Semesterarbeit

Im Studiengang Wirtschaftsinformatik der FH Nordakademie

Implementierung ausgewählter Entwurfsmuster in Delphi

von Heiko Behrens (1787-I01a)

Im Rahmen der Vorlesungsreihe Moderne Softwaretechnologien
bei Herrn Prof. Dr. Johannes Brauer

September 2005

Erklärung zur Semesterarbeit

Ich erkläre, dass ich die vorliegende Semesterarbeit in vollem Umfang selbstständig erarbeitet und verfasst habe. Sämtliche wörtlich oder sinngemäß wiedergegebenen Quellen oder sich an den Gedankengängen anderer Autoren anlehnende Ausführungen sind durch Quellenangaben als solche gekennzeichnet.

Neumünster, 19. September 2005

Heiko Behrens

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1. Einleitung

Ein *Muster* ist, gemäß Christopher Alexander nach einer Übersetzung von Dirk Riehle [ALEXANDER1997], „ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen". Im Jahr 1987, zehn Jahre nach ihrer Veröffentlichung, berichtet Kent Beck in einer kleinen Arbeitsgruppe der OOPSLA[1] von dieser Arbeit und sieht Zusammenhänge zwischen der Errichtung von Brücken und Gebäuden und der Herstellung von Software [RIEL1996]. Die Art und Weise der Darstellung von Ähnlichkeiten komplexester, weltweit verteilter Konstruktionen zueinander, sollte gleichfalls auf die Systeme der objektorientierten Programmierung angewendet werden können.

Heute bilden Muster dieser Art vielerorts die Grundlage eines soliden Entwurfs bei der Erstellung von Software und sind aus dem täglichen Gebrauch zum Austausch von Ideen auf Konferenzen und in der Literatur nur schwer wegzudenken. Sie bilden damit zweifellos eines der nützlichsten und erfolgreichsten Konzepte der gegenwärtigen Softwareentwicklung.

1.1. Entwurfsmuster

Seit Anbeginn der Softwaretechnik, der ingenieurmäßigen Herstellung von Software und den damit verbundenen Prozessen, versuchen Programmierer und Architekten ihre Ergebnisse so wirkungsvoll wie möglich zu erzielen und Quelltexte wie Entwürfe wieder zu verwenden. Um dieses Ziel zu erreichen, wird im Allgemeinen eine flexible Struktur gewählt, die Pflege und Fehlerbehebung ebenso vereinfacht wie die Erweiterung des erstellten Systems. Über die Zeit gelang es Experten, ihr Wissen und ihre Erfahrungen bei der Konstruktion solcher Softwarelösungen in abstrakter Form zu beschreiben, um es anderen Spezialisten als so genannte *Entwurfsmuster* verfügbar zu machen.

[1] Jährliche Konferenz zur Softwaretechnik mit dem Titel “Object-Oriented Programming, Systems, Languages, and Applications”

Muster als Herangehensweise und Umsetzung von Strukturen bei der Erstellung von computerbasierten Systemen sind allgegenwärtig und existieren seit jeher. Um einmal angewendete Lösungen wieder verwenden zu können, ist es jedoch notwendig, ihren Zweck und ihre Form isoliert darzustellen. Ein Entwurfsmuster in der Softwaretechnik kann dabei als eine Sammlung von Erkenntnissen zusammen mit einer Art von Sprache verstanden werden, die gemeinsam ein typisches Problem bei der Entwicklung von Software in Kombination mit einer Lösung beschreiben.

Zu den bekanntesten Darstellungen solcher Ansätze gehört neben [FOWLER2002], [BUSCHMANN1996] und [SCHMIDT2000] wohl das Werk „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“ [GAMMA1996] von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Die Autoren beschreiben vor dem Hintergrund objektorientierter Softwareentwicklung 23 Entwurfsmuster, die in der Literatur wie auch in der Lehre vor allem aber in die Praxis Einzug gefunden haben. Längst gehören „Dekorierer“, „Besucher“ oder „Fabriken“ zum Repertoire der Konstruktionswerkzeuge eines objektorientierten Softwareentwicklers und haben Einzug in die Ausdrucksmittel von Veröffentlichungen und Vorträgen gefunden.

1.2. Ziel dieser Arbeit

Bereits zur Einführung des Themas beschreibt [GAMMA1996] die Abhängigkeit eines Entwurfsmusters von der eingesetzten Programmiersprache. Zwar stellen Muster generelle und vor allem sprachübergreifende Strukturen dar, die Form der gewählten Elemente zur Erlangung eines angestrebten Ziels hängt jedoch stark von äußeren Randbedingungen ab. Die Autoren wählten die Sprachen C++ und Smalltalk aus pragmatischen Gründen, weil sie ihre täglichen Erfahrungen widerspiegelten. Während sich die vorgestellten Muster in jeder „handelsüblichen objektorientierten Programmiersprache“ [GAMMA1996] abbilden ließen, sind sich die Verfasser des Katalogs sehr wohl bewusst, dass „man manche Muster leicher in der einen als der anderen Sprache umsetzen kann“ [GAMMA1996].

Sieht man von Veröffentlichungen in elektronischer Form ab, ist die Literatur mit [KLEINER2003], als einziges Buch zum Thema „Entwurfsmuster“ im Zusammenhang mit der Programmiersprache Delphi [Delphi], dünn gesät. Die

Veröffentlichungen zu dieser auch als „ObjectPascal“ bekannten, objektorientierten Sprache auf der Basis von Pascal [WIRTH1971], beschäftigen sich dabei grundsätzlich mit der Realisierung auf syntaktischer Ebene. Auch [KLEINER2003] vernachlässigt die Spezifika der Plattform und stellt damit nicht viel mehr zur Verfügung als das, was ein Entwickler, der sich mit Entwurfsmustern beschäftigt, ohnehin bewältigen können sollte: Die Codierung einer als Klassendiagramm vorliegenden Struktur.

Diese Arbeit befasst sich mit der Umsetzung einer ausgewählten Menge von Entwurfsmustern unter Berücksichtigung der Besonderheiten der Programmiersprache Delphi und den 32-Bit-basierten Betriebssystemen von Microsoft als Zielplattform. Mit den Mustern „Iterator“, „Fabrik“ und „Befehl“ werden Lösungen vorgestellt, die den Unzulänglichkeiten eines fehlenden Garbage Collectors entgegenwirken, das Metaklassenmodell von Delphi einsetzen und Methodenzeiger zur Vermeidung von Unterklassenbildung verwenden. Die Quelltextbeispiele im Anhang demonstrieren nicht nur die Implementierung der vorgestellten Verfahren, sondern zeigen mit den vorliegenden Testklassen auch den Gebrauch der vereinfachten Strukturen. Auf diese Weise dient die Arbeit zugleich als Vorlage für die Implementierung von Delphi-basierter Software als auch zur Inspiration, sich mit den Spezifika einer Programmiersprache auseinander zu setzen, um den Nutzen von Entwurfsmustern weiter zu erhöhen und ihre Umsetzung zu vereinfachen.

2. Iterator

Das Muster *Iterator* nach [GAMMA1996] beschreibt ein Objekt, das den sequentiellen Zugriff auf ausgewählte Teile oder Elemente eines umgebenden Objekts zulässt. Typischerweise handelt es sich bei diesen umgebenden *Aggregaten* um Listen, Bäume oder andere Strukturen zur Abbildung von geordneten oder ungeordneten Mengen.

2.1. Struktur und Verwendung

Ein *Iterator* kann als Referenz auf ein Element des *Aggregats* verstanden werden, das über eine Operation zum Dereferenzieren hinaus die Möglichkeit bietet, vom betrachteten auf das nachfolgende Element zu schließen, sofern ein solches vorhanden ist. Die Logik zum Iterieren über die Struktur des *Aggregats* wird mithilfe eines *Iterators* externalisiert. Operationen des *Aggregats* zur Bekanntmachung seiner internen Beschaffenheit werden somit entbehrlich und die grundsätzlich unterschiedlichen Aspekte der Beherbergung und dem sequentiellen Anbieten von Elementen werden in unterschiedliche Klassen aufgeteilt.

