Mit der Anweisung SET HANDLER wird zu jeder auslösenden Instanz für jedes Ereignis, für das eine Behandlermethode registriert wird, eine für den Verwender *unsichtbare Behandlertabelle* angelegt (siehe auch Abb. 11-11). Die Behandlertabelle enthält die Namen der Behandlermethoden und Referenzen auf die registrierten Instanzen. Die Einträge der Tabelle werden einzig und allein durch die Anweisung SET HANDLER dynamisch verwaltet.

Einfluss von SET HANDLER auf Lebensdauer

Eine Referenz auf eine Instanz in einer Behandlertabelle zählt wie eine Referenz in einer Referenzvariablen als Verwendung der Instanz und hat somit direkten Einfluss auf deren Lebensdauer. Dies bedeutet, dass auch nach einer eventuellen Initialisierung der Referenzvariablen die Instanzen nicht von der Garbage Collection erfasst werden, solange nicht ihre Registrierung in der Behandlertabelle zurückgesetzt wird.

Hinweis

Die Registrierung kann statisch oder dynamisch erfolgen. Letzteres ist nur möglich mit dem SET HANDLER-Zusatz ACTIVATION flag mit der Semantik, dass die Registrierung bei flag = 'X' gesetzt und bei SPACE zurückgenommen wird (siehe auch Onlinehilfe).

Situation

Wenn ein Ereignis ausgelöst wird, durchläuft das System die zugehörige Ereignistabelle und führt die dort angegebenen Methoden in ihren Instanzen (bzw. bei statischen Behandlermethoden in ihren Klassen) aus.

Damit die Klasse weiß, dass sie auf das Ereignis reagieren soll, wird ein Ereignis registriert (Eintrag in der so genannten *Ereignistabelle*) durch den Befehl SET HANDLER.

Da stehen ein paar Methoden zur Verfügung, quasi auf Abruf – aber sie tun nichts, da keiner da ist, der ihnen sagt, dass sie etwas tun sollen!

Die Methoden sind zwar fähig, aber träge und unselbstständig. Sie können nur reagieren, aber nicht agieren! Es agiert einzig und allein der Handler.

SET HANDLER arbeitet nach dem Prinzip »welche Methode soll bei welchem Ereignis ausgeführt werden?«. SET HANDLER ist bei dynamischer Steuerung der Ersatz für die bei statischer Steuerung vorliegende »feste Verdrahtung« zwischen vorhandener Methode und Auslöser (statisch: CALL METHOD, dynamisch: SET HANDLER + EVENTS + METHOD FOR EVENT).

Beispielszenario aus Abb. 11-12

Beim normalen Methodenaufruf (siehe Abb. 11-12, oberer Teil) sieht es so aus: Es wird explizit eine Methode im Code des Hauptprogramms aufgerufen; sie ist damit statisch vorgegeben. Im Vergleich

dazu sind die logischen Abhängigkeiten bei der ereignisgesteuerten Methodenauslösung illustriert.

Was passiert in unserem Beispiel in Abb. 11-12, unterer Teil? Nachdem alles programmiert ist, wird durch Starten des Programms die Methode `m_A` aufgerufen. Diese löst das Event `e1` aus. Da durch die Methode `m_H` (oder den Konstruktor) `SET HANDLER` ausgeführt wurde, ist `m_B` für `e1` registriert. Das heißt, jetzt wird – da `e1` eingetreten ist – `m_B` aufgerufen. `m_B` wiederum ruft intern `m0` auf (siehe auch 4. Schritt). Die wesentlichen Codeteile sehen Sie hier:

```
CLASS <cl> DEFINITION.
   ... SECTION.
       METHODS <m_H>. "oft: im Konstruktor!
       METHODS <m_B> FOR EVENT <e1> OF <cl>
                          [IMPORTING ...].
   ...
ENDCLASS.
CLASS <cl> IMPLEMENTATION.
   METHOD <m_H>.
       ...
       SET HANDLER <m_B> FOR ALL INSTANCES|<ref>.
   ENDMETHOD.
   METHOD <m_B>.
*      hier kann man nun reagieren...
       CALL METHOD m0.
   ENDMETHOD.
ENDCLASS.
```

