Um einheitliche Objekte für eine Anwendung zu erzeugen, können Sie standardisierte Entwurfsmuster einsetzen. Dazu steht Ihnen die Gruppe der Erzeugungsmuster zur Verfügung, deren Entwurfsmuster unterschiedliche klassenbasierte oder objektbasierte Objekte erzeugen.

3 Erzeugungsmuster

Alle Entwurfsmuster, die in die Kategorie der Erzeugungsmuster eingeordnet sind, dienen dazu, Objekte zu erstellen. Die für die Objekterzeugung notwendigen Details werden bei den Entwurfsmustern dieser Kategorie in Klassen gekapselt. Der Vorteil dieses Verfahrens ist, dass die Implementierung der Objekterzeugung komplett getrennt von den zu erstellenden Objekten erfolgt. Das heißt, es müssen keine Änderungen am Objekt selbst vorgenommen werden, wenn ein neues Objekt erstellt wird.

Klassen- und Objektmuster

Wie bereits in Abschnitt 1.1.2, »Softwaredesign mithilfe von Entwurfsmustern«, erläutert, unterscheiden wir bei Entwurfsmustern grundsätzlich zwischen zwei Varianten:

- Einerseits gibt es objektbasierte Entwurfsmuster, die durch objektbasierte Programmierung umgesetzt werden. Durch die Verwendung von ABAP-Objects-Schlüsselwörtern (z. B. `CREATE OBJECT` oder ab SAP NetWeaver 7.4 das Schlüsselwort `NEW`) werden dabei Instanzen der Klassen erzeugt.
- Die zweite Variante sind Entwurfsmuster, die auf Klassenmustern basieren. Diese werden mithilfe statischer Methoden umgesetzt.

Klassen- und objektbasierte Erzeugungsmuster

Im Folgenden ordnen wir die in diesem Kapitel vorgestellten Muster der Gruppe der Erzeugungsmuster diesen beiden Kategorien zu:

- **Klassenmuster**
  - *Factory Pattern*
    Wie alle Erzeugungsmuster erzeugt das Factory Pattern neue Objekte. Diese Objekte werden über Klassen anhand von

Importparametern definiert. Bei diesen Klassen handelt es sich jeweils um Ableitungen einer abstrakten Klasse. Das Entwurfsmuster setzt dabei das Polymorphismus-Prinzip um.

- *Abstract Factory Pattern*
  Das Abstract Factory Pattern basiert auf dem Factory Pattern. Im Falle des Abstract Factory Patterns ist die Factory-Klasse selbst eine abstrakte Klasse. Dies hat den Vorteil, dass die Factory-Klasse noch flexibler und austauschbarer ist. Das Abstract Factory Pattern erzeugt keine konkreten Objekte, sondern konkrete Factory-Klassen. Dies bedeutet, eine abstrakte Factory-Klasse ist eine Factory für die konkreten Factory-Klassen. Durch dieses Konzept lassen sich jederzeit neue Factory-Klassen integrieren oder bestehende durch andere Factory-Klassen ersetzen.

▸ **Objektmuster**

- *Builder Pattern*
  Beim Builder Pattern wird die Erzeugung komplexer Objekte von der eigentlichen Repräsentation getrennt. Dadurch können unterschiedliche Repräsentationen der Objekte erzeugt werden. Diese sind immer abhängig von den Anforderungen der jeweiligen Klassen bzw. von der Erzeugung. Dieses Entwurfsmuster arbeitet mit unterschiedlichen Konstruktoren, die oft in Verbindung mit Kompositum-Entwurfsmustern verwendet werden. Das Kompositum-Entwurfsmuster wird genutzt, um Hierarchien aus Objekten aufzubauen, die zusammen als Teile eines Ganzen dienen. Dabei werden die Objekte zu Baumstrukturen zusammengesetzt.
- *Prototype Pattern*
  Das Prototype Pattern wird zunächst mit einer vorläufigen Instanz eines komplexen Objekts initialisiert. Im weiteren Verlauf wird immer das ursprüngliche Objekt des Prototyps geklont und kann unabhängig vom eigentlichen Objekt verwendet werden.
- *Singleton Pattern*
  Das Singleton Pattern wird mithilfe einer Klasse realisiert. Von dieser Klasse kann nur ein Objekt instanziiert werden. Greifen Sie anschließend auf das Entwurfsmuster zu, erhalten Sie immer das ursprünglich instanziierte Objekt. Da stets mit Referenzen auf diese einmalige Instanz gearbeitet wird, sind alle

Änderungen an dem Objekt für alle Aufrufer gültig. Dieses Pattern wird daher für den globalen Zugriff auf Instanzen verwendet.

Aufbau dieses Kapitels

In den folgenden Abschnitten erläutern wir die einzelnen Erzeugungsmuster näher. Um Ihnen einen einfachen Überblick über die Entwurfsmuster zu geben und die Vergleichbarkeit zu gewährleisten, sind die Abschnitte dabei jeweils nach der folgenden Struktur gegliedert:

1. **Problem**
   Zu jedem Entwurfsmuster beschreiben wir zunächst die Situation, in der es angewendet werden kann.
2. **Ansatz und Lösung**
   Im zweiten Schritt beschreiben wir den Lösungsansatz des jeweiligen Entwurfsmusters. Wir gehen anhand eines abstrakten Beispiels darauf ein, wie das Entwurfsmuster und dessen Komponenten aufgebaut sind und wie diese in Beziehung zu anderen Klassen stehen. Ziel dieser Beschreibungen ist es, die Muster in der Praxis anwenden zu können.
3. **Einsatzbeispiele**
   Im nächsten Schritt führen wir mögliche Einsatzgebiete für die einzelnen Entwurfsmuster an.
4. **Umsetzung in ABAP**
   In den Abschnitten »Umsetzung in ABAP« zeigen wir Ihnen, wie die Entwurfsmuster in ABAP realisiert werden können. Wir erläutern dabei jeweils, wie das Entwurfsmuster aufgebaut wird und welche Funktionen die einzelnen Methoden haben. Die Implementierung der einzelnen Entwurfsmuster erfolgt jeweils in ABAP-Klassen, die wiederum verschiedene Unterklassen besitzen. Aus Gründen der Übersichtlichkeit erstellen wir in unseren Beispielen alle Klassen über die Transaktion SE24, den Class Builder.
5. **Evaluation**
   Abschließend stellen wir jeweils die Vor- und Nachteile des besprochenen Entwurfsansatzes einander gegenüber. Dadurch möchten wir Ihnen die Entscheidung erleichtern, welches Entwurfsmuster Sie am besten in Ihren Anwendungen und für Ihren konkreten Zweck verwenden können. Viele Entwurfsmuster weisen problematische Seiteneffekte auf, die aber durch eine Kombination mehrerer Entwurfsmuster minimiert werden können.

3.1 Builder Pattern

Das *Builder Pattern* wird im Deutschen auch als *Erbauermuster* bezeichnet und gehört zu den GoF-Mustern, also jenen Entwurfsmustern, die ursprünglich von der Gang of Four beschrieben wurden (siehe auch Abschnitt 1.1, »Was sind Entwurfsmuster?«). Das Builder Pattern trennt den Erzeugungsprozess eines komplexen Objekts von der eigentlichen Verwendung dieses Objekts mithilfe von Klassenmethoden. Dadurch ist es möglich, den Erzeugungsprozess, der in diesem Zusammenhang auch oft *Konstruktionsprozess* genannt wird, unabhängig von der Repräsentation anzuwenden.

3.1.1 Problem

Beispiel: verschiedene Exportwege

Das Builder Pattern wird verwendet, um den Erzeugungsprozess komplexer Objekte von der Repräsentation dieser Objekte zu entkoppeln. Wir verdeutlichen dies anhand eines Beispiels. Sie können dieses Entwurfsmuster z. B. für eine Anwendung verwenden, die Daten aus einer Datenbanktabelle liest, diese Daten überarbeitet und dann wieder in die Datenbank zurückschreibt. Möchten Sie dieser Anwendung eine Exportschnittstelle hinzufügen, die Exportmöglichkeiten in andere Anwendungen anbietet, können Sie unterschiedliche Builder definieren, die vom Anwender über die Anwendungsoberfläche ausgewählt werden können, und sie jeweils in anderen Datenformaten speichern. Ein solcher Builder erzeugt die Objekte, die dann die Daten als CSV-Datei exportieren, ein weiterer Builder die Objekte, die Daten als XML-Dateien exportieren, und ein letzter Builder könnte die Daten über SAP Smart Forms als PDF aufbereiten und exportieren. Auf diese Weise wird die eigentliche Verarbeitung der Daten vom Exportprozess getrennt.

3.1.2 Ansatz und Lösung

Mithilfe eines UML-Diagramms zeigen wir Ihnen zunächst, wie das Builder Pattern aufgebaut ist. In Abbildung 3.1 sehen Sie fünf ABAP-Klassen, die zusammen das Builder Pattern bilden.

Bedeutung der einzelnen Klassen

Diese Klassen nehmen jeweils eine bestimmte Rolle im Entwurfsmuster ein, die wir in Tabelle 3.1 genauer erläutern.

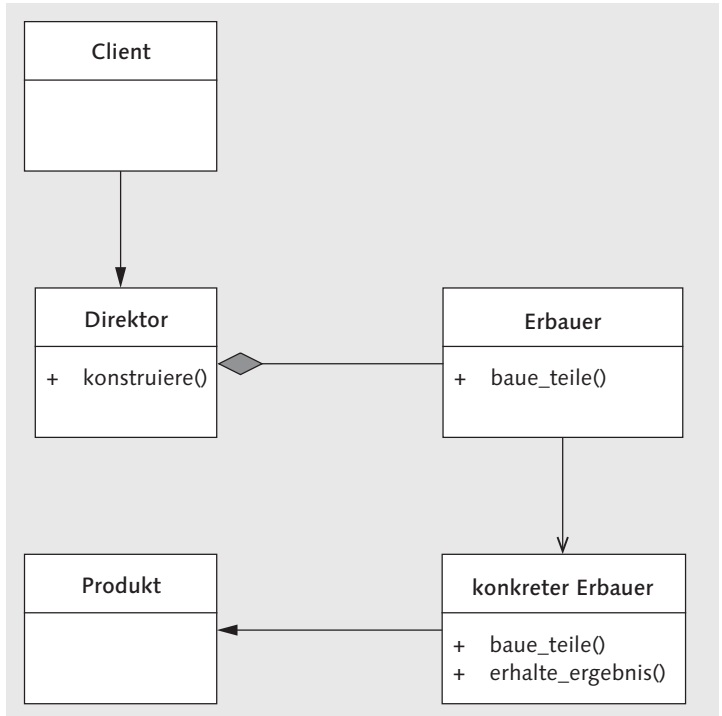

Abbildung 3.1 Builder Pattern – Aufbau im UML-Klassendiagramm

Klasse und ihre Rolle	Beschreibung
Erbauer	Die Erbauerklasse ist eine abstrakte Klassenschnittstelle, die für die Erzeugung der komplexen Objekte und auch für die Erzeugung der Bauteile zuständig ist.
konkreter Erbauer	Die Klasse des konkreten Erbauers kümmert sich darum, dass die einzelnen Bauteile während der Implementierung des Produkts erzeugt werden. Die Erbauerklasse bzw. -schnittstelle gibt dem konkreten Erbauer den Rahmen sowie die Definition vor, wie die Klasse des konkreten Erbauers durch die Repräsentationsschicht implementiert werden kann. Über die Erbauerschnittstelle kann das Objekt an das Produkt weitergegeben werden.

Tabelle 3.1 Builder Entwurfsmuster – beteiligte Klassen

Klasse und ihre Rolle	Beschreibung
Direktor	Der Direktor erzeugt die komplexen Objekte mithilfe der Erbauerschnittstelle. Hier können unterschiedliche Varianten hinterlegt werden, die definieren, welche Arten von Produkten erzeugt werden können und wie diese zusammengebaut werden müssen. Beispiele sind ein Auslagerungsauftrag für das Lager, ein Verschrottungsauftrag oder ein Auftrag für die Ausbuchung von Material als Varianten des Objekts AUFTRAG.
Produkt	Als Produkt wird das zu konstruierende komplexe Objekt bezeichnet. Dieses Produkt basiert auf Klassen, die zur Beschreibung der einzelnen Bauteile und Produkte erforderlich sind. Ein konkretes Beispiel wäre ein Auftrag z. B. für die Versendung o. Ä.
Client	Der Client ist der Endanwender oder eine Systemaktion, die z. B. über die Oberfläche einer Web-Dynpro-Anwendung eine Aktion auslöst, z. B. durch den Klick auf einen Button. Der Client definiert so die Rahmenbedingungen für die Produkterzeugung und gibt diese an den Direktor weiter.

Tabelle 3.1 Builder Entwurfsmuster – beteiligte Klassen (Forts.)

Sequenzdiagramm

Das Sequenzdiagramm des Builder Patterns in Abbildung 3.2 können Sie wie folgt lesen: Der Client erzeugt zu Beginn ein Objekt vom Typ der Klasse für den konkreten Erbauer. Anschließend erzeugt der Client ein Objekt vom Typ Direktor. Nun ist die Grundlage für das Builder Pattern geschaffen, und der Client kann eine der zentralen Methoden aus dem Direktor-Objekt aufrufen. Dabei werden die Methoden der Erbauerschnittstelle aufgerufen, die wiederum unterschiedliche Bauteile für das Produkt über den konkreten Erbauer erzeugen lassen. Dies kann ein Bauteil sein, es können aber auch mehrere Bauteile in Kombination sein. Als Ergebnis erhält der Client das konstruierte standardisierte Objekt vom konkreten Erbauer zurück.