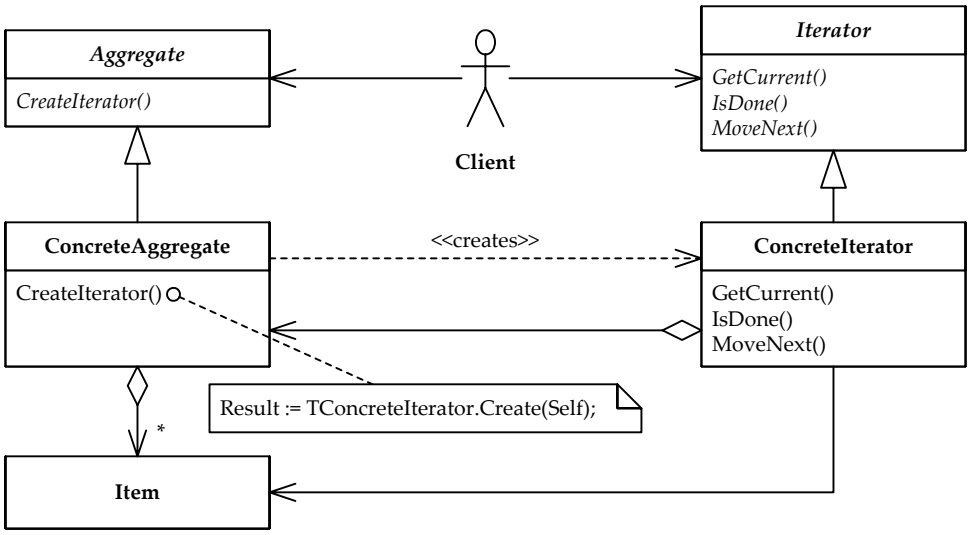

Abbildung 2-1: Struktur des Musters „Iterator“ nach [GAMMA1996]

Während dem Klienten aus Abbildung 2-1 lediglich die abstrakte Klasse Iterator und dessen Operationen bekannt sind, implementiert ConcreteIterator die letztlich verwendete Logik zum Iterieren über ein konkretes *Aggregat*. Ohne das Wissen um

die eigentliche Klasse des verwendeten *Iterators* können geeignete Exemplare mit der Methode CreateIterator() eines *Aggregats* angelegt werden. Der Gebrauch dieser speziellen *Fabrikmethode* [GAMMA1996] führt so zu zwei Vererbungshierarchien, eine für konkrete *Aggregate* und eine für geeignete *Iteratoren*.

Der Einsatz des Iterator-Musters erlaubt es daher, Zugriff auf enthaltene Objekte zu erlangen, ohne die interne Struktur des Aggregats offen zu legen oder spezielle Algorithmen durch Klienten mehrfach implementieren zu lassen. Iteratoren erlauben die gleichzeitige, mehrfache Betrachtung desselben *Aggregats*, weil die Zustände der *Iteratoren* voneinander unabhängig sind, und bieten eine einheitliche Schnittstelle unterschiedlichster Traviersierungsmechanismen auch bei Zusammengesetzten Strukturen über das *Kompositionsmuster* [GAMMA1996].

Iteratoren können zusätzliche Restriktionen verwirklichen, um sicherzustellen, dass auf Elemente nur einmalig zugegriffen werden kann oder einem speziellen Kriterium entsprechen. Durch die uniforme Schnittstelle eignen sie sich hervorragend, um über einen *Dekorierer* [GAMMA1996] um zusätzliches Verhalten ergänzt zu werden. Ein Invertieren von Folgen ist so ebenso generisch möglich, wie das Filtern von Elementen, ohne dass der Klient angepasst werden muss.

Die zwei grundsätzlichen Mechanismen zum Iterieren eines Aggregats unterteilt [BOOCH1994] in aktive und passive Verfahren. Abbildung 2-1 stellt mit dem *externen Iterator* eine aktive Lösung vor, bei der es die Aufgabe des Klienten ist, die Verfügbarkeit von weiteren Elementen mit IsDone() zu erfragen oder mithilfe von MoveNext() zum nächsten Element zu gelangen. Bei einem passiven Verfahren obliegt diese Kontrolle einem *internen Iterator*, der einem Empfänger jedes Element in einem gesonderten Aufruf zur Verfügung stellt, wie es mit den Standardklassen für Mengen in Smalltalk mit der Nachricht do: gelöst ist. Aspekte wie die Robustheit eines *Iterators* oder die Kopplung zwischen *Iterator* und *Aggregat* hängen unmittelbar von der Wahl zwischen *internem* und *externem Iterator* ab. Die nachfolgende Lösung beschreibt, entsprechend dem durch [GAMMA1996] vorgestellten Verfahren, die Lösung *externer Iteratoren*.

Durch die Abstraktion vom Algorithmus zur Traversierung sind weiterhin spezielle *Iteratoren* denkbar, die eine leere Menge oder eine algorithmisch erzeugte Folge repräsentieren. Die transparente Behandlung von Sonderfällen ist damit ebenso möglich wie die Verarbeitung von errechneten Mengen, ohne das der Klient angepasst werden müsste.

2.2. Implementierung und Verwendung

Die Klassen TSimpleList und TSimpleIterator aus Abbildung 2-2 demonstrieren die Umsetzung eines Aggregats und die eines Iterators. Während die vorgestellte Liste die eigentliche Funktionalität an ein Exemplar der Standardklasse TObjectList delegiert, implementiert TSimpleIterator den Algorithmus zum Iterieren über die Liste.

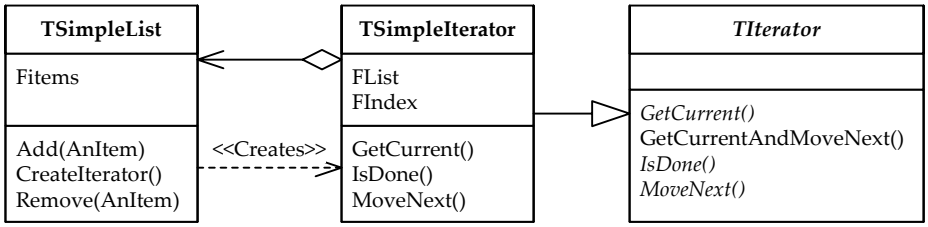

Abbildung 2-2: Einfache Umsetzung einer Liste mit einem Iterator

Ein lesender Zugriff auf den internen Zustand des *Aggregats* verdeutlicht die enge Kopplung beider Beteiligter während die Realisierung von GetCurrent() und MoveNext() wie in Quelltext 2-1 zeigt, dass der gewählte Ansatz gemäß [KOFLER1993] keinesfalls robust ist. Werden Elemente aus einer Liste entfernt oder hinzugefügt während ein Exemplar von TSimpleIterator mit GetCurrent() nach einem Element erfragt wird, können trotz unverändertem *Iterator* unterschiedliche Exemplare zurückgegeben werden[1].

[1] Die Tests der Klasse TIteratorTests im Anhang verdeutlichen diesen Effekt

```
function TSimpleIterator.GetCurrent: TObject;
begin
  Result := FList.FItems[FIndex];
end;

function TSimpleIterator.IsDone: Boolean;
begin
  Result := FIndex>=FList.FItems.Count;
end;

procedure TSimpleIterator.MoveNext;
begin
  Inc(FIndex);
end;
```

Quelltext 2-1: Implementierung von TSimpleIterator

Im Zusammenhang mit *Iteratoren* schreibt [GAMMA1996]: „Der Klient ist dafür zuständig, sie zu löschen. Dies führt schnell zu Fehlern, weil man leicht vergisst, einen auf dem Heap allozierten Iterator zu löschen, wenn er nicht mehr gebraucht wird“ und beschreibt damit die Speicherverwaltung der Sprache C++. In Delphi ist der Entwickler gleichfalls verantwortlich, den Speicher manuell freizugeben, der Einsatz eines Garbage Collectors wie in Java oder Smalltalk ist dabei nicht vorgesehen. Die Schwierigkeit, als Entwickler den Überblick der noch freizugebenden Objekte zu behalten, wird dadurch erschwert, dass eine Methode mehrere Austrittspunkt besitzen kann. Dies ist insbesondere bei Ausnahmebedingungen der Fall, die der Entwickler beim Einsatz von Objekten berücksichtigen muss.

```
// allocate resource
try
  // use resource
finally
  // free resource
end;
```

Quelltext 2-2: Empfohlener Umgang mit Ressourcen in Delphi

Zur fehlerfreien Verarbeitung von Ressourcen auch im Fall eines Fehlers schlägt [BORLAND2002#2] den Einsatz einer unbedingt abzuarbeitenden Fehlerbehandlungsroutine vor und führt das in Quelltext 2-2 dargestellte Idiom auf der Grundlage von Try…Finally ein.