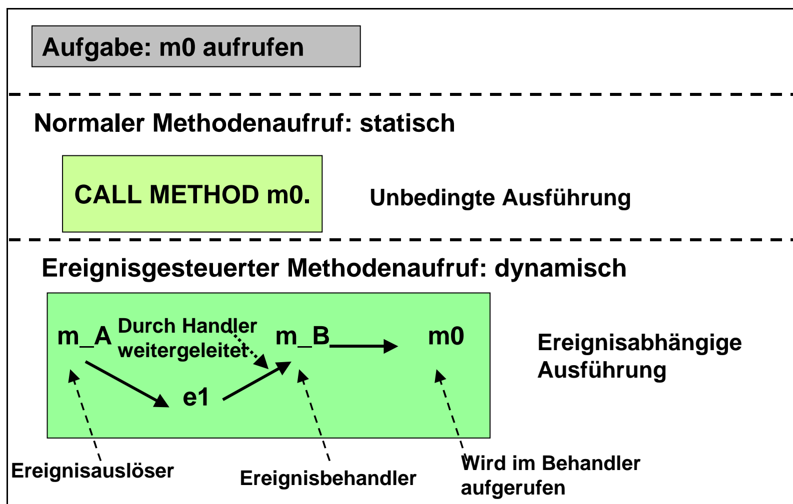


Abb. 11–12
Methodenaufrufe: statisch und ereignisgesteuert

Bei der Registrierung eines Ereignisbehandlers eines Instanzereignisses muss entweder eine betreffende Instanzreferenzvariable oder `FOR ALL INSTANCES` angegeben werden. Innerhalb der Klasse können Sie auch mit `FOR ME` Bezug nehmen auf alle Instanzen der aktuellen Klasse.

Hinweis

1. Die Registrierung von Ereignisbehandlern durch den ABAP-Befehl `SET HANDLER` kann auch außerhalb von Methoden im *normalen* ABAP-Programm erfolgen.
2. Ereignisse und Ereignisbehandler können auch in Interfaces definiert werden.

Zeitpunkt der Behandlung und Reihenfolge der Abarbeitung

Nach der Anweisung `RAISE EVENT` werden erst alle registrierten Behandlermethoden ausgeführt, bevor die nächste Anweisung bearbeitet wird (synchrone Ereignisbehandlung). Wenn eine Behandlermethode selbst Ereignisse auslöst, werden erst deren Behandlermethoden ausgeführt, bevor die Programmausführung in der Behandlermethode fortgesetzt wird. Um endlose Rekursionen zu vermeiden, ist die maximale Schachtelung von Ereignissen zurzeit auf 64 beschränkt.

Hinweis Hinsichtlich der ausgelösten Ereignisse sind vier Fälle zu unterscheiden: Das Ereignis ist ein Instanzereignis, das in einer Klasse bzw. in einem Interface deklariert ist. Beide Möglichkeiten gibt es auch noch für statische Ereignisse. Je nach Ereignis ist die Syntax und die Wirkung der Anweisung `SET HANDLER` unterschiedlich.

Aufgabe 12: Autos, die auf vollen bzw. leeren Tank selbsttätig reagieren

Folgende Aufgabenstellung ist gegeben (siehe Abb. 11-13): Ein Auto soll von A nach B fahren. Die Entfernung, der durchschnittliche Spritverbrauch sowie der aktuelle Tankinhalt ist bekannt. Es kann beliebig viel (unendlich großer Tank!) dazu getankt werden. Nach Eingabe der variablen Größen (Zielort, Entfernung) über ein Selektionsbild soll dynamisch über eventuell notwendige Tankaktionen entschieden und diese dann durchgeführt werden. Danach soll selbsttätig die Fahrt durchgeführt werden.