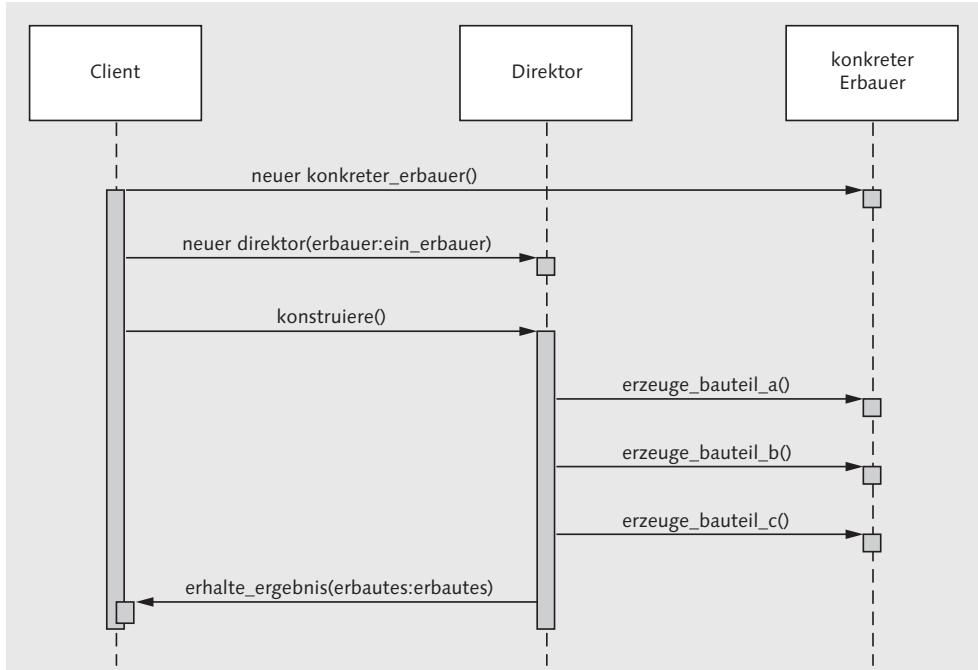

Abbildung 3.2 Builder Pattern – UML-Sequenzdiagramm

3.1.3 Einsatzbeispiel

Export aus Web-Dynpro-Anwendung

Um Ihnen die Einsatzgebiete des Builder Patterns zu verdeutlichen, erläutern wir seine Verwendung exemplarisch anhand eines Datenexports aus einer Anwendung in Web Dynpro ABAP. Dabei hat der Anwender verschiedene Möglichkeiten, einen Auftrag, den er aufgerufen hat, zu exportieren. Der Auftrag kann nicht direkt als PDF exportiert werden, sondern muss vor dem Export bearbeitet werden, um sicherzustellen, dass z. B. die Druckausgabe fehlerfrei funktioniert und alle Zeichen auf dem Formular korrekt und lückenlos dargestellt werden.

Klassendiagramm

Abbildung 3.3 veranschaulicht dieses Einsatzbeispiel anhand eines UML-Diagramms. Die Klasse, die den Direktor repräsentiert, wird in diesem Fall von der Web-Dynpro-Oberfläche über eine Aktion aufgerufen und stößt dadurch die Erzeugung des PDF-Exports (Objekt der konkreten Erbauerklasse) an. Auf dieser Web-Dynpro-Oberfläche gibt es in unserem Beispiel einen Button, der die Methode `BUTTON_AUFTRAG_EXPORT_TO_PDF()` auslöst. Klickt der Anwender auf diesen Button, werden ihm unterschiedliche Exportmöglichkeiten

zur Auswahl gestellt. Die konkrete Klasse `ERBAUER_EXPORT_1` enthält alle Methoden, die zur Verfügung stehen, um das Auftragsdokument zu überarbeiten und mit weiteren Informationen anzureichern. In der Methode `UEBERPRUEFE_EINGABEN()` wird z. B. geprüft, ob diese Daten korrekt sind. An dieser Stelle kann geprüft werden, ob der Kunde, für den der Auftrag angelegt wurde, bereits im System vorhanden ist, ob für ihn eine Bonitätsprüfung vorliegt und vieles mehr.

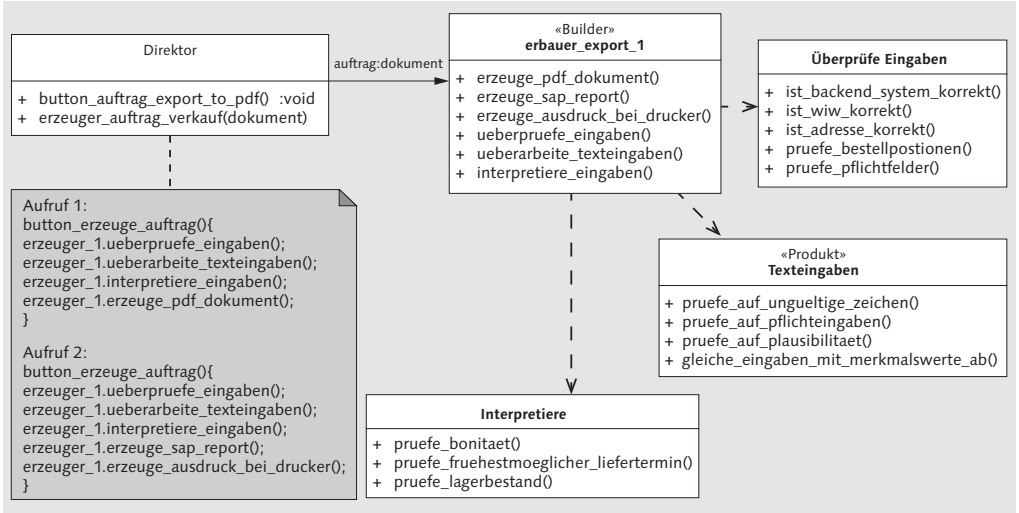

Abbildung 3.3 Builder Pattern – UML-Klassendiagramm für Beispielanwendung

Unterklassen

Die Methoden, die von der Erbauerklasse angesprochen werden, beinhalten einen standardisierten Aufruf der Unterklassenmethoden `INTERPRETIERE()`, `TEXTEINGABEN()` und `UEBERPRUEFE_EINGABEN()`. Die Unterklassen sind so realisiert worden, dass jede dieser Klassen andere Methoden aufrufen kann. Sie sind als ein Pool von Funktionen zu verstehen, die Sie verwenden können. Dazu werden jeweils nur die Informationen benötigt, die die Methoden selbst anfordern. Die Erzeugungsmethoden der Builder-Klasse rufen mit den Unterklassen mehrere dieser Funktionen auf, um die Texte und Eingaben im Auftragsformular zu überprüfen. Sie können an dieser Stelle alle komplexen Methoden programmieren, die für Ihren Einsatzzweck notwendig sind.

[zB] Aufruf mehrerer Funktionen

Mithilfe des Builder Patterns können Sie z. B. einen Methodenaufruf implementieren, der die erforderlichen Buchungen im Buchungssystem

durchführt, den Kunden darüber benachrichtigt, dass sein Auftrag in Bearbeitung ist, und zusätzlich noch einen weiteren Workflow startet. Dies kann sehr einfach über den Methodenpool der Subklassen realisiert werden. In dem Subsystem werden dabei z. B. zentrale Klassen des SAP-Systems oder Eigenentwicklungen aufgerufen, die die gewünschten Aktionen im System durchführen.

3.1.4 Umsetzung in ABAP

Report

In diesem Abschnitt zeigen wir Ihnen, wie Sie das Builder Pattern in ABAP umsetzen können. Wir greifen dazu auf das Beispiel aus Abbildung 3.2 zurück. Zur Implementierung des Builder Patterns erstellen wir einen ABAP-Report in Transaktion SE38. In diesem Report wird das Pattern umgesetzt und kann hier später auch ausgeführt werden. Wird der Report später in eine Anwendung integriert, findet dies meistens in einer Verarbeitungsklasse statt, die wiederum ihre eigenen Klassen besitzt. Wir verzichten in diesem Kapitel darauf, alle diese Klassen in Transaktion SE24 anzulegen, um das Beispiel für Sie übersichtlicher zu gestalten.

Durch das Erzeugen des ABAP-Reports öffnet sich der ABAP Editor mit der folgenden Codezeile:

```
REPORT zdp_ar_erz_builder.
```

Produktklasse

Listing 3.1 zeigt die Definition der Klasse, in der das Produkt definiert wird. Diese Produktklasse beinhaltet zwei Methoden und ein internes Attribut. Das interne Attribut hat den Datentyp `STRING` und enthält den Namen des Bauteils. Es können noch weitere Informationen und Objekte in dieser Klasse abgelegt werden. Die Methoden fügen die neuen Bauteile in das Attribut `I_BAUTEILE` der Klasse `ZDP_CL_BUILD_PRODUKT` hinzu. Über die Methode `ANZEIGEN()` können die Bauteile ausgegeben werden, die im Produkt erzeugt wurden. Die Methode `HINZUFUEGEN()` fügt der Bauteilliste neue Bauteile hinzu.

```
CLASS zdp_cl_build_produkt DEFINITION
  PUBLIC  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
    DATA bauteile TYPE TABLE OF  string.
    METHODS hinzufuegen
      IMPORTING
        !i_bauteile TYPE string.
```

```
    METHODS anzeigen.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 3.1 Builder Pattern – Definition der Produktklasse

Nach der Definition der Klasse muss diese noch implementiert werden. Die Implementierung sehen Sie in Listing 3.2.

```
CLASS zdp_cl_build_produkt IMPLEMENTATION.
  METHOD hinzufuegen.
    APPEND i_bauteile TO bauteile.
  ENDMETHOD.
METHOD anzeigen.
    DATA: ls_ausgabezeile TYPE string.
    WRITE:/ 'Produkt -- Welche Bauteile sind vorhanden? --'.
    LOOP AT bauteile INTO ls_ausgabezeile.
      WRITE:/ 'Objekt: ' && ls_ausgabezeile.
    ENDLOOP.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.2 Builder Pattern – Implementierung der Produktklasse

Erbauerklasse

Im nächsten Abschnitt, den Sie in Listing 3.3 sehen, definieren wir die Klasse `ZDP_CL_BUILD_ERBAUER`. Die abstrakte Klasse `ZDP_CL_BUILD_ERBAUER` definiert eine Hülle für die eigentlichen Erbauerklassen `ZDP_CL_BUILD_KONERB1` und `ZDP_CL_BUILD_KONERB2`. In der Erbauerklasse wird definiert, welche Methoden von den konkreten Erbauerklassen implementiert werden müssen. In unserem Beispiel sind dies die Methoden `ERZEUGE_BAUTEIL_A()` und `ERZEUGE_BAUTEIL_B()`. Anschließend wird das Objekt `PRODUKT` über den Rückgabeparameter `R_PRODUKT` der Methode `ERHALTE_ERGEBNIS()` an den Aufrufer/Direktor bzw. Client zurückgegeben.

```
CLASS zdp_cl_build_erbauer DEFINITION
  PUBLIC   ABSTRACT
  CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS erzeuge_bauteil_a.
    METHODS erzeuge_bauteil_b.
    METHODS erhalte_ergebnis
      RETURNING
        VALUE(r_produkt) TYPE REF TO zdp_cl_build_produkt.
```



```
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 3.3 Definition der Erbauerklasse innerhalb des Builder Patterns

Erster konkreter Erbauer

Als erster konkreter Erbauer wird in Listing 3.4 die Klasse des konkreten Erbauers `ZDP_CL_BUILD_KONERB1` erstellt. Durch die abstrakte Klasse `ZDP_CL_BUILD_ERBAUER` wurde definiert, welche Methoden in den einzelnen konkreten Erbauern implementiert werden müssen.

```
CLASS zdp_cl_build_konerb1 DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_build_erbauer
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS constructor.
    METHODS erzeuge_bauteil_a
         REDEFINITION.
    METHODS erzeuge_bauteil_b
         REDEFINITION.
    METHODS erhalte_ergebnis
         REDEFINITION.
  PROTECTED SECTION.
  PRIVATE SECTION.
DATA produkt TYPE REF TO zdp_cl_build_produkt.

CLASS zdp_cl_build_konerb1 IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD super->constructor.
    CREATE OBJECT me->produkt.
  ENDMETHOD.

  METHOD erzeuge_bauteil_a.
    produkt->hinzufuegen('Bauteil-A').
  ENDMETHOD.
  METHOD erzeuge_bauteil_b.
    produkt->hinzufuegen('Bauteil-B').
  ENDMETHOD.