Der konsequente Einsatz dieses Verfahrens zur Vermeidung von Speicherlecks lässt den Einsatz von *Iteratoren* in Delphi recht unattraktiv erscheinen. Das Beispiel aus Quelltext 2-3 verdeutlicht diesen Effekt und zeigt, dass bereits der Einsatz von zwei externen *Iteratoren* zum Vergleich der von ihnen zur Verfügung gestellten Elemente zu schwer einsehbarem Quelltext führt.

```
var
  iter1, iter2: TIterator;
begin
  iter1 := FList1.CreateIterator;
  try
    iter2 := FList2.CreateIterator;
    try
      while not (iter1.IsDone or iter2.IsDone) do
        CheckSame(iter1.GetCurrentAndMoveNext,
                  iter2.GetCurrentAndMoveNext);
    finally
      FreeAndNil(iter2);
    end;
  finally
    FreeAndNil(iter1);
  end;
end;
```

Quelltext 2-3: Explizite Speicherverwaltung erschwert den Einsatz von Iteratoren

Auf der Grundlage von [ELLIS1990] adressiert [GAMMA1996] dieses Problem und führt einen speziellen *Stellvertreter* des *Iterators* ein, der die Speicherverwaltung übernimmt: „Wir können ein auf dem Stack alloziertes Proxy als Stellvertreter für den eigentlichen Iterator verwenden. Das Proxy löscht den Iterator in seinem Destruktor.“ Das Verfahren macht sich dabei die Besonderheit zu Nutze, das spezielle C++-Objekte automatisch freigegeben werden, sobald sie ihren Gültigkeitsbereich verlassen. Dieser Einsatz von Hilfs-Objekten zur Vereinfachung des Speichermanagements wird von [ALEXANDRESCU2001] allgemein unter dem Begriff „Smart Pointer“ als geeignetes Verfahren zum sicheren Umgang mit Ressourcen in C++ diskutiert.

2.3. Implizite Referenzzählung

Objektschnittstellen in Delphi sind mit puren abstrakten Klassen in Kombination mit Mehrfachvererbung zu vergleichen. Eine Delphi-Klasse erbt dabei von einer Klasse und kann beliebig viele Interfaces implementieren, um zugleich typkompatibel zur Oberklasse und den Schnittstellen zu sein. Es liegt in der Verantwortung einer solchen Klasse, alle durch die von ihr unterstützten Schnittstellen eingeführten Methoden zur Verfügung zu stellen. Die Umsetzung dieses Prinzip zur Trennung zwischen Schnittstelle und Implementierung genügt auf der Ebene des Compilers der Spezifikation des Component Object Model [COMSpec], nach der jede Schnittstelle mit den Methoden _AddRef() und _Release() grundlegende Fähigkeiten zur Realisierung einer Referenzzählung aufweist.

Wann immer eine Referenz einer Variablen zugewiesen wird oder als Parameter einem anderen Beteiligten zur Verfügung gestellt wird, fügt der Delphi-Compiler implizit einen Aufruf von _AddRef() hinzu, um das Objekt über dessen Verwendung zu benachrichtigen. Analog ergänzt der Compiler automatisch Aufrufe von _Release(), wann immer der Gültigkeitsbereich eines über eine Schnittstellenreferenz verwendeten Objekts verlassen wird. Auf diese Weise hat ein Exemplar die Möglichkeit, einen Referenzzähler zu führen und allozierten Speicher freizugeben, wenn keine weitere Referenz auf das Objekt gehalten wird. Die Standardimplementierung TInterfacedObject realisiert dieses Verhalten und dient in Delphi als typische Basisklasse bei der Verwendung von Schnittstellen.

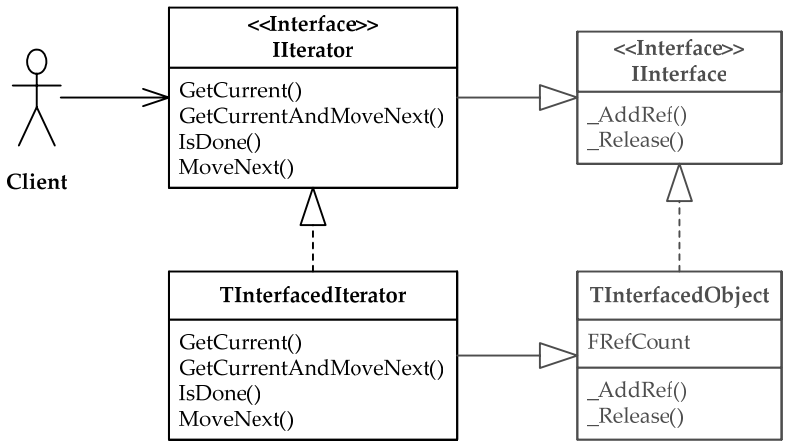

Abbildung 2-3: Implementierung eines Iterators gegen eine Schnittstelle

Der Einsatz von Schnittstellen in Delphi im Zusammenhang mit einem Iterator kann die im vorherigen Abschnitt beschriebenen Unzulänglichkeiten bei der expliziten Speicherverwaltung beheben und entspricht im Wesentlichen dem Ansatz von „Smart Pointern“ nach [ALEXANDRESCU2001] ohne jedoch eine zusätzliche Indirektion einzuführen. Stattdessen kann die Abstrakte Klasse TIterator wie in Abbildung 2-2 zu sehen einfach durch die Schnittstelle IIterator ausgetauscht werden. Wird entsprechend der Rückgabewert des Aggregats angepasst, kann das Beispiel aus dem vorherigen Abschnitt dramatisch vereinfacht werden. Quelltext 2-4 zeigt den Einsatz zweier Iteratoren mit impliziter Speicherverwaltung durch die automatische Referenzzählung von Schnittstellen.

```
var
  iter1, iter2: IIterator;
begin
  iter1 := FList1.CreateIterator;
  iter2 := FList2.CreateIterator;

  while not (iter1.IsDone or iter2.IsDone) do
    CheckSame(iter1.GetCurrentAndMoveNext,
              iter2.GetCurrentAndMoveNext);
end;
```

Quelltext 2-4: Implizite Speicherverwaltung beim Einsatz von Iteratoren

Das vorangegangene Beispiel zeigt den vereinfachten Umgang mit Iteratoren, die ohne eine explizite Speicherverwaltung auf der Grundlage von *Try…Finally* verwendet werden können. Insbesondere, wenn lediglich über die Elemente eines einzigen Aggregats iteriert werden sollen, bietet sich darüber hinaus der Einsatz des Schlüsselworts *with* wie in Quelltext 2-5 an.

```
with AList.CreateIterator do

  while not IsDone do
    CheckNotNull(GetCurrentAndMoveNext);
```

Quelltext 2-5: Schlüsselwort with vereinfacht die Verwendung eines Iterators

Quelltext 2-5 verdeutlicht den reduzierten Codierungsaufwand beim Einsatz eines Iterators nach dem vorgestellten Prinzip. Auch bei einem Aggregat mit komplexerer Struktur bleibt die Lösung dabei unverändert. Alle notwendigen Schritte und Hilfsobjekte zur Analyse des Aggregats werden vom *Iterator* verborgen und für den Klienten transparent verwaltet. Selbst im Vergleich zum trivialen Fall, dem Iterieren eines Index-basierten Aggregates, erfordert die vorgestellte Lösung keinen erhöhten Aufwand gegenüber einer einfachen Schleife in Delphi.