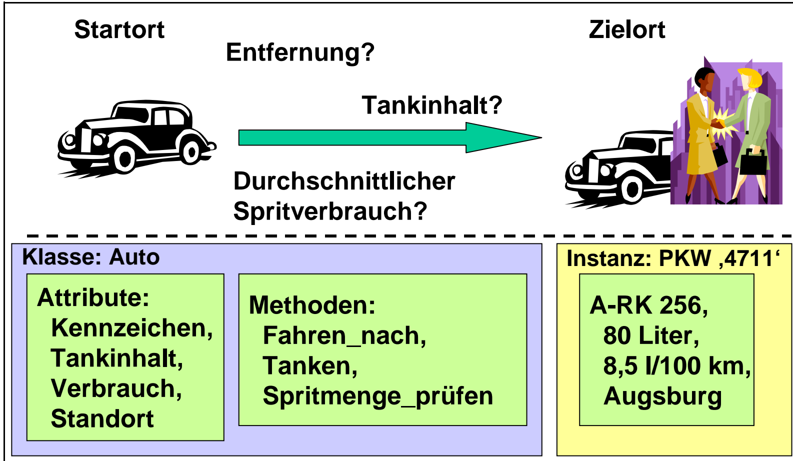


Abb. 11–13
Aufgabenstellung: Auto zum Fahren und Tanken

Lösung

Was ist hier zu tun? Dazu schauen wir uns Abb. 11-14 an. Dort sehen Sie eine Art Entscheidungsdiagramm (ereignisgesteuerte Prozessketten sehen ähnlich aus). Diese Logik wollen wir jetzt in ABAP Objects-Code umsetzen.

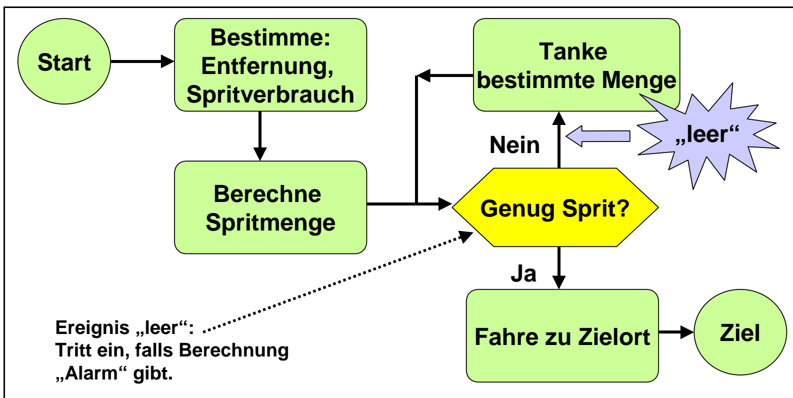
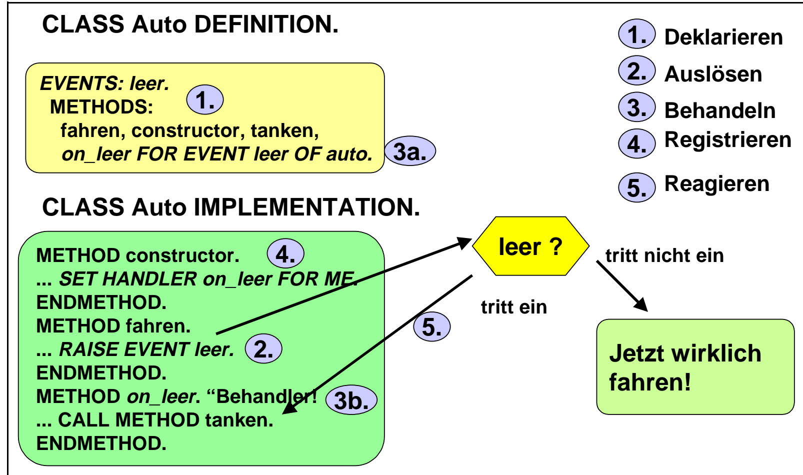


Abb. 11–14
Aufgabe zum Autofahren: Ereignisbedingungsdiagramm

Die in Abb. 11-14 illustrierte Situation formulieren wir in Abb. 11-15 nochmals etwas technischer.

Abb. 11–15
Ereignislogik für Beispiel



Wir programmieren diese Aufgabe mit Events und Interfaces. Interfaces binden wir hier gleich mit ein, um die Voraussetzungen für weiterführende Aufgaben zu schaffen. In Listing 11-3 sehen Sie das Musterprogramm.