  METHOD erhalte_ergebnis.
    produkt = me->produkt.
    r_produkt = me->produkt.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.4 Builder Pattern – Definition des konkreten Erbauers

Zweiter konkreter Erbauer

In Listing 3.5 wird die Klasse des zweiten konkreten Erbauers `ZDP_CL_BUILD_KONERB2` implementiert. Es können unterschiedliche Erbauer erstellt werden, die dem Produkt unterschiedliche Bauteile hinzufügen können. In einer Unternehmensanwendung kann das Produkt z. B. ein Auftrag sein. Der erste konkrete Erbauer erzeugt dann z. B. Produktionsaufträge, ein zweiter konkreter Erbauer z. B. Lieferaufträge.

```
CLASS zdp_cl_build_konerb2 DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_build_erbauer
  FINAL   CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS constructor.
    METHODS erzeuge_bauteil_a
         REDEFINITION.
    METHODS erzeuge_bauteil_b
         REDEFINITION.
    METHODS erhalte_ergebnis
         REDEFINITION.
  PROTECTED SECTION.
  PRIVATE SECTION.
    DATA produkt TYPE REF TO zdp_cl_build_produkt.
ENDCLASS.

CLASS zdp_cl_build_konerb2 IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD super->constructor.
    CREATE OBJECT me->produkt.
  ENDMETHOD.

  METHOD erzeuge_bauteil_a.
    produkt->hinzufuegen('Bauteil-X').
  ENDMETHOD.

  METHOD erzeuge_bauteil_b.
    produkt->hinzufuegen('Bauteil-Y').
  ENDMETHOD.

  METHOD erhalte_ergebnis.
    produkt = me->produkt.
    r_produkt = produkt.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.5 Builder Pattern – Definition des zweiten konkreten Erbauers

Direktor

Als Nächstes implementieren Sie den Direktor wie in Listing 3.6. Der Direktor ist dafür zuständig, die Erzeugung der unterschiedlichen Bauteile des Produkts anzustoßen und diese in dem Produkt abzuspeichern. Der Direktor wird vom Client über die Methode `KONSTRUIERE()` aufgerufen. Dieser Aufruf führt dazu, dass die unterschiedlichen Bauteile für das Produkt erzeugt werden.

```
CLASS zdp_cl_build_direktor DEFINITION
  PUBLIC  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS konstruiere
      IMPORTING
        !i_erbauer TYPE REF TO zdp_cl_build_erbauer.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_build_direktor IMPLEMENTATION.
  METHOD konstruiere.
    i_erbauer->erzeuge_bauteil_a( ).
    i_erbauer->erzeuge_bauteil_b( ).
  ENDMETHOD.
ENDCLASS.
```

Listing 3.6 Builder Pattern – Definition der Direktorklasse

Client

Die Klasse `ZDP_CL_BUILD_APPLIKATION`, deren Implementierung Sie in Listing 3.7 sehen, ist die Anwendungsklasse, die den Client repräsentiert. In dieser Klasse treffen alle Komponenten zusammen, die für das Builder Pattern benötigt werden. In der einzigen Methode `START()` werden zunächst alle Objekte der unterschiedlichen Klassen erzeugt, die benötigt werden, um den Direktor aufzurufen. Der Direktor lenkt dann automatisch alle Aktionen, um das Produkt mit seinen Bauteilen zu erzeugen.

```
CLASS zdp_cl_build_applikation DEFINITION
  PUBLIC   FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    DATA lo_erbauer TYPE REF TO zdp_cl_build_erbauer.
    METHODS start.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

```
CLASS zdp_cl_build_applikation IMPLEMENTATION.
  METHOD start.
    DATA: lo_direktor TYPE REF TO zdp_cl_build_direktor.
    DATA: lo_b1 TYPE REF TO zdp_cl_build_konerb1.
    DATA: lo_b2 TYPE REF TO zdp_cl_build_konerb2.
    DATA: lo_p1 TYPE REF TO zdp_cl_build_produkt.
    DATA: lo_p2 TYPE REF TO zdp_cl_build_produkt.

    CREATE OBJECT lo_direktor.
    CREATE OBJECT lo_b1.

    CALL  METHOD lo_direktor->konstruiere( lo_b1 ).
    lo_p1 = lo_b1->erhalte_ergebnis( ).
    lo_p1->anzeigen( ).

    CREATE OBJECT lo_b2.

    CALL METHOD lo_direktor->konstruiere( lo_b2 ).
    lo_p2 = lo_b2->erhalte_ergebnis( ).
    lo_p2->anzeigen( ).
  ENDMETHOD.
ENDCLASS.
```

Listing 3.7 Builder Pattern – Definition und Implementierung der Applikationsklasse

Integration in Report

Um das Builder Pattern in einer Anwendung verwenden zu können, muss jetzt nur noch die Applikationsklasse aufgerufen werden. Der Quellcode dieser Klasse kann auch direkt in eine Anwendung integriert werden. In unserem Beispiel integrieren wir den Quellcode in einen ABAP-Report mit der folgenden Codezeile:

```
REPORT zdp_ar_erz_builder.
DATA lo_app TYPE REF TO zdp_cl_build_applikation.
CREATE OBJECT lo_app.
lo_app->start( ).
```

Listing 3.8 Builder Pattern – Aufruf im ABAP-Report

Report ausführen

Das Ergebnis dieser Implementierung erhalten Sie durch die Ausführung des ABAP-Reports `ZDP_AR_ERZ_BUILDER` über Transaktion SE38. Dabei wird die in Abbildung 3.4 dargestellte Ausgabe angezeigt.

Anhand der Ausgabe in der Ausführungskonsole erkennen Sie, wie die einzelnen Objekte aufgebaut sind, die durch das Builder Pattern erzeugt wurden. Sie sehen, dass das Produkt 1 erzeugt wurde, in dem die Bauteile A und B enthalten sind. Diese Objekte wurden

durch den konkreten Erbauer 1 erzeugt. Hingegen wurden bei Produkt 2 die Bauteile X und Y durch den konkreten Erbauer 2 erzeugt.

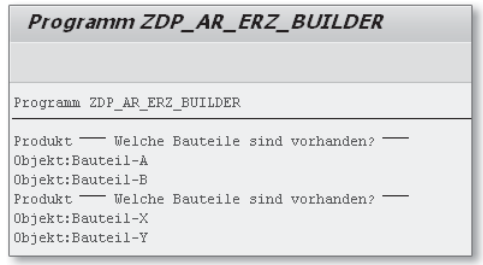

Abbildung 3.4 Builder Pattern – Reportausgabe

3.1.5 Evaluation

Vorteile

Ein entscheidender Vorteil des Builder Patterns ist, dass bei der Implementierung die Objekte der Repräsentationsschicht komplett von den Konstruktionsobjekten bzw. der Konstruktionsschicht isoliert sind. Dies wird durch die Objektkapselungen erreicht. Das bedeutet, dass die Anwendung sehr robust und weniger angreifbar durch Injections (Angriffe z. B. durch Hacking) sowie weniger fehleranfällig durch die Kapselungen ist.

Ein weiterer Vorteil ist, dass die Erbauerklasse als Repräsentant nach außen dient, d .h. für andere Klassen der Anwendung oder für den Client über die Benutzeroberfläche aufrufbar ist. In der Repräsentationsschicht werden die internen Klassen, d. h. die konkreten Erbauer, vor dem Direktor versteckt. Die einzelnen Klassen sind dabei voneinander isoliert, obwohl sie in einer Anwendung zusammenarbeiten.

Ein weiterer wichtiger Grund, dieses Entwurfsmuster zu verwenden, ist, dass der gesamte Konstruktionsprozess über eine zentrale Stelle, den Direktor, gesteuert wird. Dieser Konstruktionsprozess kann über die verschiedenen konkreten Erbauerklassen in mehrere Phasen untergliedert werden, was z. B. beim Factory Pattern (siehe Abschnitt 3.2) nicht der Fall ist. Dazu sind seitens des Clients keine weiteren Aktivitäten notwendig.

Auch bei einer späteren Weiterentwicklung der Anwendung hat die Verwendung des Builder Patterns Vorteile. Da die Methoden in einzelnen Klassen gekapselt vorliegen, können sehr einfach weitere

Klassen hinzugefügt werden, um neue Produkte zu erzeugen. Möchten Sie eine Anwendung entwickeln, die auf neue Anforderungen durch neue Objekte sehr schnell und einfach reagieren kann, empfehlen wir Ihnen die Verwendung dieses Entwurfsmusters.

Nachteile

Um das Builder Pattern zu implementieren, sind allerdings umfassende und vollständige Kenntnisse des Konstruktionsprozesses aller Objekte der Anwendung erforderlich. Nur so können die Objekte in der Laufzeitumgebung richtig eingesetzt werden. Dies bedeutet, dass bei komplexen Anwendungen alle Objekte vor der Umsetzung erst einmal dokumentiert bzw. evaluiert werden müssen. Dies ist umso mehr erforderlich, als die Objekte später bei der Implementierung nur noch isoliert voneinander betrachtet werden.

Ein weiterer erheblicher Nachteil ist, dass die Klassen beim Builder Pattern sehr stark miteinander verzahnt sind. Diese Kopplung erlaubt es nicht, die Klassen in anderen Klassen einfach wiederzuverwenden.

3.2 Factory Pattern

Das *Factory Pattern* wird im Deutschen auch als Fabrik-Entwurfsmuster bezeichnet und gehört ebenso wie das Builder Pattern zu den GoF-Entwurfsmustern. Das Factory Pattern wird eingesetzt, um die Konstruktion von Objekten von ihren Repräsentationen zu trennen. Im Unterschied zum Builder Pattern kann das Factory Pattern aber nicht so einfach nach Belieben ausgebaut werden. Ein mit dem Factory Pattern verwandtes Pattern ist das Singleton Pattern, das wir in Abschnitt 3.3 erläutern.

Merkmale des Konstruktionsprozesses

Ein mithilfe des Factory Patterns implementierter Konstruktionsprozess kann jederzeit wiederverwendet werden. Beim Factory Pattern werden privat deklarierte Konstrukte, die Methoden und Konstruktoren, dazu verwendet, eine direkte Instanziierung von anderen Klassen zu verhindern. Stattdessen erfolgt die Objekterzeugung mithilfe einer statischen Methode. Das Objekt wird über diese statische Methode zurückgegeben.

Factory-Typen

Es gibt die folgenden drei Möglichkeiten, ein Factory Pattern zu implementieren:

- **Factory Pattern auf Basis von Methoden**
  Factory-Methoden sind statische Methoden, die zur Erzeugung von Objekten eines Klassentyps verwendet werden. Das Singleton Pattern nutzt die Factory-Methode. Dabei liest das Pattern das zu erstellende Objekt durch die statischen Methoden aus oder lässt dieses Objekt instanziieren.
- **Factory Pattern auf Basis von Klassen**
  Eine Factory-Klasse basiert auf dem Factory-Methoden-Konzept und erweitert dieses Konzept durch verschiedene Klassen. In Factory-Klassen werden nicht nur die zur Entwurfsmusterimplementierung gehörenden Klassen instanziiert, sondern auch vollständige und komplexe andere Objekte. So wird ein höherer Abstraktionsgrad erzielt, der es viel leichter erlaubt, eine Factory-Klasse durch andere Klassen auszutauschen, die ebenfalls dem Konzept der Factory-Klasse entsprechen.
- **Factory Pattern auf Basis abstrakter Klassen und Interfaces**
  Die Implementierung mit einer abstrakten Factory-Klasse ist eine sehr aufwendige Implementierungsvariante des Entwurfsmusters. Oft trägt diese Umsetzungsvariante auch den Spitznamen Toolkit. Die abstrakte Factory-Klasse hat zwei wichtige Merkmale, die gleichzeitig ihre Vorteile ausmachen:
  – Während des Erzeugungsprozesses wird nicht nur ein konkretes Element (d. h. ein Objekttyp) erzeugt, sondern es können viele und unterschiedliche Elemente erzeugt werden.
  – Darüber hinaus ist die abstrakte Factory-Klasse sehr flexibel gebaut, was das Hinzufügen und Austauschen von Objekten anhand zusätzlicher Factories vereinfacht.

3.2.1 Problem

Gruppierung der zu erzeugenden Parameter

Ein Objekt besitzt verschiedene Parameter, die bei der Erzeugung dieses Objekts unterschiedlich angelegt werden müssen. Es können z. B. unterschiedliche Aufträge erzeugt werden, die alle anhand eines Parameters definiert werden. Dazu können Standardinformationen, wie z. B. die Auftragsnummer, das Tagesdatum etc. bereits im Rahmen der Objekterzeugung festgelegt werden.

Das Factory Pattern stellt dazu eine Schnittstelle bereit, die die Objekte entsprechend erzeugt. Die erzeugten Objekte können miteinander verwandt oder auf andere Weise voneinander abhängig sein.

3.2.2 Ansatz und Lösung

Aufbau

Eine Skizze, wie das Factory Pattern umgesetzt werden kann, finden Sie in Abbildung 3.5. Hier sollen Aufträge in verschiedenen SAP-ERP-Komponenten erzeugt werden. Auf der linken Seite ist der Anwender eingezeichnet, der auch als *Client* bezeichnet wird. Dieser kann entweder eine abstrakte Factory oder eine konkrete Factory-Klasse anstoßen. Eine abstrakte Factory wird verwendet, wenn später unterschiedliche konkrete Factory-Klassen implementiert werden sollen. Die einfachste Variante ist es, nur eine Factory-Klasse zu implementieren. Dabei greift der Anwender direkt auf die konkrete Factory-Klasse zu. Die konkrete Factory-Klasse bzw. die -Klassen erzeugen einzelne Objekte und geben diese als Objektreferenz zurück. In Abbildung 3.5 ist dieses Objekt der konkrete Auftrag, der auf einer Klasse basiert.