2.4. Konsequenzen

Die Wahl eines schnittstellenbasierten Entwurfs geschieht im beschriebenen Szenario weniger aus Gründen der Modularisierung und dem Streben nach einer wartbareren Struktur [FOWLER2001] als zur Einführung einer impliziten Speicherverwaltung. So wird die abstrakte Klasse TIterator durch eine Schnittstelle IIterator ersetzt und führt im Vergleich zu [GAMMA1996] keine zusätzlichen oder abweichenden Beteiligten ein.

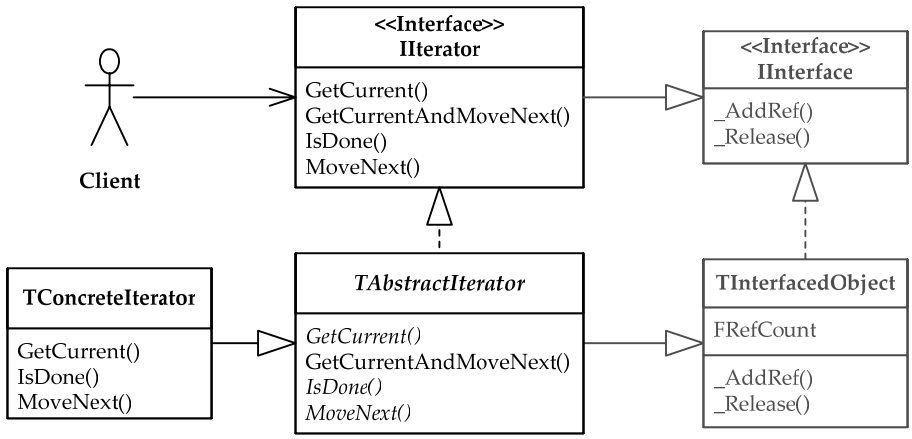

Abbildung 2-4: Einführung einer Schnittstelle zur impliziten Speicherverwaltung

Entsprechend dem Prinzip einer puren abstrakten Klasse kann eine Schnittstelle keine Operationen einführen, Methoden werden mit ihren Signaturen als Forderungen formuliert, die ausschließlich von Klassen realisiert werden können. Die vereinfachende Methode GetCurrentAndMoveNext() lässt sich so zwar auf die Methoden GetCurrent() und MoveNext() zurückführen, muss aber von jeder Klasse, die IIterator implementiert, erneut realisiert werden.

Zur Vermeidung dieser Redundanzen dient die Klasse TAbstractIterator aus Abbildung 2-4 als Oberklasse konkreter *Iteratoren*. Sie führt als abstrakte Klasse alle notwendigen Methoden des *Iterators* ein und implementiert auf der Grundlage dieser abstrakten Methoden alle zusätzlichen Methoden zur Steigerung des Komforts. Der Klient kann so den vollen Umfang der Schnittstelle nutzen, während bei der Implementierung von *Iteratoren* lediglich die notwendigen Aspekte umgesetzt werden.

3. Fabrik

Das Entwurfsmuster *abstrakte Fabrik* nach [GAMMA1996] bietet eine Operation zum Erzeugen eines Objekts, ohne seine konkrete Klasse zu benennen. In einer typstrengen Sprache wie Delphi führen diese Operationen den Typen eines *Fabrikats* ein, zu dem die Rückgabewerte einer *Fabrik* kompatibel sind.

3.1. Struktur und Verwendung

Eine *Fabrik* stellt ergänzend zu einer Klasse eine Stelle bereit, an die sich Klienten wenden können, um ein Exemplar erzeugen zu lassen. Anders als Klassen, beschreibt der Mechanismus jedoch nicht den konkreten Typen des *Fabrikats* als vielmehr einen Typen, zu dem die tatsächlich erzeugten Exemplare kompatibel sind. Ein Klient stützt sich nun nicht länger eine benannte Klasse sondern verwendet eine Ihm zur Verfügung gestellte *Fabrik*, die Objekte des geforderten Typs erzeugt.

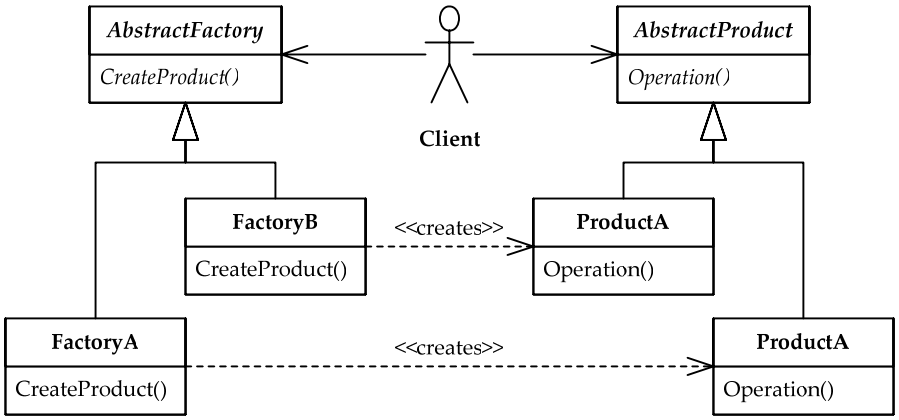

Abbildung 3-1: Struktur des Musters „Abstrakte Fabrik“ nach [GAMMA1996]

Der Klient aus Abbildung 3-1 verwendet eine konkrete Fabrik, die er lediglich über die Schnittstelle ihres Vorfahren AbstractFactory anspricht. Mit der abstrakten Methode CreateProduct() lässt er Exemplare des Typs AbstractProduct erzeugen, die von einer ihm unbekannten Klasse sind und von der konkreten *Fabrik* abhängen.

Durch den Einsatz von *Fabriken* kann die Wahl der vom Klienten eingesetzten Implementierungen zu Laufzeit durch einen anderen Teil des Systems bestimmt werden, so dass die Entscheidung der eingesetzten Logik nicht länger innerhalb des Klienten bestimmt werden muss. Während so gestaltete Lösungen ein hohes Maß an

Flexibilität aufweisen, begünstigen sie gleichfalls die Entwicklung gegen Schnittstellen unbekannter *Fabrikate* und damit letztlich einen dezentralen Entwicklungsprozess. Bis zur Fertigstellung tatsächlicher *Fabrikate* oder in Testszenarien können Mock-Objekte [LINK2005] verwendet werden, um die Funktionalität des Klienten fertig zu stellen. Für das produktive System kann ohne die Veränderung so erstellten Programmcodes die Implementierung des *Fabrikats* ausgetauscht werden, indem der Klient mit einer anderen *Fabrik* konfiguriert wird.

Fabriken werden nach [GAMMA1996] häufig als systemweit einmal vorhandene Exemplare beschrieben, über das Muster *Prototyp* oder konfigurierbare Fabriken im Allgemeinen lassen sich jedoch wesentlich flexiblere Systeme gestalten. Insbesondere in Kombination mit dem Muster *Dekorierer* können *Fabriken* so kombiniert werden, dass die erzeugten *Fabrikate* einer beliebigen *Fabrik* verändert oder gleichfalls dekoriert werden, bevor sie dem Klienten zur Verfügung gestellt werden. Neben der Erzeugung von Objekten können *Fabriken* gleichfalls eingesetzt werden, um mehrfachen Zugriff auf dasselbe Objekt zu realisieren. Sie können so denselben Effekt erzielen wie das Muster *Singleton*, ohne jedoch den Klienten an den zentralen Zugriffspunkt des *Singletons* zu binden, was analog zur Entkopplung einer benannten Klasse die Testbarkeit des Klienten erhöht.

3.2. Fabrikkompositionen

Die allgemeine Struktur der *abstrakten Fabrik* nach [GAMMA1996] betrachtet Objekte, die in der Lage sind, eine oder mehrere *Fabrikate* durch entsprechende Methoden zu erstellen, die zu einer *Produktfamilie* gehören. In seiner Konsequenz führt dieser Entwurf jedoch zu Problemen bei der Veränderung der Beschaffenheit dieser Produktfamilie. Wann immer neue Methoden der Basisklasse AbstractFactory hinzugefügt oder bestehende Signaturen verändert werden, um dem Klienten zusätzliche *Fabrikate* einer speziellen Beschaffenheit zur Verfügung zu stellen, sind alle Erben in der Verantwortung diese Veränderung abzubilden.