Musterlösung

Listing 11–3
Musterprogramm für Aufgabe 12

```
REPORT YRKBSP12 .

* Globale Typen und Konstanten
TYPES: ty_boolean(1),
       ty_kennzeichen(10),
       ty_tankinhalt(4) TYPE p DECIMALS 2,
       ty_ort(30).
CONSTANTS: co_true VALUE 'X',              Deklarationsteil: global
           co_false VALUE space,
           tankmenge TYPE ty_tankinhalt VALUE '20'.

* Interface
INTERFACE lif_fahrzeug.
    EVENTS: nicht_genug_tankinhalt.        1. Ereignisdefinition
    METHODS:
      get_kennzeichen
        RETURNING VALUE(re_kennzeichen) TYPE ty_kennzeichen,
      get_ort
        RETURNING VALUE(re_ort) TYPE ty_ort,
      get_verbrauch
        RETURNING VALUE(re_verbrauch) TYPE ty_tankinhalt,
      get_tankinhalt
        RETURNING VALUE(re_tankinhalt) TYPE ty_tankinhalt,
```

```
      tanken
        IMPORTING VALUE(im_tankmenge) TYPE ty_tankinhalt,
      fahren_nach
        IMPORTING
          VALUE(im_zielort) TYPE ty_ort
          VALUE(im_entfernung) TYPE p,
      on_nicht_genug_tankinhalt
        for event nicht_genug_tankinhalt of lif_fahrzeug.
ENDINTERFACE.

* Klassen-Definition
CLASS fahrzeug DEFINITION.
  PUBLIC SECTION.
    TYPES: ty_kennzeichen(10),
           ty_tankinhalt(4) TYPE p DECIMALS 2,
           ty_ort(30).
    INTERFACES: lif_fahrzeug.
    METHODS:
      constructor
        IMPORTING
          VALUE(im_kennzeichen)
            TYPE ty_kennzeichen
          VALUE(im_verbrauch)
            TYPE ty_tankinhalt
          VALUE(im_ort)
            TYPE ty_ort
          VALUE(im_tankinhalt)
            TYPE ty_tankinhalt DEFAULT 0.
  PRIVATE SECTION.
    DATA: kennzeichen TYPE ty_kennzeichen,
          tankinhalt     TYPE ty_tankinhalt,
          verbrauch_per_100_km TYPE ty_tankinhalt,
          aktueller_ort         TYPE ty_ort.
ENDCLASS.

*
CLASS fahrzeug IMPLEMENTATION.

  METHOD constructor. "Initialisierungen
    kennzeichen = im_kennzeichen.
    verbrauch_per_100_km = im_verbrauch.
    aktueller_ort = im_ort.
    tankinhalt = im_tankinhalt.
    SET HANDLER lif_fahrzeug~on_nicht_genug_tankinhalt
      for all instances.
  ENDMETHOD.
```

3a. Ereignis-behandlermethode/ Definition

Klassen:
Definition der Schnittstelle →
Größtenteils durch Interface übernommen

Klassen:
Implementation der Methoden der Klasse

4. Registrierung

```
  METHOD lif_fahrzeug~get_kennzeichen.
    re_kennzeichen = kennzeichen.
    WRITE: / 'Mein Auto:', kennzeichen.
  ENDMETHOD.
  METHOD lif_fahrzeug~get_ort.
    re_ort = aktueller_ort.
    WRITE: / 'steht in:', aktueller_ort.
  ENDMETHOD.
  METHOD lif_fahrzeug~get_verbrauch.
    re_verbrauch = verbrauch_per_100_km.
    WRITE: / 'Verbrauch:', verbrauch_per_100_km,
             'Liter pro 100 km'.
  ENDMETHOD.

  METHOD lif_fahrzeug~get_tankinhalt.
    re_tankinhalt = tankinhalt.
    WRITE: / 'aktueller Tankinhalt:', tankinhalt.
  ENDMETHOD.

  METHOD lif_fahrzeug~tanken.
*   im Moment ist der Tank noch unendlich groß
    ADD im_tankmenge TO tankinhalt.
  ENDMETHOD.

  METHOD lif_fahrzeug~fahren_nach.
    DATA: estim_verbrauch TYPE ty_tankinhalt.
    estim_verbrauch =
      im_entfernung * verbrauch_per_100_km / 100.
    WHILE estim_verbrauch > tankinhalt.
      WRITE:  ' .. Sprit reicht nicht -->'.
      RAISE EVENT lif_fahrzeug~nicht_genug_tankinhalt.
    ENDWHILE.
      SUBTRACT estim_verbrauch FROM tankinhalt.
      aktueller_ort = im_zielort.
  ENDMETHOD.
  METHOD lif_fahrzeug~on_nicht_genug_tankinhalt.
    CALL METHOD lif_fahrzeug~tanken( tankmenge ).
    WRITE: / kennzeichen, 'hat selbsttätig',
             tankmenge, 'Liter getankt.'.
  ENDMETHOD.
ENDCLASS.
```