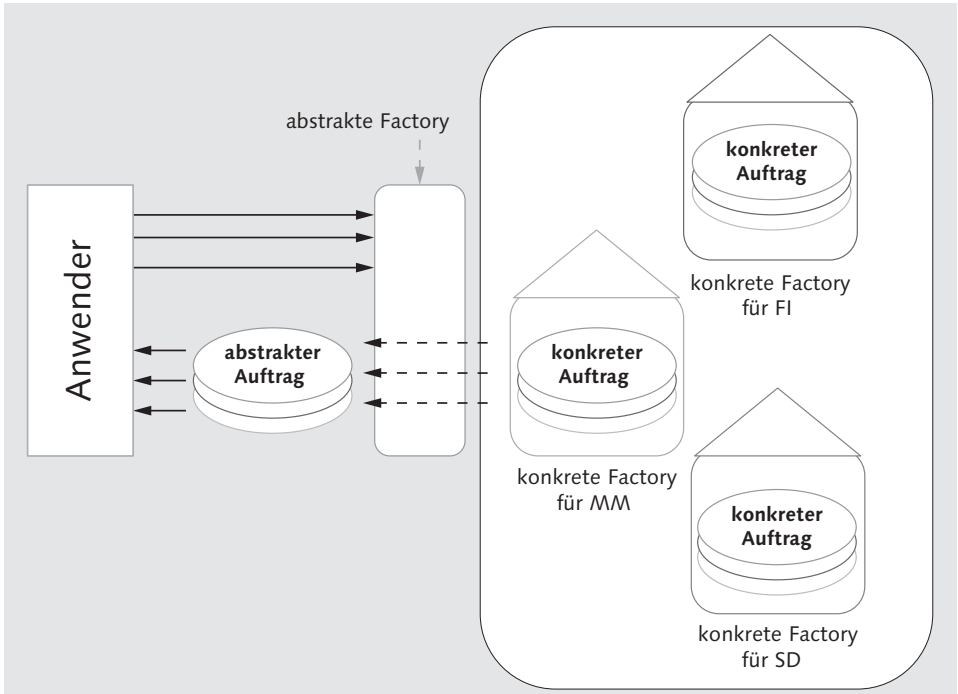

Abbildung 3.5 Factory Pattern – Aufbau

Diese Factory-Klassen erzeugen die Produkte für den Client, z. B. einen Auftrag. Wie dies im Detail abläuft, erläutern wir in diesem Abschnitt.

Umsetzung in Klassen

Abbildung 3.6 zeigt ein UML-Klassendiagramm zur technischen Umsetzung dieses Modells. Sie erkennen fünf ABAP-Klassen, über die der Konstruktionsprozess in der Factory-Klasse abgebildet wird:

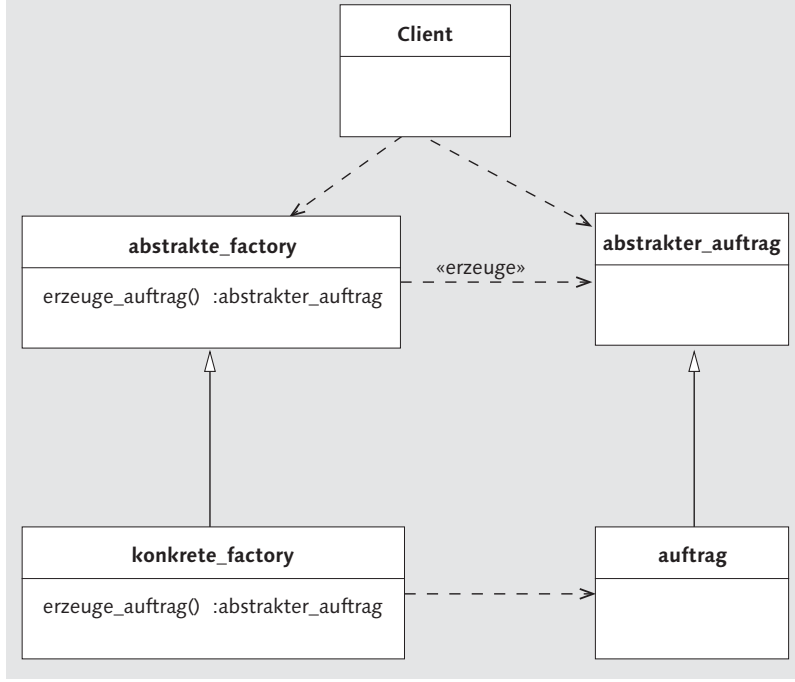

Abbildung 3.6 Factory Pattern – UML-Klassendiagramm

- **Abstrakte Factory-Klasse**
  Die Klasse `abstrakte_factory` hat die Aufgabe, Objekte von dritten Klassen zu erzeugen und diese dem Anwender bereitzustellen. Dies können abstrakte oder implementierte Klassen sein. Die `abstrakte_factory` wird im Klassendiagramm als Superklasse dargestellt.
- **Konkrete Factory-Klasse**
  Neue Factory-Klassen erben von der abstrakten Superklasse und heißen `konkrete_factory`. Die konkreten Factory-Klassen enthalten die Methoden zur Objekterzeugung, in unserem Beispiel der Klasse `auftrag`. Abhängig von der Komplexität des zu erzeugen-

den Objekts können dazu unterschiedlich viele Factory-Klassen verwendet werden.

Die konkreten Factory-Klassen übergeben die Objekte mithilfe eines Rückgabewertes an die Klasse `client`, also der Anwender, zurück. Die abstrakte Factory-Klasse definiert den Rahmen, welche Methoden in den konkreten Factory-Klassen zur Verfügung stehen. Durch den Aufruf eines Objekts dieser Factory-Klasse wird dieses Objekt durch die Methode `erzeuge_auftrag()` erzeugt. Der Anwender erhält von dem abstrakten Factory-Klassen-Objekt ein Objekt zurück, nämlich seinen konkreten Auftrag. In unserem Beispiel können unterschiedliche Aufträge auf Basis verschiedener Klassen erzeugt werden, die eine gemeinsame Superklasse `abstrakter_auftrag` haben. Um die konkreten Klassen ausprägen zu können, müssen diese zunächst klassifiziert werden. Diese Klassifizierung erfolgt ebenfalls über eine Factory-Klasse, die dem Anwender in der Klasse `client` zur Verfügung gestellt wird.

- **Client**
  Der Client selbst hat mit der eigentlichen Erzeugung der konkreten Objekte nichts zu tun. Seine einzige Aufgabe ist es, die Factory-Klasse auszuwählen, die das Objekt erzeugen soll. Für den Auslagerungsauftrag in der SAP-Komponente SD muss z. B. eine Klasse erzeugt werden. Das Factory Pattern stellt alle erforderlichen Funktionen für die Erzeugung der Objekte zur Verfügung.

Zusätzliche Abstraktionsebene

Der entscheidende Vorteil des abstrakten Factory Patterns ist die Nutzung der zusätzlichen Abstraktionsebene für die Objekterzeugung. Der Client nutzt zur Laufzeit die durch die abstrakte Klasse `abstrakte_factory` zur Verfügung gestellten Methoden. Diese Klasse ist ein Platzhalter für eine konkrete Klasse zur Auftragserzeugung, die von dieser Klasse erbt und dadurch alle Standardmethoden von ihr übernimmt. Die Ausimplementierung der abstrakten Klasse erfolgt in der Klasse `konkrete_factory`. Diese bleibt dem Client verborgen, er kann sie jedoch für die Objekterzeugung nutzen.

Ablauf einer Factory-Pattern-Anwendung

In Abbildung 3.7 haben wir den Ablauf einer nach dem Factory Pattern konzipierten Anwendung in einem Sequenzdiagramm skizziert. Der Client ruft hier eine Methode auf. Diese Methode wird z. B. über die Anwendungsoberfläche vom Anwender ausgelöst, z. B. durch den Klick auf einen Button. Die Implementierung dieser Methode ruft wiederum eine Methode der abstrakten Factory-Klasse auf.

Dabei werden weitere Methoden angestoßen, die dann sowohl das Objekt des Auftrags als auch das der konkreten Factory-Klasse erzeugen. Diese abstrakte Factory ruft die Methode `factory_methode()` in der konkreten Factory-Klasse auf. Diese Methode erzeugt ein konkretes Objekt, in diesem Fall einen Auftrag auf Basis der Informationen, die sie über die abstrakte bzw. die konkrete Factory-Klasse erhalten hat. Dieses erzeugte Objekt wird über die Rückgabeparameter der Methode zurück an die Client-Klasse gegeben. Zur Objekterzeugung spricht der Client die konkrete Factory-Klasse über die abstrakte Factory-Klasse an.

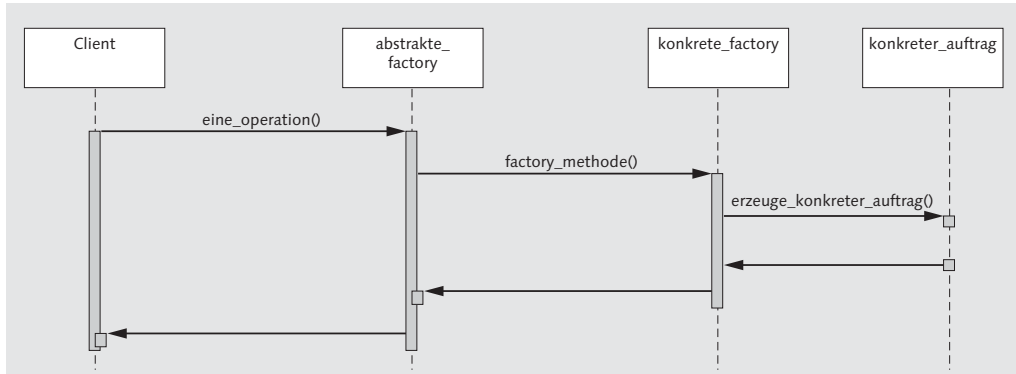

Abbildung 3.7 Factory Pattern – UML-Sequenzdiagramm

3.2.3 Einsatzbeispiel

Das Factory Pattern kann auch dann verwendet werden, wenn das zu erzeugende Objekt der erzeugenden Klasse unbekannt sein soll. Erst durch die Unterklassen wird definiert, welche Objekte erzeugt werden sollen. Auf diese Weise können durch die Factory-Klassen ganz unterschiedliche Objekte erzeugt werden. Der Aspekt der Erzeugung wird in der Factory-Klasse zentralisiert und völlig von den zu erzeugenden Objekten getrennt.

Beispiel: Belegarten erzeugen

In Abbildung 3.8 skizzieren wir als Beispiel eine Anwendung zur Erzeugung verschiedener Belegarten im SAP-Finanzwesen FI (Einbuchungs-, Ausbuchungs- und Umbuchungsbelege). Dabei kann es sich z. B. um Einbuchungsbelege für bestimmte Paletten in einem Lager handeln. Mit der konkreten Factory-Klasse `ZDP_CL_FI_VERARBEITUNG` lassen sich unterschiedliche Belegarten erzeugen. Die Belegobjekte werden hier zentral erzeugt und zurückgegeben. Die konkrete Erzeu-

gung der unterschiedlichen Belegarten erfolgt dann in unterschiedlichen Factory-Methoden `ERZEUGE_FI_AUSBUCHUNG()`, `ERZEUGE_FI_EINBUCHUNG()` und `ERZEUGE_FI_UMBUCHUNG()`.

Anstelle der konkreten Factory-Klasse können Sie auch mit einer abstrakten Factory-Klasse arbeiten und erbende Factory-Klassen zwischenschalten. So hätten Sie eine breitere Schnittstelle, die Ihnen mehr Möglichkeiten bietet, unterschiedliche Belegarten zu erzeugen. Sinnvoll ist diese Abstraktionsebene nur, wenn die Clusterung der Belegerzeugung innerhalb eines Projekts möglich ist. Die Klasse `ZDP_CL_FI_VERARBEITUNG` kann in der Implementierung sehr einfach durch die Anwendung, also den Client, angesprochen werden. Beim Erzeugen der einzelnen FI-Belege werden diese Belege über das Attribut `I_BELEGE` durch die Methode `HINZUFUEGEN()` einer internen Tabelle hinzugefügt. Diese Methode `HINZUFUEGEN()` wird über eine Client-Klasse aufgerufen, in unserem Beispiel bezeichnen wir sie als `ZDP_CL_CLIENT`.