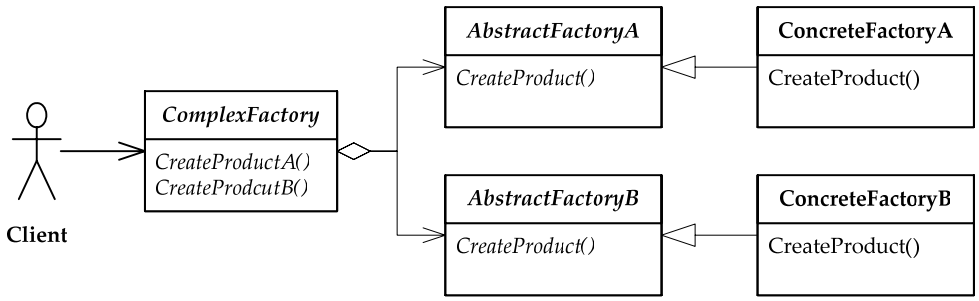

Abbildung 3-2: Komposition minimaler Fabriken

Die in Abbildung 3-2 vorgestellte Struktur geht stattdessen von *Fabriken* aus, die lediglich *Fabrikate* eines einzigen Typs erstellt. Die Anwendung einer Komposition so gearteter *Fabriken* fasst sie zusammen, um mit ComplexFactory eine *Produktfamilie* erzeugen zu lassen und entspricht damit dem Vorgehen „Replace Inheritance with Delegation“ nach [FOWLER1999]. Veränderung der Methoden von ComplexFactory führen im Falle eines neuen *Fabrikats* zum Bedarf einer neuen *Fabrik*, belassen die bestehenden *Fabriken* aber unverändert.

Neben der flexibleren Fortentwicklung eines auf *Fabriken* basierenden Systems nach diesem Ansatz erlaubt er darüber hinaus die feingranulare Konfiguration einer *Fabrik*, bei der ihre tatsächliche Beschaffenheit für jedes *Fabrikat* unabhängig festgelegt werden kann. Über *Metafabriken*, die solche Fabrikkompositionen erstellen, ist auch deren Erzeugung vor ihrer eigentlichen Verwendung transparent lösbar. Im Folgenden werden *Fabriken* betrachtet, die *Fabrikate* genau eines Typs zur Verfügung stellen.

3.3. Überschreibbare Klassenmethoden

Sowohl mit den Mustern *abstrakte Fabrik* als auch der *Fabrikmethode* beschreibt [GAMMA1996] Verfahren, bei denen über Polymorphie die tatsächliche Erzeugung von Exemplaren in Abhängigkeit der Vererbungslinie variiert. Die gebotenen Beispiele der Programmiersprache C++ aber auch Quellen wie [J2EEPatterns] für die Sprache Java oder [KLEINER2003] für Delphi und C# setzen zur Realisierung überschriebener Operationen dabei Objekte von Fabrikklassen voraus.

Delphi führt mit dem Konzept der Klassenreferenzen [BORLAND2002#1] die Möglichkeit zur Verwendung von Klassen als typisierte Parameter und Variablen. Ihrer Vererbungslinie folgend entspricht die Zuweisungskompatibilität solcher Referenzen der ihrer Exemplare. Wie auch die anderen der oben genannten Sprachen ist Delphi in der Lage, Methoden an eine Klasse zu binden, die gleichfalls über die Klassenreferenz eines Nachfahrens aufgerufen werden kann.

Diese Möglichkeiten werden in Delphi dadurch ergänzt, dass Klassenmethoden von Erben überschrieben werden können. Die Prinzipien der Polymorphie gelten nach diesem Ansatz, vergleichbar mit dem Prinzip der Metaklassen in Smalltalk, somit auch für Klassen und genügen allen Erwartungen an eine typstrenge Sprache.

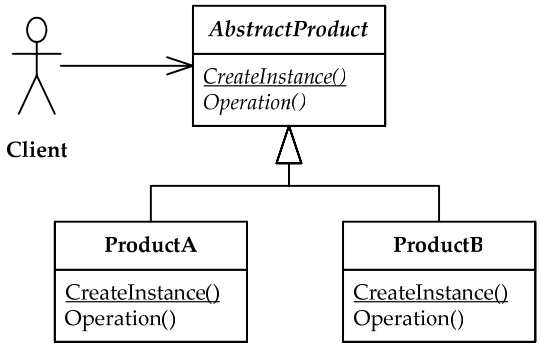

Abbildung 3-3: Erzeugung von Exemplaren mit Klassenmethoden

Die Aufgabe einer *abstrakten Fabrik*, wie sie im vorherigen Abschnitt beschrieben wurde, besteht nunmehr darin, ein Exemplar eines bestimmten Typs zu erzeugen. Das Wissen um die tatsächliche Klasse der erzeugten Fabrikate wird jedoch in die Implementierung einer konkreten *Fabrik* verlagert. Nach dem Entwurf aus Abbildung 3-3 führt das abstrakte *Fabrikat* AbstractProdukt die Operation CreateInstance() zum Erzeugen eines Exemplars von AbstractProdukt als

Klassenmethode des *Fabrikats* selbst ein. Die Klassen ProduktA und ProduktB überschreiben diese Methode entsprechend, um konkrete Exemplare ihrer Klassen zu erzeugen.

```
var
  productClass: class of TAbstractProduct;
  product: TAbstractProduct;
begin
  productClass := TProductA;
  product := AProductClass.CreateInstance;
  try
    CheckEquals(TProductA, product.ClassType);
    CheckEquals('A', product.Operation);
  finally
    FreeAndNil(product);
  end;
end;
```

Quelltext 3-1: Klassenmethode zur Erzeugung eines Exemplars

Das Beispiel aus Quelltext 3-1 zeigt den Einsatz einer Klassenreferenz productClass auf die Klasse TProductA zur Erzeugung eines *Fabrikats* vom Typ TAbstractProduct. Der Test prüft zunächst, ob das so erstellte Produkt von der Klasse TProductA stammt und erfragt anschließend den Rückgabewert von Operation() des *Fabrikats*.

Die Klassenmethode CreateInstance() übernimmt nach dem vorgestellten Ansatz die Aufgabe der Objekterzeugung, die sonst von einem Konstruktor realisiert wird. Im Gegensatz zum Aufruf von Konstruktoren benannter Klassen können mit der beschriebenen Struktur jedoch auch Exemplare von Klassen erzeugt werden, die dem Klienten nicht direkt bekannt sind. Die Polymorphie auf der Grundlage überschreibbarer Klassenmethoden erlaubt so eine spezialisierte Objekterzeugung in einer Unterklasse.

Technisch unterscheiden sich Klassenmethoden von Konstruktoren, was insbesondere damit zusammen hängt, das Konstruktoren sowohl zur Objekterzeugung als auch zur Initialisierung verwendet werden, Delphi erweitert das beschriebene Prinzip der Polymorphie jedoch ebenfalls auf Konstruktoren selbst. So kann die mit Abbildung 3-3 beschriebene Lösung ebenfalls unter Verwendung von optional überschreibbaren Konstruktoren realisiert werden, und Objekte von nur als Referenzen bekannte Klassen erzeugt werden.

```
type
  TAbstractProduct = class
  public
    constructor Create; virtual;
    function Operation: string; virtual; abstract;
  end;
```

Quelltext 3-2: Deklaration einer Klasse mit überschreibbarem Konstruktor

Die Verwendung eines überschreibbaren Konstruktors wie in Quelltext 3-2 erfüllt zum einen den Bedarf einer überschreibbaren Operation, die allein mit einer Klassenreferenz ausgeführt werden kann, und erhält zum anderen den Komfort bei der Entwicklung. Macht es die Beschaffenheit einer Klasse notwendig, einen speziellen Konstruktor einzuführen, steht er nach den Prinzipien als überschriebene Operation unmittelbar zu Verfügung, ohne dass zusätzlich eine ergänzende Klassenmethode überschrieben werden müsste. In Fällen, in denen keine speziellen Operationen zur Initialisierung ausgeführt werden müssen, kann der Konstruktor der Oberklasse wieder verwendet werden, eine spezielle Implementierung zur Erzeugung von Exemplaren der Klasse entfällt gänzlich.