Klassen: Implementation der Methoden des Interfaces

2. Ereignis auslösen

3b. Ereignis-Behandler-methode/ Implementation

```
* Hauptprogramm
DATA: mein_auto TYPE REF TO fahrzeug,
      mein_auto_lif TYPE ref to lif_fahrzeug,
      sprit_reicht TYPE ty_boolean,
            inh TYPE ty_tankinhalt.
PARAMETERS: nr TYPE ty_kennzeichen default 'A-RK 256',
            verbr TYPE ty_tankinhalt default '8.5',
            ort TYPE ty_ort default 'Augsburg',
            ziel TYPE ty_ort default 'Rom',
            strecke TYPE p default '1500'.

* Start der Datenselektion
START-OF-SELECTION.
* mein_auto erzeugen
  CREATE OBJECT mein_auto
         EXPORTING im_kennzeichen = nr
                   im_verbrauch = verbr
                   im_ort = ort.
  mein_auto_lif = mein_auto.
* Ausgabe der privaten Attribute nach Erzeugung
  CALL METHOD mein_auto_lif->get_kennzeichen
              RECEIVING re_kennzeichen = nr.
  CALL METHOD mein_auto_lif->get_ort
              RECEIVING re_ort = ort.
  CALL METHOD mein_auto_lif->get_verbrauch
              RECEIVING re_verbrauch = verbr.
  CALL METHOD mein_auto_lif->get_tankinhalt
              RECEIVING re_tankinhalt = inh.
* Tanken, falls notwendig: solange, bis es reicht!
   WRITE: / nr, 'soll nach', (21) Ziel, 'fahren '.
   CALL METHOD mein_auto_lif->fahren_nach
           EXPORTING im_zielort = ziel
                     im_entfernung = strecke.
  WRITE: /5 '==> jetzt reicht die Füllung
                   für die gesamte Strecke!'.
  WRITE: /5 '    Fahrt geht los..'.
* Ausgabe der privaten Attribute nach Zielankunft
  skip.
  CALL METHOD mein_auto_lif->get_kennzeichen
    RECEIVING re_kennzeichen = nr.
  CALL METHOD mein_auto_lif->get_ort
    RECEIVING re_ort = ort.
  CALL METHOD mein_auto_lif->get_tankinhalt
    RECEIVING re_tankinhalt = inh.
```

Datendeklaration: Hauptprogramm

Hauptprogramm:
Nutzung der Klassen, Arbeiten mit Objekten

5. Aufruf der
Ereignisauslösermethode

In Abb. 11-16 sehen Sie das mit obigem Programm erzeugte Selektionsbild und die Ergebnisliste mit den ereignisgesteuerten Aktionen.

Abb. 11–16
Ereignisgesteuertes Tanken, Ein- und Ausgabe: Musterlösung zu Aufgabe 12

```
NR        A-RK 256
VERBR         8,50
ORT       Augsburg
ZIEL      Rom
STRECKE              1.500
```

```
Lektion 12

Mein Auto: A-RK 256
steht in: AUGSBURG
Verbrauch:      8,50  Liter pro 100 km
aktueller Tankinhalt:     0,00
A-RK 256   soll nach ROM                    fahren   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.   .. Sprit reicht nicht —>
A-RK 256   hat selbsttätig    20,00  Liter getankt.
    ==> jetzt reicht die Füllung für die gesamte Strecke!
        Fahrt geht los..

Mein Auto: A-RK 256
steht in: ROM
aktueller Tankinhalt:    12,50
```

© SAP AG