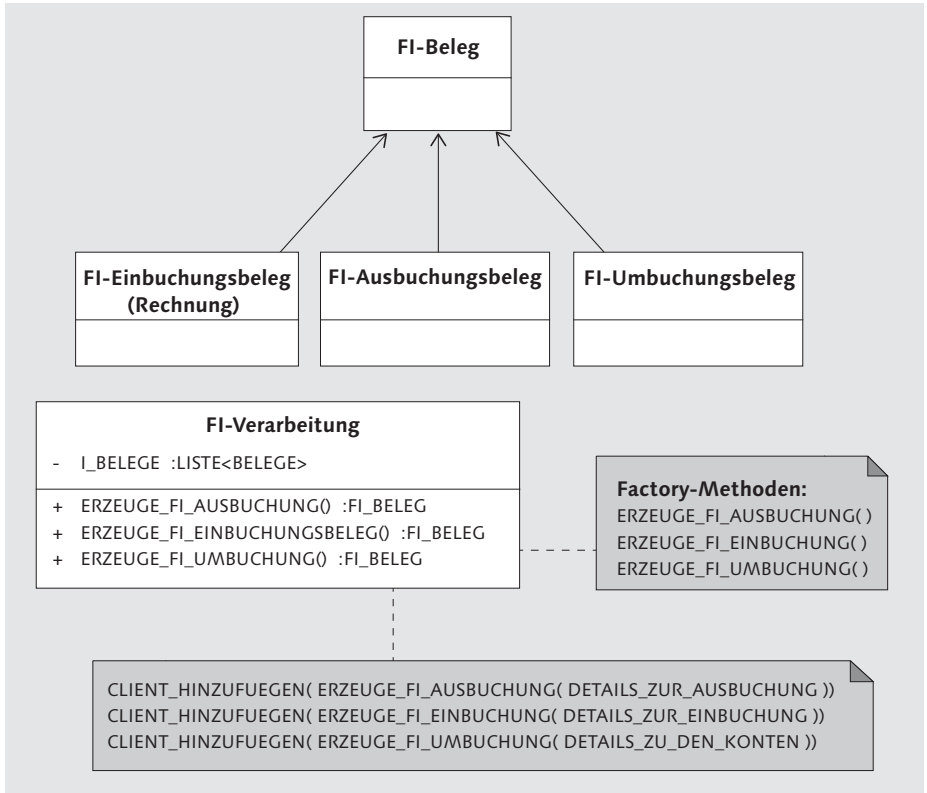

Abbildung 3.8 Factory Pattern – UML-Klassendiagramm einer Beispielanwendung

3.2.4 Umsetzung in ABAP

Belegerzeugung mit abstrakter Factory

In diesem Abschnitt setzen wir das Factory Pattern in einer ABAP-Anwendung um, die zwei verschiedene Belegarten erzeugt, und setzen dabei in Erweiterung des Beispiels aus dem vorangegangenen Abschnitt eine abstrakte Factory-Klasse ein. Die beiden Belegarten Einbuchungs- und Ausbuchungsbeleg werden von zwei verschiedenen Factory-Klassen erzeugt. Diese beiden Objektklassen basieren auf der abstrakten Superklasse `ZDP_CL_FACT_BELEG`. Die konkreten Erbauerklassen basieren auf der abstrakten Factory-Superklasse `ZDP_CL_FACT_ERBAUER`, die alle Informationen zentral für die konkreten Factory-Klassen bereitstellt. Die konkreten Erbauerklassen erben also von dieser Superklasse `ZDP_CL_FACT_ERBAUER`.

Report zur Klassendefinition

Für das Factory Pattern legen Sie zunächst über Transaktion SE38 einen ABAP-Report an.

```
REPORT zdp_ar_erz_factory.
```

Definition Superklasse

In Listing 3.9 wird die abstrakte Superklasse `ZDP_CL_FACT_BELEG` definiert.

```
CLASS zdp_cl_fact_beleg DEFINITION
  PUBLIC
  ABSTRACT
  CREATE PUBLIC.

  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 3.9 Factory Pattern – Definition der Beleg-Superklasse

Definition der konkreten Belegklassen

Im nächsten Schritt werden die konkreten Belegklassen `ZDP_CL_FACT_KONBELEIN` für den Einbuchungsbeleg und `ZDP_CL_FACT_KONBELAUS` für den Ausbuchungsbeleg definiert (siehe Listing 3.10). Diese Belegklassen erben von der in Listing 3.9 definierten Superklasse `ZDP_CL_FACT_BELEG`. Die Belegklassen selbst können Sie noch ausgestalten, um die verschiedenen Belegausprägungen zu definieren.

```
CLASS zdp_cl_fact_konbelein DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_fact_beleg
  FINAL
  CREATE PUBLIC.
```

```
  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_fact_konbelaus DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_fact_beleg
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 3.10 Factory Pattern – Definition der erbenden Belegklassen

Definition der abstrakten Erbauer-Factory

In Listing 3.11 wird die abstrakte Factory-Klasse definiert und implementiert – also die Klasse, der innerhalb des Factory Patterns die Rolle des Erbauers zukommt. Die Factory-Klasse ist für den Erzeugungsprozess der zwei Belegtypen zuständig. Von der abstrakten Superklasse `ZDP_CL_FACT_ERBAUER` erben in unserem Beispiel zwei konkrete Factory-Klassen. Durch Realisierung der Factory-Klasse als abstrakte Superklasse ist es möglich, die Anzahl der erbenden Factory-Klassen beliebig zu erhöhen. Diese Factory-Klassen verfügen jeweils über die gleiche Methode `ERZEUGE_BELEG()` zur Erzeugung von Belegen. In der abstrakten Factory wird der Methodenrahmen der Methode `ERZEUGE_BELEG()` definiert. Diese Methode wird in den konkreten Factory-Klassen so ausimplementiert, dass die Belegobjekte erzeugt werden können.

```
CLASS zdp_cl_fact_erbauer DEFINITION
  PUBLIC
  ABSTRACT
  CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS erzeuge_beleg
      RETURNING
        VALUE(r_beleg) TYPE REF TO zdp_cl_fact_beleg.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 3.11 Factory Pattern – Definition der abstrakten Factory-Klasse

Definition der erbenden Factory-Klassen

In Listing 3.12 wird definiert, dass die konkrete Erbauerklasse `ZDP_CL_FACT_KONERBEIN` von der abstrakten Klasse `ZDP_CL_FACT_ERBAUER` erbt. Dies bedeutet, dass alle Methoden der vererbenden Klasse in der Klasse `ZDP_CL_FACT_KONERBEIN` zur Verfügung stehen. Damit die konkrete Erbauerklasse den Einbuchungsbeleg erzeugt, muss die Methode `ERZEUGE_BELEG()` redefiniert werden. Dabei bleibt die Methode der Superklasse unverändert. Sie wird bei der Redefinition lediglich für die Objekte der Klasse `ZDP_CL_FACT_KONERBEIN` überschrieben. Bei der Klasse `ZDP_CL_FACT_KONERBAUS` funktioniert dieses Prinzip auf die gleiche Weise.

```
CLASS zdp_cl_fact_konerbein DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_fact_erbauer
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS erzeuge_beleg
          REDEFINITION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_fact_konerbein IMPLEMENTATION.
  METHOD erzeuge_beleg.
    CREATE OBJECT r_beleg TYPE zdp_cl_fact_konbelein.
  ENDMETHOD.
ENDCLASS.

CLASS zdp_cl_fact_konerbaus DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_fact_erbauer
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.

    METHODS erzeuge_beleg
          REDEFINITION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_fact_konerbaus IMPLEMENTATION.
  METHOD erzeuge_beleg.
    CREATE OBJECT r_beleg TYPE zdp_cl_fact_konbelaus.
```

```
  ENDMETHOD.
ENDCLASS.
```

Listing 3.12 Factory Pattern – Definition der konkreten Erbauerklassen

Ausführende Klasse

Um das Pattern ausführen zu können, wird eine weitere ABAP-Klasse benötigt. Dieser Quellcode wird letztlich in die Anwendung, die das Factory Pattern beinhaltet, implementiert und sorgt für eine zusätzliche Kapselung. Diese Kapselung trägt zu einer besseren Standardisierung der Objekte bei, die die Anwendung erzeugen kann. Zusätzlich wird aufgrund der Kapselung der einzelnen Klassen die Robustheit der Entwicklung deutlich erhöht. Dies führt dazu, dass sich ein möglicher Fehler in der Anwendung zum einen deutlich schneller eingrenzen lässt, um den Fehler zu beheben, und zum anderen auch nicht unbedingt gleich die ganze Anwendung stilllegt. Der Quellcode dieser kapselnden Klasse `ZDP_CL_FACT_APPLIKATION` beinhaltet die Methode `START()`, über die das Entwurfsmuster angesteuert und die Erzeugung der Objekte gestartet wird.

Tabelle LT_ERBAUER

Zu Beginn werden einige Variablen deklariert, die später für die Verarbeitung benötigt werden. Dabei ist die Tabelle `LT_ERBAUER` hervorzuheben, die alle Objekte der konkreten Erbauerklassen enthält. Diese Tabelle wurde mit dem Referenztyp der Superklasse `ZDP_CL_FACT_ERBAUER` deklariert, was bedeutet, dass alle Objekte, die von der Superklasse geerbt haben, in der ausführenden Klasse hinzugefügt werden können. Durch das Durchlaufen dieser Tabelle `LT_ERBAUER` in einer Schleife können wir über den Klassennamen prüfen, um welchen konkreten Erbauer es sich bei dem Objekt handelt. In dieser Schleife ruft die ausgewählte Erbauerklasse auch ihre Erzeugungsmethode `ERZEUGE_BELEG()` auf, die dann die einzelnen Belege erzeugt. Bei der Ausgabe können Sie sehen, dass über die konkrete Erbauerklasse das Belegobjekt, also der Einbuchungs- oder Ausbuchungsbeleg, erzeugt wurde.

In der Variable `LR_DESCRIBER` werden die Objekteigenschaften, die über den Befehl `CL_ABAP_REFDESCR=>DESCRIBE_BY_OBJECT_REF( <FS> )` aus dem Objekt ausgelesen werden, vorgehalten. In unserem Beispiel lesen wir den absoluten Klassennamen über den Ausdruck `LR_DESCRIBER->ABSOLUTE_NAME+7` aus.

```
CLASS zdp_cl_fact_applikation DEFINITION
  PUBLIC   FINAL
  CREATE PUBLIC.
```



```
  PUBLIC SECTION.
    DATA lt_erbauer TYPE TABLE OF REF TO zdp_cl_fact_erbauer.
    METHODS start.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_fact_applikation IMPLEMENTATION.
   METHOD start.
   DATA: lo_erbauer TYPE REF TO zdp_cl_fact_erbauer.
   DATA: lt_erbauer TYPE TABLE OF REF TO zdp_cl_fact_erbauer.
   DATA: lo_konkreter_erbauer_ein TYPE REF TO
                                        zdp_cl_fact_konerbein.
   DATA: lo_konkreter_erbauer_aus TYPE REF TO
                                        zdp_cl_fact_konerbaus.
   DATA: lo_beleg TYPE REF TO zdp_cl_fact_beleg.
   DATA: class_name TYPE abap_typename.

   FIELD-SYMBOLS <fs> TYPE any.

   CREATE OBJECT lo_konkreter_erbauer_ein.
   APPEND lo_konkreter_erbauer_ein TO lt_erbauer.

   CREATE OBJECT lo_konkreter_erbauer_aus.
   APPEND lo_konkreter_erbauer_aus TO lt_erbauer.

   LOOP AT lt_erbauer ASSIGNING <fs>.
     DATA:
      lr_describer         TYPE REF TO cl_abap_typedescr,
      lv_cls_absolute_name TYPE abap_abstypename.
      lr_describer = cl_abap_refdescr=>
                            describe_by_object_ref( <fs> ).
      lv_cls_absolute_name = lr_describer->absolute_name+7.
      CASE lv_cls_absolute_name.
        WHEN 'ZDP_CL_FACT_KONERBEIN'.
          WRITE: / 'Einbuchungsbeleg erzeugt '.
        WHEN 'ZDP_CL_FACT_KONERBAUS'.
          WRITE: / 'Ausbuchungsbeleg erzeugt'.
      ENDCASE.
      lo_erbauer = <fs>.
      lo_beleg = lo_erbauer->erzeuge_beleg( ).
      WRITE: / 'Auftrag - ',lv_cls_absolute_name.
    ENDLOOP.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.13 Factory Pattern – Definition der Hauptausführungsklasse, d. h. der Applikationsklasse

Aufruf im ABAP-Report

Zur Verwendung des Factory Patterns müssen wir im letzten Schritt die Applikationsklasse `ZDP_CL_FACT_APPLIKATION` über den zu Beginn definierten ABAP-Report `ZDP_AR_ERZ_FACTORY` aufrufen. Dies erfolgt über folgenden Quellcode:

```
REPORT zdp_ar_erz_factory.
DATA lo_app TYPE REF TO zdp_cl_fact_applikation.
CREATE OBJECT lo_app.
lo_app->start( ).
```

Listing 3.14 Factory Pattern – Aufruf im ABAP-Report

Wird das in unserem Beispiel implementierte Factory Pattern in einem ABAP-Report gestartet, werden durch das Durchlaufen der `LOOP AT`-Schleife die einzelnen Erbauer initialisiert. Durch den Aufruf der Erzeugungsmethoden wird eine Ausgabe erzeugt, anhand derer Sie erkennen können, wie dieses Entwurfsmuster arbeitet (siehe Abbildung 3.9). Bei der Prüfung des von der ersten konkreten Erbauerklasse zu erzeugenden Objekts wird die erste Ausgabe erzeugt. Dabei wird anhand des absoluten Klassennamens des Objekts erkannt, welches der Objekte erzeugt wurde. Wie auch bei der Applikationsklasse `ZDP_CL_FACT_APPLIKATION` gibt es hierfür eine Standardklasse `CL_ABAP_REFDESCR` von SAP. In unserem Beispiel können wir auf diese Weise ableiten, welcher Beleg erzeugt wurde.