3.4. Konsequenzen

Die Verwendung von Konstruktoren, die von Unterklassen überschrieben werden können, erlaubt es zusammen mit der Möglichkeit, Referenzen auf Klassen in Variablen abzulegen, Klassen wie Objekte einer Metaklasse zu verwenden. Auf diese Weise können die Klassen unterschiedlicher *Fabrikate* für ihre Konstruktion verwendet werden, ohne dass zusätzliche Fabrikklassen erstellt werden müssen oder der für Delphi typische Entwurf einer Klasse angepasst wird.

Die Tatsache, dass Klassen immer existent sind, stellt darüber hinaus eine Vereinfachung der Speicherverwaltung dar. Während bei dem klassischen Entwurf nach [GAMMA1996] die Zuständigkeit zur Freigabe des Speichers nach der Verwendung einer *Fabrik* geklärt werden müsste, sind Klassen als referenzierbare, globale Variable zu verstehen. Das Laufzeitsystem stellt die Gültigkeit von Klassenreferenzen sicher, während der Compiler die Typkompatibilität prüft.

Obgleich der vorgestellte Entwurf eine elegante Vereinfachung des allgemeinen Entwurfsmusters der *abstrakten Fabrik* bietet, stellt er sogleich eine Spezialisierung dar. *Fabriken* nach diesem Ansatz sind lediglich in der Lage, *Produkte* eines einzigen Typs zu erzeugen. Sollten *Produktfamilien* hinter der gemeinsamen Schnittstelle einer *Fabrik* notwendig sein, bedarf es der Hilfe einer Objektkomposition, wie in Abschnitt 3.2 beschrieben.

4. Befehl

Ein *Befehl* oder *Kommando* nach [GAMMA1996] kapselt eine Aktion als Objekt und ermöglicht es so, seine Klienten zu parametrisieren, Anfragen in Warteschlangen zu stellen oder ein Logbuch zu erstellen. Unterschiedlichste *Befehle* halten sich an die Schnittstelle des abstrakten *Befehls* und lassen spezialisierte Operationen zu, ohne dass bei Ihrer Verwendung Kenntnis über die Absicht eines *Kommandos* existieren muss.

4.1. Struktur und Verwendung

Das Muster *Befehl* abstrahiert von benannten Nachrichten an einen bekannten *Empfänger* und führt an deren Stelle eine uniforme, meist einfache Schnittstelle, die allen Aufrufen gemein ist und von einem Befehlsobjekt realisiert wird. Ein konkreter *Befehl* ruft seinerseits im Falle seiner Aktivierung die eigentliche Methode des *Empfängers* auf und ergänzt die dafür notwendigen Parameter um geeignete Werte, die einem *Befehl* als Exemplarvariablen zur Verfügung stehen. Vor dessen Verwendung durch einen *Aufrufer*, muss ein *Befehl* zunächst mit den erforderlichen Daten, wie den *Empfänger* und die eigentliche Nachricht sowie gegebenenfalls notwendigen Parametern durch den Klienten konfiguriert werden.

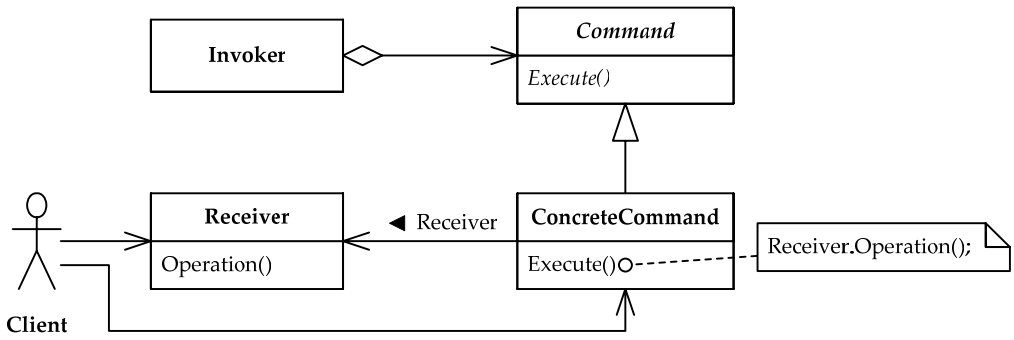

Abbildung 4-1: Struktur des Musters „Befehl“ nach [GAMMA1996]

Die Darstellung in Abbildung 4-1 zeigt die allgemeine Struktur des Befehlsmusters zusammen mit seinen Beteiligten. Während der Klient Kenntnis vom gewünschten *Empfänger* Receiver einer Nachricht Operation() hat, und ein Exemplar des konkreten *Befehls* ConcreteCommand konfiguriert, verwendet der *Aufrufer*

Invoker lediglich die Schnittstelle Command, die Oberklasse aller *Befehle*. Typischerweise wird ein vorbereitetes Exemplar des *Befehls* dem *Aufrufer* gleichfalls vom Klienten bekannt gemacht, dies ist jedoch auch indirekt möglich und daher nicht teil der abstrakten Struktur.

Das Sequenzdiagramm in Abbildung 4-2 verdeutlicht die entkoppelte Verwendung des *Empfängers* AReceiver. Der *Aufrufer* Invoker aktiviert die Methode Operation() nur indirekt über die ihm bekannte Nachricht Execute() an ACommand auf.

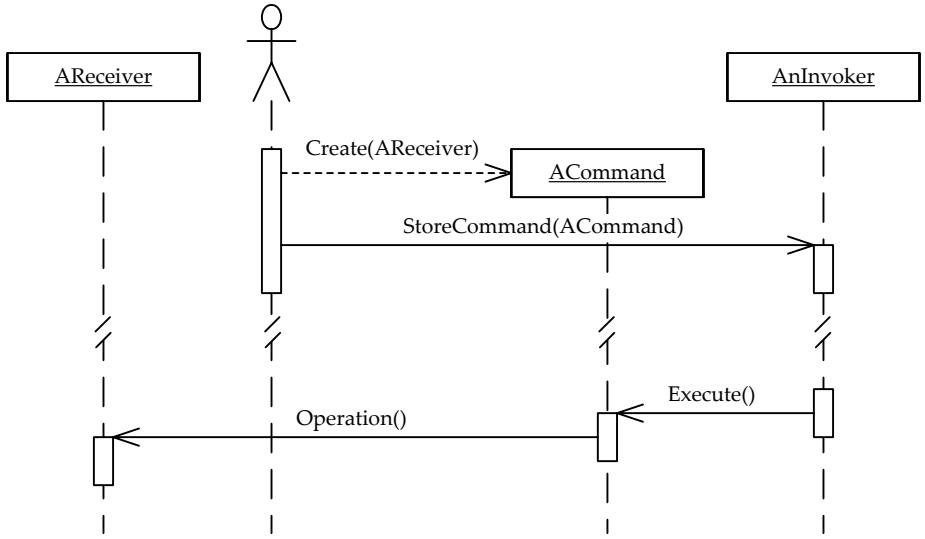

Abbildung 4-2: Verwendung eines Befehls als Sequenzdiagramm nach [GAMMA1996]

Die Funktionsweise eines *Befehls*, wie sie in Abbildung 4-2 dargestellt wird, entspricht damit im Wesentlichen dem Prinzip eines *Adapters* nach [GAMMA1996], der von einer erwarteten Schnittstelle zu einem Aufruf einer Methode des *Empfängers* einer dem *Aufrufer* unbekannten Schnittstelle übersetzt. *Befehle* werden zu diesem Zweck häufig eingesetzt, um ein komplexes Objekt an die Anforderungen eines Bedienelements einer graphischen Benutzeroberfläche anzubinden oder einen *Empfänger* für Callback-Routinen zur Verfügung zu stellen.

4.2. Implementierungen anderer Programmiersprachen

Während Sprachen wie Lisp oder Smalltalk so genannte *Closures* unterstützen, also Objekte, die Ausführbaren Code repräsentieren und über spezielle Konstruktoren direkt im Quelltext erzeugt werden können, sieht Java die Erzeugung von inneren Klassen vor, um Befehlsobjekte zu erzeugen, die einen vollständigen Zugriff auf die umgebende Klasse besitzen.

```
private void operation(){
  // Do sth. useful
}

  private class ConcreteCommand extends Command{
    public void execute(){
      operation();
  }

protected void wireCommand() {
  anInvoker.addCommand(new ConcreteCommand());
}
```

Quelltext 4-1: Private innere Klasse in Java zur Realisierung eines konkreten Befehls

Das Beispiel aus Quelltext 4-1 zeigt den Einsatz der inneren Java-Klasse ConcreteCommand, auf die ausschließlich von der umgebenden Klasse zugegriffen werden kann. In der Methode wireCommand() wird ein Exemplar dieses *Befehls* mit dem Aufrufer anInvoker verknüpft. Die Implementierung execute() übersetzt schließlich auf die Methode operation() der umgebenden Klasse. Das ist insbesondere deshalb möglich, weil innere Klassen in Java implizit eine Referenz auf das Exemplar der umgebenden Klasse innehaben, von dem sie erzeugt wurden.

Während es auf syntaktischer Ebene durch den Einsatz innerer Klassen zusammen mit der Möglichkeit auf auf private Felder der umgebenden Klasse zuzugreifen und Java darüber hinaus ebenfalls den Einsatz von anonymen Klassen zulässt, deren Implementierung direkt bei der Verwendung angegeben wird, ändert der Einsatz solcher Hilfsobjekte nicht die grundlegende Struktur des Entwurfs. Wie bei Funktionsblöcken in Smalltalk wird zunächst ein spezielles Objekt erzeugt, das bei seiner Aktivierung in eine Nachricht des eigentlichen Empfängers übersetzt.

4.3. Methodenzeiger in Delphi

Als konsequente Weiterführung von Prozedurzeigern des Sprachumfangs von Pascal [WIRTH1971] unterstützt Delphi die Idee von *Methodenzeigern*. Sie ergänzen das Konzept zur Referenzierung einer durch eine Klasse beschriebene Methode um die Fähigkeit, ein Exemplar als Empfänger des Methodenaufrufs zu halten. Wie die meisten Variablen in Delphi sind auch Methodenzeiger von einem bestimmten Typen. Er beschreibt die Signatur der referenzierbaren Methoden. Quelltext 4-2 zeigt einen prozeduralen Typen TCommandMethod an deren Variablen die Methoden InternalMethod(…) oder Operation(…) gebunden werden können, um ein Objekt der Klasse TArbitraryClass zu referenzieren.

```
type
  TCommandMethod = procedure(AParam: Integer) of object;

  TArbitraryClass = class
  private
    procedure InternalMethod(Value: Integer);
  public
    procedure Operation(AParameter: Integer); virtual;
  end;
```

Quelltext 4-2: Deklaration eines Methodenzeigertyps sowie referenzierbarer Methoden

Variablen eines prozeduralen Typs in Delphi können hinsichtlich ihrer Syntax wie ordinäre Prozeduren unabhängig von ihrem Kontext verwendet werden, um sie in eine signaturidentische Nachricht an das referenzierte Objekt übersetzen zu lassen. Die Prinzipien der Polymorphie werden dabei ebenso eingehalten wie zur Übersetzungszeit die Typkompatibilität bei der Verwendung durch den Klienten und die Zuweisung einer Methode an die Variable überprüft wird.

```
procedure TTestCommand.Expects42(Param: Integer);
begin
  CheckEquals(42, Param);
end;

procedure TTestCommand.TestSimpleCall;
begin
  FInvoker.AddCommandMethod(Self.Expects42);
  FInvoker.InvokeCommandMethods(42);
end;
```

Quelltext 4-3: Verwendung von Methodenzeigern zur Vereinfachung des Musters Befehl

Das Beispiel aus Quelltext 4-3 zeigt, wie die Methode AddCommandMethod(…) als Parameter die Methode Expects42(…) verknüpft mit dem Exemplar Self entgegennimmt. Die Implementierung von InvokeCommandMethods(…) der Klasse TInvoker iteriert über alle registrierten Methodenzeiger und ruft sie mit dem übergebenen Parameter auf. Ein so implementierter *Aufrufer* ist nicht länger an eine spezielle Schnittstelle von verwalteten Objekten gebunden, sondern ist in der Lage, Methoden mit geeigneter Signatur von beliebigen Objekten entgegen zu nehmen.

4.4. Konsequenzen

Mit dem Einsatz von Methodenzeigern ist eine weitestgehende Entkopplung zwischen dem *Aufrufer* und dem *Empfänger* von *Befehlen* möglich. Der Einsatz von minimalen Hilfsobjekten zur Adaption zwischen der erwarteten Schnittstelle und der letztlich aufzurufenden Methode entfällt und vereinfacht die Struktur des klassischen Entwurfsmuster nach [GAMMA1996] entscheidend.

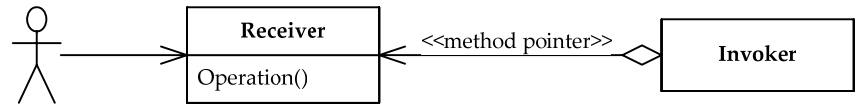

Abbildung 4-3: Methodenzeiger vereinfachen die Struktur des Entwurfsmusters Befehl

Im Vergleich zur klassischen Struktur, wie in Abschnitt 4.1 vorgestellt, zeigt Abbildung 4-3 die Beteiligten eines Entwurfs des Musters *Befehl* auf der Grundlage von Methodenzeigern. Eine abstrakte Oberklasse Command ist ebenso wenig notwendig, wie ein spezieller Nachfahre ConcreteCommand, um den *Aufrufer* Invoker mit dem tatsächlichen *Empfänger* Receiver zu verbinden. Die Lösung

stützt sich auf die Möglichkeit, Methoden typstreng verfügbar zu machen und als Tupel zusammen mit einem Empfängerobjekt in Variablen abzulegen.

Durch die Möglichkeit, Methoden beliebiger Signatur auf diese Weise bekannt zu machen, eignet sich die Vorgeschlagene Struktur darüber hinaus auch für Muster wie dem *Beobachter* oder dem *Mediator*. Insbesondere das Muster *Strategie* kann auf diese Weise ohne die Kopplung zwischen den Rollen *Kontext* und *Strategie* auskommen.

In einigen Fällen übernimmt ein *Kommando* neben der Aufgabe, zwischen Schnittstellen zu übersetzen, zusätzliche Aufgaben. So Schlägt [GAMMA1996] eine erweiterte Schnittstelle der Oberklasse Command vor, um einen entgegengesetzten *Befehl* erzeugen zu lassen, um so Funktionalitäten zum Zurücknehmen und erneutem Ausführen eines *Befehls* anbieten zu können. Gleichwohl sich diese Anforderung durch einen weiteren Methodenzeiger lösen ließe, verliert der vorgeschlagene Ansatz mit steigender Komplexität der Schnittstelle des erwarteten *Befehls* an Eleganz. Wenn die erwartete Schnittstelle jedoch ausschließlich die Signatur einer einzigen Methode beschreibt, ist der vorgestellte Entwurf an Einfachheit schwer zu überbieten.

Kommandos kapseln nach [GAMMA1996] einen konkreten Methodenaufruf als Objekt. Insbesondere alle Parameter einer signaturfremden Methode müssen daher bei der Aktivierung des *Befehls* auf der Grundlage seiner Exemplarvariablen oder anderen Kriterien rekonstruiert werden, bevor die eigentliche Nachricht an den *Empfänger* gesendet werden kann. Sollen beispielsweise *Kommandos* zum selektiven Entfernen von Elementen einer Menge angeboten werden, lassen sich nur schwerlich statische Methoden formulieren, die mit Methodenzeigern ohne zusätzliche Exemplarvariablen das jeweils zu löschende Element der Menge als *Empfänger* identifizieren. Müssen entsprechend zur Formulierung der tatsächlichen Nachricht zusätzliche Daten abgelegt werden, oder soll ein *Empfänger* zusammen mit vielen *Aufrufern* verwendet werden, die zueinander heterogene Methodensignaturen erwarten, sollten weiterhin Hilfsobjekte eingesetzt werden, um den Empfänger nicht mit zweckfremden Operationen zu belasten. In Fällen, bei denen der *Empfänger*

jedoch wenige, klar abgrenzbare Operationen ausführt, können nicht nur diese Hilfsobjekte sondern auch die abstrakte Schnittstelle eingespart werden.