Das Ergebnis dieser Implementierung erhalten Sie durch die Ausführung des ABAP-Reports `ZDP_AR_ERZ_FACTORY` über Transaktion SE38. Dabei wird folgende Ausgabe angezeigt:

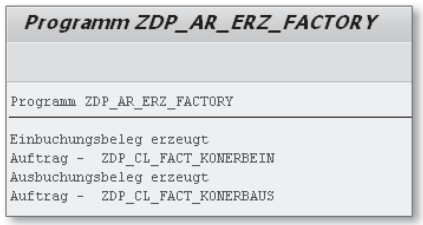

Abbildung 3.9 Factory Pattern – Reportausgabe

3.2.5 Evaluation

Einsatzgründe

Das Factory Pattern bietet wie das Builder Pattern erhebliche Vorteile bei der Erzeugung von Objekten. Sechs entscheidende Faktoren sprechen dafür, dieses Pattern zu nutzen:

- **Kapselung**
  Durch das Factory Pattern können konkrete Klassen abgeschirmt werden, sodass Entwickler die internen Klassen nicht verwenden bzw. Objekte damit erzeugen können. Dadurch werden die Robustheit und Sicherheit der Anwendung verbessert.
- **Konsistenz**
  Durch die Kapselung kann außerdem die Konsistenz der Daten sichergestellt werden, wodurch sich unterscheiden lässt, welche Daten zur eigentlichen Factory-Klasse und welche zum konkreten erzeugenden Objekttyp (z. B. Beleg, Auftrag) gehören.
- **Flexibilität**
  Die zu erzeugenden Objekte können sehr einfach durch andere Objekte ausgetauscht werden.
- **Abstraktion**
  Durch die Abstraktion in der Superklasse können die Factory- und Erbauerklassen einfach und schnell erweitert werden. Dazu müssen nur die Factory-Schnittstelle und die konkrete Factory-Klasse erzeugt werden, um das Entwurfsmuster um einen zusätzlichen Objekttyp zu erweitern.
- **Wiederverwendbarkeit**
  Durch diese Flexibilität und Abstraktion ist das Factory Pattern leicht wiederverwendbar. Jedes beliebige Produkt jeder Produktfamilie kann mithilfe des Patterns sehr einfach erzeugt werden. Die zu erzeugenden Produkte können in einem anderen Kontext einfach durch andere ausgetauscht werden.
- **Einfachheit**
  Das komplette Factory Pattern kommt sowohl aufseiten des Clients als auch aufseiten der Factory-Klasse mit nur wenig Code aus.

Nachteile

Das Factory Pattern hat in bestimmten Fällen aber auch Nachteile. Sollen mit diesem Pattern Produkte erzeugt werden, die nicht zur ursprünglichen Produktfamilie gehören, muss die Schnittstelle der abstrakten Factory-Klasse vollständig angepasst werden bzw. eine neue konkrete Factory-Klasse entwickelt und der abstrakten Factory-Klasse durch Methoden bekannt gemacht werden.

Builder vs. Factory Pattern

Häufig stellt sich bei der Konzeption einer Anwendung die Frage, ob zur Erzeugung von Objekten besser das Builder oder das Factory Pattern verwendet werden sollte. In der Regel erweist sich die Verwen-

dung des Factory Patterns als vorteilhafter. Bei der Umsetzung des Builder Patterns muss nur der konkrete Erbauer bekannt sein, um mit diesem Entwurfsmuster Objekte zu erzeugen. Bei der Verwendung des Factory Patterns hingegen müssen alle Factory-Objekte bekannt sein, um Objekte zu erzeugen. Dies stellt bei einfachen, aber auch komplexeren Anwendungen kein Problem dar.

Um ein Builder Pattern zu verwenden, sollten die zu erzeugenden Produkte die gleiche Basis haben, d. h., es gibt eine gemeinsame Superklasse, die alle Standardinformationen enthält und von der die Produkte alle erben. Mit dem Factory Pattern können hingegen sehr einfach ganz unterschiedliche Objekte, die nicht über eine gemeinsame Superklasse miteinander verbunden sind, erzeugt werden. Dieser Vorteil kommt vor allem bei Verwendung der abstrakten Factory deutlich zum Tragen.

3.3 Singleton Pattern

Auch das *Singleton Pattern* (Einzelstück-Entwurfsmuster) gehört zu den GoF-Mustern. Es wird eingesetzt, um die Instanziierung einer Klasse auf genau eine Instanz zu beschränken.

In der SAP-Buchhaltung kann das Singleton Pattern z. B. dazu verwendet werden, ein Objekt, in dem allgemeine Informationen wie Buchungskreise und Kontenpläne zum Auslesen bereitstehen, für alle Klassen zur Verfügung zu stellen – so ähnlich wie eine globale Variable. Auf dieses Objekt kann jede Klasse in der Anwendung zugreifen und Informationen für alle Mitverwender zentral in dem Objekt ablegen. Beachten Sie jedoch, dass der Einsatz dieses Patterns nicht als Alternative zu globalen Variablen dienen sollte, sondern als Lösung für die Probleme, die durch den Einsatz von globalen Variablen entstehen können. Unter diese Probleme fallen z. B. unbeabsichtigte Variablenänderungen, Mehrfachreferenzen auf ein Objekt, eine schlechtere Wiederverwendbarkeit und Mehraufwand beim Testen.

3.3.1 Problem

Singuläres Objekt

Durch die Verwendung des Singleton Patterns in einer Anwendung können Sie sicherstellen, dass sich aus einer Klasse immer nur ein

konkretes Objekt instanziieren lässt. Durch weitere Instanziierungen (also weitere Aufrufe der Methode `erhalte_instanz()`) wird immer das gleiche Objekt zurückgegeben.

Die Verwendung des Singleton Patterns kann in SAP-Anwendungen vorteilhaft sein, da es sowohl in ABAP als auch in Web Dynpro ABAP den Ansatz der globalen Variablen nicht so gibt, wie man ihn aus anderen Programmiersprachen kennt.

Durch das Singleton-Pattern wird in ABAP und Web Dynpro ABAP ein ähnlicher Ansatz wie der der globalen Variablen zur Verfügung gestellt.

Durch die Verwendung des Singleton Patterns erhalten Sie dasselbe Objekt einer Klasse immer wieder, es kann statisch ausgelesen werden, und Sie können es direkt weiterverarbeiten. Es wird nur eine Instanz eines Objekts erzeugt und sichergestellt, dass auch nur eine Instanz verwendet wird.

3.3.2 Ansatz und Lösung

Singleton-Klasse

In dem Klassendiagramm in Abbildung 3.10 haben wir den Aufbau des Singleton Patterns skizziert. Das Pattern enthält eine einzige Klasse `Singleton`. In dem UML-Klassendiagramm stellt der Client die Anwendungsklasse dar bzw. die Methode, wie diese Klasse an unterschiedlichen Stellen in einer Anwendung integriert werden kann.

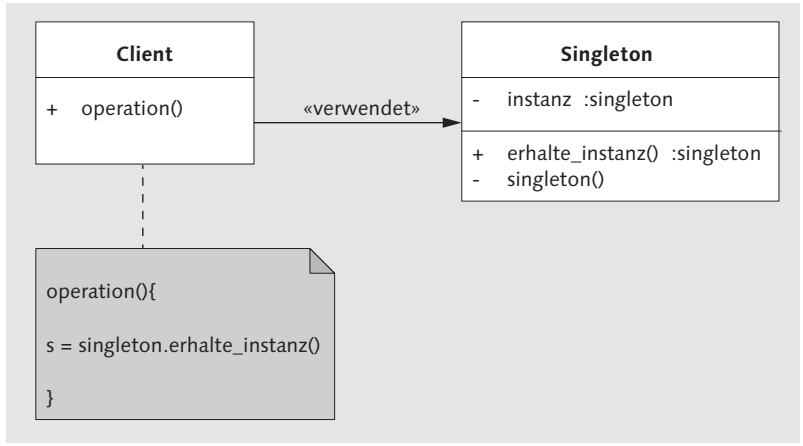

Abbildung 3.10 Singleton Pattern – UML-Klassendiagramm

Methode erhalte_instanz()

Die Singleton-Klasse besitzt das private Attribut `instanz`, das das einzige instanziierte Objekt der Singleton-Klasse beinhaltet. In dieser Klasse gibt es neben dem Konstruktor nur eine wichtige Methode, die Methode `erhalte_instanz()`. Beim Erzeugen der Klasse in ABAP erzeugt der Konstruktor ein Objekt und speichert es im privat definierten Attribut der Klasse ab, dem Attribut `instanz`. Durch den Initialisierungsaufruf der Methode erhält man dann das intern abgelegte Objekt der Klasse zurück.

Wird das Objekt erneut aufgerufen, kommt die Methode `erhalte_instanz()` zum Einsatz. Diese Methode gibt das Objekt der Klasse aus dem internen Attribut zurück. Ist die Klasse zuvor noch nicht initialisiert worden, wird das Objekt der Klasse erzeugt und intern in dem Attribut `instanz` abgelegt. Dieses Objekt erhält dann der Anwender zurück.

Das Singleton Pattern können Sie an jeder Stelle einer Software einsetzen, um Objekte über die Methode `erhalte_instanz()` aufzurufen. Sie sollten jedoch immer prüfen, ob die Informationen und Objekte zentral über ein eindeutiges Objekt zur Verfügung gestellt werden sollen. Ist dies der Fall, kann diese Anforderung für den Einsatz des Singleton Patterns sprechen. Ein sinnvoller Einsatzbereich des Entwurfsmusters ist es z. B., wenn eine Information an unterschiedlichen Stellen benötigt wird. Diese Information kann durch den Einsatz des Singleton Patterns zur Laufzeit geändert werden.

Ablauf

In Abbildung 3.11 haben wir den zeitlichen Ablauf des Erzeugungsprozesses mithilfe des Singleton Patterns veranschaulicht. Der Client ruft die statische Methode `erhalte_instanz()` der Singleton-Klasse auf und erhält die Referenz des Objekts sowie die Objektspeicheradresse, in diesem Fall 50A1, zurück. Ruft ein weiterer Anwender nochmals diese Methode auf, erhält dieser das exakt gleiche Objekt mit der gleichen Referenz zurück. Anwender 1 und 2 können auch verschiedene Programmabschnitte sein, die in einer Anwendung implementiert wurden. Der SAP NetWeaver Application Server ABAP unterstützt diesen Prozess nur bei einem Anwender und mehreren Programmabschnitten. Ein softwareübergreifendes Variablensystem kann leider nur durch Tabellen oder mithilfe ähnlicher Workarounds abgebildet werden. In ABAP gibt es dazu sogenannte *Shared Objects*, die immer für einen Benutzer in einer Session zur Verfügung stehen.

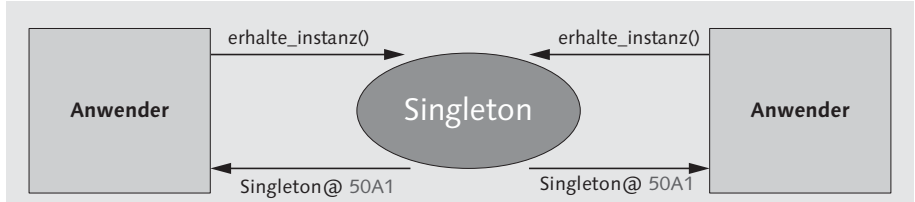

Abbildung 3.11 Singleton Pattern – Ablauf

3.3.3 Einsatzbeispiele

Kombination mit Façade Pattern

Das Singleton Pattern wird sehr oft in Kombination mit dem Façade Pattern verwendet, das wir Ihnen in Abschnitt 4.4 vorstellen. Das Façade Pattern benötigt in vielen Fällen immer wieder das gleiche Objekt. Dies wird durch das Singleton Pattern realisiert.

Anti Pattern

Bei der Verwendung in einem gewachsenen System, aber auch bei der Verwendung als globale Variable wird das Singleton Pattern häufig auch als Anti Pattern bezeichnet, weil es in diesen Fällen eher nachteilige Auswirkungen hat. Die Verwendung als globale Variable führt z. B. zum Problem der *Nebenläufigkeit*. Wenn mehrere Instanzen eines Programms gleichzeitig ablaufen, haben diese beim Einsatz globaler Variablen immer nur Zugriff auf den momentan gültigen Wert der Variablen, auch wenn dieser Wert sich z. B. auf der Datenbank ändert. Beim Singleton Pattern wird allen Instanzen des Programms derselbe Wert bereitgestellt, auch wenn dieser während der Laufzeit eine Änderung erfahren hat. Alle Instanzen nutzten also das gleiche Objekt über die Methode `erhalte_instanz()`.

Ob die Verwendung des Singleton Patterns sinnvoll ist, hängt davon ab, ob die Systeme durch die Globalisierung der Objekte beim Anwenden des Entwurfsmusters effizienter werden oder nicht. Wird dieses Entwurfsmuster in eine Anwendung implementiert und nur einmal verwendet, entsteht dadurch kein Mehrwert für die Entwickler und die Performance.

Beispiel: Data Store Objects

In ABAP und Web Dynpro wird dieses Entwurfsmuster sehr oft angewendet, weil dadurch viele Laufzeitvariablen erfasst und dann zwischengespeichert werden können. Ein häufiger Verwendungszweck besteht z. B. im Kontext zentral gespeicherter Einstellungen. Dabei werden zentrale *Data Store Objects* (DSO) auf der Datenbankschicht im System angelegt, in denen alle wichtigen Parameter einer

Anwendung konfiguriert werden können. In solchen DSOs sollten nur Bewegungsdaten gespeichert werden. Andere Informationen sollten in Datenbanktabellen abgespeichert werden.

In dieser Anwendung kommt nun das Singleton Pattern zum Einsatz. Es lädt alle Parameter beim Aufruf der Anwendung und speichert diese intern als eigenes Objekt ab. So kann bei einem erneuten Aufruf der Parameter viel schneller auf diese Variablen zugegriffen werden, als wenn man bei jedem Aufruf auf die Datenbankschicht zugreifen und die Parameter dort über eine `SELECT`-Anweisung abrufen müsste.

Beispiel: Artikeldefinition

Ein weiteres Beispiel wäre eine unternehmensinterne Web-Dynpro-Anwendung, über die unterschiedliche Artikel, z. B. Büromaterialien, bestellt werden können. Diese werden zunächst definiert, und anschließend wird ein Bestellformular ausgegeben. Das Singleton Pattern kann hier dazu eingesetzt werden, die verschiedenen Artikel in der Singleton-Klasse in einer Objektliste als Attribute der Klasse zu sammeln. Auf diese zentrale Liste kann jede Klasse zugreifen und hat so alle aktuellen Artikel zur Verfügung, die als Objekt bereits zu der Liste hinzugefügt worden sind.

Als Programmierer müssen Sie so nicht umständlich den Web-Dynpro-Kontext verwenden, um alle Details der Artikel abzuspeichern; die Anwendung kann daher viel einfacher umgesetzt werden. Stattdessen sammeln Sie alle Informationen zu den Artikeln als Objekte in der Tabelle in der Singleton-Klasse. Beim Weitergeben der Bestellung, also des Singleton-Objekts, das über die Methode `erhalte_instanz()` ausgelesen werden kann, können Sie die Objekte der internen Liste einfach abrufen und verarbeiten. Das Hinzufügen der Artikel in das Singleton-Objekt kann durch diesen Ansatz an verschiedenen Stellen im Quellcode flexibel erfolgen.

Mehrfachinstanzen verhindern

Als Anwendungszweck besonders hervorzuheben ist die Verhinderung von Mehrfachinstanzen von Objekten. Dies kann zwar theoretisch auch über statische Attribute realisiert werden. Durch das Singleton Pattern wird jedoch verhindert, dass mehrere Objekte einer Klasse erzeugt werden.

Null-Referenz verhindern

Darüber hinaus kann auch eine Null-Referenz verhindert werden. Wenn bei einem Zugriff auf das Entwurfsmuster noch kein Objekt

vorhanden ist, wird dies automatisch erzeugt und erst danach dem Anwender zurückgegeben.

3.3.4 Umsetzung in ABAP

Zunächst wird ein ABAP-Report benötigt, den Sie in Transaktion SE38 erzeugen können:

```
REPORT zdp_ar_erz_singleton.
```

In diesem ABAP-Report rufen wir später die Entwurfsmusterimplementierung auf, in der verschiedene Klassen realisiert sind.

Singleton-Klasse implementieren

Im ersten Schritt wird die Klasse `ZDP_CL_SING_SINGLETON` erzeugt (siehe Listing 3.15). Die einzige Methode dieser Klasse, `ERHALTE_INSTANZ()`, gibt das initialisierte Objekt zurück, das in der Klasse gespeichert wurde. Ist die Klasse noch nicht initialisiert worden, wird automatisch ein neues Objekt der Klasse erzeugt und in das interne Attribut `LO_INSTANZ` abgespeichert.

```
CLASS zdp_cl_sing_singleton DEFINITION
  PUBLIC   FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS constructor.
  PUBLIC SECTION.
    METHODS constructor.
    CLASS-METHODS erhalte_instanz
      RETURNING
        VALUE(r_instanz) TYPE REF TO zdp_cl_sing_singleton.
  PROTECTED SECTION.
  PRIVATE SECTION.
    CLASS-DATA lo_instanz TYPE REF TO zdp_cl_sing_singleton.
ENDCLASS.

CLASS zdp_cl_sing_singleton IMPLEMENTATION.
  METHOD constructor.
  ENDMETHOD.
  METHOD erhalte_instanz.
    IF lo_instanz IS INITIAL.
      CREATE OBJECT lo_instanz.
    ENDIF.
    r_instanz = lo_instanz.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.15 Singleton Pattern – Definition und Implementierung der Singleton-Klasse

Klasse zum Aufruf des Patterns

Um das Singleton Pattern verwenden zu können, implementieren wir noch die Klasse ZDP_CL_SING_APPLIKATION. Dies ist die Client-Klasse, die definiert, wie dieses Entwurfsmuster später in einer Anwendung implementiert werden kann. Alternativ ist es allerdings auch möglich, diese Klasse wegzulassen und den Quellcode direkt in den ABAP-Report zu schreiben.

Die Klasse ZDP_CL_SING_APPLIKATION dient in unserem Beispiel zur Verdeutlichung, wie der Aufruf in einer Klasse ihres Reports implementiert werden kann. Die Klasse beinhaltet die Methode START(), die das Entwurfsmuster aufruft (siehe Listing 3.16). Um zu zeigen, dass es sich bei den erzeugten Objekten immer um das gleiche Objekt handelt, werden die beiden Objekte LO_OBJEKT1 und LO_OBJEKT2 abgespeichert und dann miteinander verglichen.

```
CLASS zdp_cl_sing_applikation DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS start.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_sing_applikation IMPLEMENTATION.
  METHOD start.
    DATA: lo_objekt1 TYPE REF TO zdp_cl_sing_singleton.
    DATA: lo_objekt2 TYPE REF TO zdp_cl_sing_singleton.

    lo_objekt1 = zdp_cl_sing_singleton=>erhalte_instanz( ).
    lo_objekt2 = zdp_cl_sing_singleton=>erhalte_instanz( ).

    IF lo_objekt1 = lo_objekt2.
      WRITE: / 'Das Objekt hat die gleiche Instanz.'.
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.16 Ausführung des Singleton-Entwurfsmusters

Methode START aufrufen

Um den ABAP-Report ZDP_AR_ERZ_SINGLETON ausführen zu können, sehen wir die in Listing 3.17 aufgeführtem Codezeilen in dem ABAP-Report vor.


```
REPORT zdp_ar_erz_singleton.
DATA lo_app TYPE REF TO zdp_cl_sing_applikation.
CREATE OBJECT lo_app.
lo_app->start( ).
```

Listing 3.17 Singleton Pattern – Aufruf im ABAP-Report

Das Ergebnis dieser Implementierung erhalten Sie durch die Ausführung des ABAP-Reports ZDP_AR_ERZ_SINGLETON über Transaktion SE38. Dabei wird die Ausgabe wie in Abbildung 3.12 angezeigt:

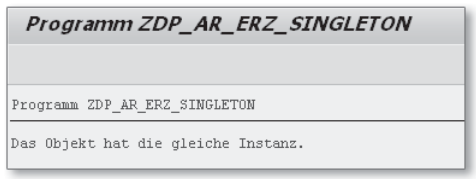

Abbildung 3.12 Singleton Pattern – Reportausgabe

Die Prüfung, ob die Objekte LO_OBJEKT1 und LO_OBJEKT2, die in der Klasse ZDP_CL_SING_APPLIKATION erzeugt werden, gleich sind, verlief erfolgreich. Diese Objekte haben die gleiche Referenz, und daher ist auch ihr Inhalt komplett identisch.

3.3.5 Evaluation

Das Singleton Pattern kann sehr vielseitig und auch im Rahmen der Implementierung anderer Erzeugungsmuster verwendet werden. Es hilft Entwicklern dabei, Aufgaben wie das Schreiben einer Protokolldatei oder das Durchlaufen von Warteschleifen durch mehrere Prozesse zu vereinheitlichen. Durch das Entwurfsmuster kann einfach immer das gleiche Objekt für die entsprechende Aufgabe bereitgestellt und verwendet werden. Was für ein Objekt das ist, ist abhängig vom Anwendungsfall.

Globale Variablen

Globale Variablen sind bei der Anwendungsentwicklung immer ein wichtiges Thema, um Informationen zentral und für alle Klassen, Methoden und Objekte zur Verfügung zu stellen. Das Singleton Pattern verfolgt einen objektbasierten Ansatz zur Verwendung globaler Variablen. Es kann sehr einfach klassenübergreifend verwendet werden, ohne dabei spezielle ABAP-Konstrukte verwenden zu müssen, um es verfügbar zu machen. Diese ABAP-Konstrukte wären sehr aufwendig und fehleranfällig in der Implementierung.

Vorteile

Die Vorteile des Singleton Patterns lassen sich wie folgt zusammenfassen:

- **Einfachheit**
  Eine Singleton-Klasse ist sehr einfach und schnell entwickelt und hat gegenüber den globalen Variablen insofern einen erheblichen Vorteil, als sie flexibel erweiterbar und vielfältiger einsetzbar ist. Neben Variablen können auch Objekte in der Singleton-Klasse abgespeichert werden. Darüber hinaus ist dieses Entwurfsmuster in der Anwendung besser und übersichtlicher handhabbar, als es globale Variablen sind.

  Im Kundennamensraum erhalten Sie durch die Verwendung dieses Entwurfsmusters einen deutlich übersichtlicheren ABAP-Code. Es sind keine globalen Variablen vorhanden, die überall verwendet werden, sondern alle globalen Variablen sind in der Singleton-Klasse gekapselt. Sie können z. B. zehn oder 20 globale Variablen erstellen, die entsprechend den Namenskonventionen alle mit dem Präfix `gv` anfangen. Bei der Verwendung des Singleton Patterns wird dagegen nur ein Objekt angelegt, das alle globalen Variablen enthält, auf das sehr einfach z. B. über die Bezeichnung `singletonobjekt-variable1` zugegriffen werden kann.
- **Zugriffssicherheit**
  Ein weiterer wichtiger Aspekt ist der Zugriffsschutz. Das Entwurfsmuster kapselt seine eigene Erstellung in sich selbst. Dabei wird das Objekt der Klasse instanziiert und in sich selbst abgespeichert. Darüber hinaus kann bei einer Eingabe in die Variable standardisiert durch einen Methodenaufruf eine Plausibilitätsprüfung durchgeführt werden, die prüft, ob die gesetzten Werte korrekt sind und weiterverwendet werden können. Dies kann direkt in dem Entwurfsmuster realisiert werden.
- **Ableitungen**
  Von der Singleton-Klasse können Sie auch weitere Ableitungen erstellen, sodass typspezifische Variablen zur Verfügung stehen, die während der Laufzeit gefüllt werden. Erst dann wird auch entschieden, welchen Datentyp sie erhalten.

  Zum Beispiel sind in der Auftragsklasse die typspezifischen Variablen für die Auslagerung und die Einlagerung enthalten. Während

der Laufzeit kann entschieden werden, welche Objekte für diese zwei Variablen erzeugt werden sollen. In der Definition enthalten diese beiden Variablen generell die Definition eines Interfaces.

- **Lazy Loading**
  Sie können die Erzeugung der Singleton-Klasse immer nur dann anstoßen, wenn sie auch benötigt wird – dies nennt man auch Lazy Loading.

Einschränkungen

Auch bei diesem Pattern gibt es Einschränkungen für die Verwendbarkeit. Wenn das Pattern in einer Anwendung zu häufig eingesetzt wird, leidet deren Übersichtlichkeit. Dies ist vor allem beim objektorientierten Programmieren in ABAP der Fall, weil dadurch das Entwurfsmuster an vielen Stellen mit den gleichen Abfragen verwendet werden kann.

Ein anderer Nachteil ist, dass bei der Anwendung des Singleton Patterns unterschiedliche Klassen entstehen, die jeweils für bestimmte Objekte gültig sind. Daher muss die Zuordnung der zu verwendenden Pattern-Klasse klar definiert werden, da es sonst zu Inkonsistenzen kommen kann. Die Transparenz und die Wartbarkeit der Anwendung werden infolgedessen eingeschränkt.

Im Gegensatz zu anderen Entwurfsmustern sind die Möglichkeiten zur Wiederverwertung sehr eingeschränkt. Implementieren Sie viele unterschiedliche Singleton-Klassen, kann die Performance der gesamten Anwendung leiden, da diese implementierten Klassen für alle entsprechenden Objekte erzeugt und bereitgestellt werden müssen.

Beim Entwurfsmuster stehen besonders die Synchronität der Daten und die Zentralisierung an einer Stelle (Flaschenhalsprinzip) im Vordergrund. Diese Prinzipien können durch das Pattern einfach und übersichtlich umgesetzt werden.

3.4 Prototype Pattern

Das *Prototype Pattern* gehört ebenso wie die drei anderen vorgestellten Erzeugungsmuster zu den GoF-Mustern.

3.4.1 Problem

Kopie einer Vorlage

Das Prototype Pattern wird eingesetzt, um Objekte auf Basis von Vorlagen, d. h. *Prototypen* einer Instanz, zu erzeugen. Dabei werden die vorhandenen Instanzen der Objekte kopiert, und das neue Objekt wird an die neuen Bedürfnisse angepasst. Innerhalb des Prototype Patterns kann nicht nur ein Objekt, sondern es können unterschiedlich viele Objekte kopiert werden, die dazu dem Entwurfsmuster nicht bekannt sein müssen.

3.4.2 Ansatz und Lösung

Klassendiagramm

Abbildung 3.13 skizziert das Prototype Pattern in einem UML-Klassendiagramm. Das Diagramm enthält die Klasse `prototype_factory`, die dieses Entwurfsmuster repräsentiert. Diese Klasse wird über den Singleton-Ansatz realisiert, da immer die gleiche Instanz der Klasse verwendet werden kann.

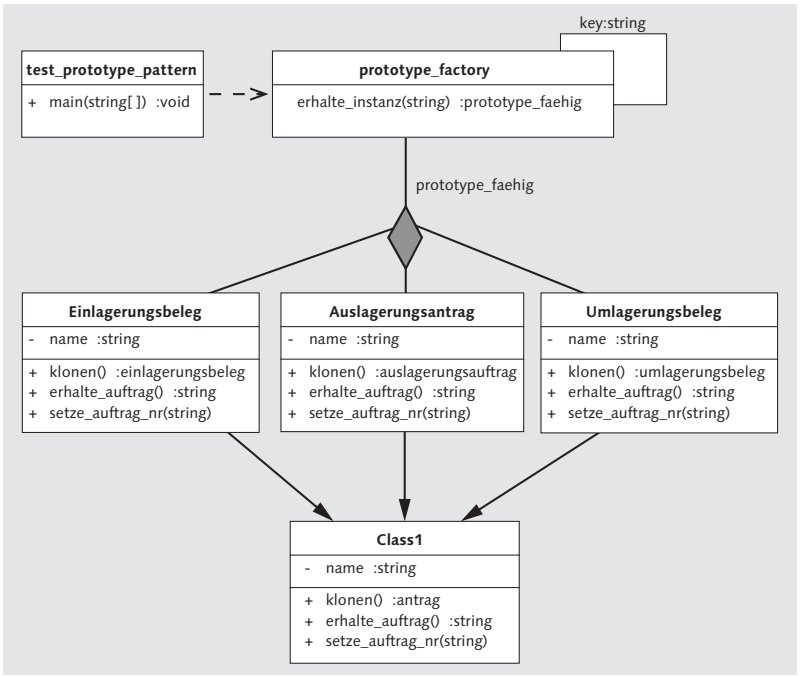

Abbildung 3.13 Prototype Pattern – UML-Klassendiagramm

Um das Prototype Pattern funktionsbereit zu machen, wird eine Implementierung in den unterschiedlichen Objektklassen des Interfaces `prototype_faehig` benötigt. Dieses Interface stellt in den einzelnen Objekten die standardisierte Methode `klonen()` bereit. Durch diese Methode hat der Entwickler die Möglichkeit, das Objekt zu duplizieren. Dieses duplizierte Objekt erhält er als Rückgabeparameter von der Methode zurück.

Um das Entwurfsmuster zu initialisieren, rufen Sie die statische Methode `erhalte_instanz()` der Klasse `prototype_factory` auf, über die Sie das Objekt erhalten.

3.4.3 Einsatzbeispiele

Teure Neuentwicklungen vermeiden

Das Prototype Pattern kann vor allem dann gut in Anwendungen integriert werden, wenn Neugenerierungen von Objekten sehr aufwendig, d. h. teuer, wären. Dies ist der Fall, wenn Objekte sehr viele Abhängigkeiten haben, die bei einer vollständigen Neuerzeugung ebenfalls neu definiert werden müssten. Eine wichtige Voraussetzung für die Verwendung des Prototype Patterns ist, dass das Ursprungsobjekt dem neuen Objekt ähnelt.

Factory-Hierarchien ignorieren

Hinter einigen Objekten können sich Hierarchien von Factory-Klassen verbergen, die parallel zueinander bzw. gegeneinander arbeiten. Das Prototype Pattern kann die Hierarchien einfach ignorieren und das Objekt so kopieren, dass keine neuen Hierarchien entstehen.

Dynamischer Einsatz

Ein weiterer Vorteil entsteht, wenn die Klassen – bzw. die Objekte dieser Klassen – erst während der Laufzeit bekannt sind. Dank dieser dynamischen Programmierweise können Sie z. B. Vorlagenobjekte erzeugen, die durch das Prototype Pattern kopiert und dann erst ausgeprägt werden.

Beispiel: Workflow

In SAP-Systemen wird das Prototype Pattern häufig in komplexen Konstrukten angewendet. Beispiel dafür ist eine Workflow-Anwendung, die unterschiedliche Stadien durchlaufen muss. So kann z. B. eine Auftragsvorlage dafür verwendet werden, einen speziellen Auftrag zu erstellen, z. B. einen Auftrag zur Auslagerung eines Artikels aus einem Lager, wie in unserer Beispielanwendung in Kapitel 7, »Flexible Entwicklung einer Beispielanwendung«, beschrieben.

Das Entwurfsmuster wird hier dazu eingesetzt, die Vorlage, die durch den Entwickler definiert wurde, zu duplizieren und dadurch den Auftragsprototyp dem Anwender für die weitere Verarbeitung zur Verfügung zu stellen. Durch die Anlage des konkreten Auftrages

wird dann ein Workflow in der SAP-Anwendung gestartet, der letztlich zur Auslagerung des Artikels führt. In diesem Beispiel kann nicht nur der Standardauftrag zur Erzeugung eines individuellen Auftrags kopiert werden, sondern auch bereits vorhandenen Aufträgen können abhängige Objekte, wie z. B. auszulagernde Artikel, Genehmigungen oder Bearbeitungshistorien, angehängt werden.

3.4.4 Umsetzung in ABAP

ABAP-Report und Klassendefinition

Genau wie die bereits beschriebenen Erzeugungsmuster wird auch das Prototype Pattern in einem ABAP-Report ausgeführt, den Sie in Transaktion SE38 anlegen:

```
REPORT zdp_ar_erz_prototype.
```

Das Prototype Pattern basiert auf der abstrakten Klasse `ZDP_CL_PTYPE_PROTOTYPE`. Diese Klasse hat zwei Methoden, die notwendig sind, um Objekte zu kopieren. Im Codebeispiel in Listing 3.18 erhält das Objekt über die Konstruktor-Methode eine eindeutige ID, die sich nicht ändert, wenn ein Objekt geklont wird. Über die Methode `ERHALTE_ID()` kann diese ID abgefragt werden.

```
CLASS zdp_cl_ptype_prototype DEFINITION
  PUBLIC   ABSTRACT
  CREATE PUBLIC.
  PUBLIC SECTION.
    INTERFACES if_os_clone.
    METHODS constructor
      IMPORTING
        !i_id TYPE string.
    METHODS duplizieren
      RETURNING
      VALUE(r_prototype) TYPE REF TO zdp_cl_ptype_prototype.
    METHODS erhalte_id
      RETURNING
        VALUE(r_id) TYPE string.
  PROTECTED SECTION.
  PRIVATE SECTION.
    DATA id TYPE string.
ENDCLASS.

CLASS zdp_cl_ptype_prototype IMPLEMENTATION.
 METHOD constructor.
    me->id = i_id.
  ENDMETHOD.
  METHOD erhalte_id.
    r_id = me->id.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.18 Prototype Pattern – abstrakte Prototyp-Klasse

Konkrete Prototyp-Klasse

Die Implementierung eines konkreten Prototyps findet in der eigenen Klasse `ZDP_CL_PTYPE_KONTYPE1` statt. Diese Klasse erbt von der abstrakten Prototyp-Klasse. Durch diese Vererbung sind die Implementierungen der Methode `ERHALTE_ID()` sowie des Konstruktors in der konkreten Klasse bereits vorhanden. Zusätzlich muss noch die Methode `DUPLIZIEREN()` implementiert werden. Wie Sie in Listing 3.19 sehen, wird dabei das Objekt der Klasse zurückgegeben.

```
CLASS zdp_cl_ptype_konptype1 DEFINITION
  PUBLIC
  INHERITING FROM zdp_cl_ptype_prototype
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS duplizieren
         REDEFINITION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_ptype_konptype1 IMPLEMENTATION.
  METHOD duplizieren.
    r_prototype ?= me->if_os_clone~clone( ).
  ENDMETHOD.
ENDCLASS.
```

Listing 3.19 Prototype Pattern – Implementierung der konkreten Prototyp-Klasse

Klasse zum Aufruf des Patterns

Um das Prototype Pattern verwenden zu können, wird noch die weitere Klasse `ZDP_CL_PTYPE_APPLIKATION` benötigt, die angibt, wie dieses Entwurfsmuster in einer Web-Dynpro-Anwendung eingesetzt werden soll. Sie können diese Klasse allerdings auch weglassen und den Quellcode direkt in den ABAP-Report schreiben.

In der Methode `START()` wird zu Beginn das Objekt `LO_P1` des konkreten Prototyps erstellt. Dieses Objekt hat die `I_ID = 1`. Wird nun die Methode `DUPLIZIEREN()` aufgerufen, wird ein neues Objekt auf Basis des Originalobjekts erzeugt. Rufen Sie vor dem Aufrufen der Methode `DUPLIZIEREN()` die Methode `ERHALTE_ID()` auf, erhalten Sie

die aktuelle ID des Objekts. Im Beispiel aus Listing 3.20 ist dies `I_ID = 1`. Nach dem Klonen erhält das neue Objekt die gleiche ID wie das ursprüngliche Objekt.

```
CLASS zdp_cl_ptype_applikation DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
    METHODS start.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zdp_cl_ptype_applikation IMPLEMENTATION.
  METHOD start.
    DATA: lo_p1 TYPE REF TO zdp_cl_ptype_konptype1.
    DATA: lo_c1 TYPE REF TO zdp_cl_ptype_konptype1.
    DATA: lv_id TYPE string.

    CREATE OBJECT lo_p1
      EXPORTING
        i_id = 'id=1'.
    lv_id = lo_p1->erhalte_id( ).
    WRITE: / 'Original-Objekt: {0}', lv_id.
    lo_c1 ?= lo_p1->duplizieren( ).
    lv_id = lo_c1->erhalte_id( ).
    WRITE: / 'Geklontes Objekt: {0}', lv_id.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.20 Prototype Pattern – Umsetzung des Entwurfsmusters in der aufrufenden Klasse

Start-Methode

Um den ABAP-Report `ZDP_AR_ERZ_PROTOTYPE` ausführen zu können, sehen wir in Listing 3.21 aufgeführte Codezeilen im ABAP-Report vor.

```
REPORT zdp_ar_erz_prototype.
DATA lo_app TYPE REF TO zdp_cl_ptype_applikation.
CREATE OBJECT lo_app.
lo_app->start( ).
```

Listing 3.21 Prototype Pattern – Aufruf im ABAP-Report

Das Ergebnis dieser Implementierung erhalten Sie durch die Ausführung des ABAP-Reports `ZDP_AR_ERZ_PROTOTYPE` über Transaktion SE38. Dabei wird die Ausgabe wie in Abbildung 3.14 angezeigt:

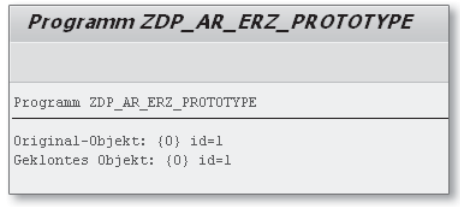

Abbildung 3.14 Prototype Pattern – Reportausgabe

Die Zeilen Original-Objekt: {0} id=1 und Geklontes Objekt: {0} id=1 werden durch den ABAP-Report ausgegeben. Man erhält also mit jeder Referenz auf ein Objekt das gleiche Objekt, ohne dass das ursprüngliche Objekt dazu verändert wurde.

3.4.5 Evaluation

Rapid Deployment

Das Prototype Pattern kann nur sehr begrenzt eingesetzt werden, und zwar vor allem, um schnell einfache Lösungen zu entwickeln (*Rapid Deployment*), die z. B. für unterschiedliche Tests benötigt werden. Dabei ist es nicht relevant, ob es sich um komplexe oder einfach strukturierte Objekte handelt. Es ist jederzeit möglich, auch während der Laufzeit noch weitere Unterklassen zu erzeugen. So kann eine hohe Flexibilität bei der Variation der Objekte und deren Strukturen erzielt werden. Dabei wird keine Erzeuger-Klassenhierarchie parallel zu einer Klassenhierarchie der Objekte aufgebaut.

Aufwand bei komplexen Objekten

Der Nachteil des Prototype Patterns ist, dass die Kopierfunktion für jedes einzelne Objekt implementiert werden muss. So ist es zwar sehr schnell und einfach möglich, das Entwurfsmuster bei einzelnen Objekten umzusetzen, bei Objekten mit sehr vielen und differierenden Unterklassen ist die Implementierung aber mit einem größeren Aufwand verbunden.