Das Prinzip dieser Methodenzeiger ist mit den so genannten *Delegates* in der Programmiersprache C# ein syntaktischer Bestandteil, der bei der Verknüpfung zwischen Objekten eine wesentliche Rolle Spielt. Anders Hejlsberg, führender Architekt der Sprache C#, beschreibt die Vereinfachung durch die Vermeidung von Hilfsobjekten zur Übersetzung zwischen Schnittstellen in [HEJLSBERG2003] wie folgt: „Delegates add a kind of expressiveness that you don't get with classes or interfaces, which is why I think they are important. […] You can indeed do with interfaces what you do with delegates, but you are often forced to do a lot of housekeeping. […] With a delegate, by contrast, as long as the signatures are compatible, you can just slot them together. The guy handling the [invocation] doesn't care how he got called. It's just a method."

Der vorgestellte Entwurf auf der Grundlage von Methodenzeigern und der Einsatz einer Schnittstelle schließen sich dabei nicht aus. So sind minimale Hilfsobjekte mit genau einer Methode denkbar, die der Signatur einer Methode entsprechen. Sie können in Fällen eingesetzt werden, bei denen ein *Befehl* zunächst konfiguriert werden muss. Auf der anderen Seite kann eine spezielle Kommandoklasse als Erbe einer erwarteten Oberklasse Command seinerseits einen Methodenzeiger verwenden, um in der Mehrzahl der Fälle die Bildung zusätzlicher Unterklassen einzusparen.

5. Zusammenfassung

Entwurfsmuster bilden die Möglichkeit, Entwicklungszeiten zu verkürzen und die Qualität von Softwarelösungen zu steigen, indem sie etablierte Verfahren als wiederverwendbare Vorlagen verfügbar machen. Sie führen eine übergreifende Sprache zwischen den unterschiedlichen Beteiligten des Erstellungsprozesses von Software ein und erlauben eine gemeinsame Nutzung von Wissen über die Grenzen einer spezifischen Problemlösung oder Zielplattform hinaus.

Mit dem Werk „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“ [GAMMA1996] ist ein Katalog von 23 Entwurfsmustern dargelegt, deren Verständnis in ihrer puren Form für jeden modernen, objektorientierten Softwarenentwickler essenziell ist. Jedes der vorgestellten Muster kann, als Programmquelltext umgesetzt, jedoch unterschiedlichste Ausprägungen aufweisen. Das Wissen um die dargestellten Muster bedarf daher zusätzlicher Kenntnis, um die abstrakte Form den gegebenen Rahmenbedingungen anzupassen.

Die vorliegende Arbeit zeigt einen kleinen Ausschnitt von spezialisierten Implementierungen in Delphi zur Umsetzung ausgewählter Entwurfsmuster. Die Möglichkeiten der Programmiersprache wurden eingesetzt, um offenkundige Probleme zu lösen oder plattformspezifische Vorteile auszunutzen. Für den praktischen Einsatz von Entwurfsmustern stellen Ansätze dieser Art eine große Bereicherung dar, können sie doch sowohl Struktur als auch die Verwendung von Entwurfsmustern entscheidend vereinfachen.

Es wird deutlich, dass Vorlagen zur Unterstützung der Softwaretechnik nicht immer unverändert übernommen werden können und selten sollten. Dies gilt nicht nur für die anwachsende Menge an öffentlich diskutierten Entwurfsmustern, die über den Katalog [GAMMA1996] hinausgehen, sondern insbesondere wohl auch für die vorgestellten Verfahren.

Anhang A: Inhalt des beigefügten Datenträgers

Der beigefügte Datenträger enthält die Ergebnisse dieser Arbeit als Quelltext im Ordner /Src. Darüber hinaus zeigen die Quelltexte aus /Tests den Gebrauch der Entwurfsmuster. Der Ordner /Bin enthält ein kompiliertes Testprogramm zur Ausführung unter Microsoft Windows.

/Bin/DUnitTests.exe	Ausführbares Testprogramm
/Doc/Assignment.pdf	Das vorliegende Dokument als PDF
/Tests/DUnitIteratorTests.pas	Verwendung des Iterators
/Tests/DUnitFactoryTests.pas	Verwendung der Fabrik
/Tests/DUnitCommandTests.pas	Verwendung des Befehls
/Src/SimpleIterator.pas	Klassenbasierter Iterator
/Src/InterfacedIterator.pas	Iterator mit Referenzzählung
/Src/ClassBasedFactory.pas	Fabrik mit virtuellem Konstruktor
/Src/MethodPointerBasedCommand.pas	Aufrufer mit Methodenzeiger

Anhang B: Quelltexte

Dieser Anhang und die in ihm aufgenommenen Quelltexte umfassen zum einen die Implementierungen der vorgestellte Muster, zum anderen zeigen Sie in Form von speziellen Testklassen eine Mögliche Form ihrer Verwendung. Das nachfolgende Verzeichnis dient der schnelleren Auffindung der unterschiedlichen Programmtexte.

Umsetzung der vorgestellten Muster

Die nachfolgenden Quelltexte zeigen minimale Implementierungen der vorangegangenen Kapitel. Mit den Units SimpleIterator und InterfacedIterator werden die Implementierung des Kapitels Iterator gezeigt, während ClassBasedFactory den Einsatz der virtuellen Konstruktoren aus dem Kapitel Fabrik verdeutlicht. Der Quelltext der Unit MethodPointerBasedCommand zeigt die Implementierung eines *Aufrufers* auf der Grundlage eines Methodenzeigers aus dem Kapitel Befehl.

Unit SimpleIterator

```
interface
uses
  Contnrs;

type
  TIterator = class // abstract
    function GetCurrent: TObject; virtual; abstract;
    function GetCurrentAndMoveNext: TObject;
    function IsDone: Boolean; virtual; abstract;
    procedure MoveNext; virtual; abstract;
  end;

  TAggregate = class // pure abstract
  public
    function CreateIterator: TIterator; virtual; abstract;
  end;

  TSimpleList = class(TAggregate)
  private
    FItems: TObjectList;
  public
    constructor Create;
    destructor Destroy; override;

    procedure Add(const AnItem: TObject);
    procedure Remove(const AnItem: TObject);
    function CreateIterator: TIterator; override;
  end;

  TSimpleIterator = class(TIterator)
  private
    FList: TSimpleList;
    FIndex: Integer;
  public
    constructor Create(const AList: TSimpleList);

    function GetCurrent: TObject; override;
    function IsDone: Boolean; override;
    procedure MoveNext; override;
  end;

implementation
```

```
uses
  SysUtils;

// TSimpleList

procedure TSimpleList.Add(const AnItem: TObject);
begin
  FItems.Add(AnItem);
end;

constructor TSimpleList.Create;
begin
  inherited Create;
  FItems := TObjectList.Create(False);
end;

function TSimpleList.CreateIterator: TIterator;
begin
  Result := TSimpleIterator.Create(Self);
end;

destructor TSimpleList.Destroy;
begin
  FreeAndNil(FItems);
  inherited;
end;

procedure TSimpleList.Remove(const AnItem: TObject);
begin
  FItems.Remove(AnItem);
end;

// TSimpleIterator
constructor TSimpleIterator.Create(const AList: TSimpleList);
begin
  inherited Create;
  FList := AList;
end;

function TSimpleIterator.GetCurrent: TObject;
begin
  Result := FList.FItems[FIndex];
end;

function TSimpleIterator.IsDone: Boolean;
begin
  Result := FIndex>=FList.FItems.Count;
end;

procedure TSimpleIterator.MoveNext;
begin
  Inc(FIndex);
end;

// TIterator
function TIterator.GetCurrentAndMoveNext: TObject;
begin
  Result := GetCurrent;
  MoveNext;
end;
end.
```

Unit InterfacedIterator

```
interface
uses
  Contnrs;

type
  IIterator = interface
  ['{6F611660-12DA-4C59-8951-C81FEB88E80F}']
    function GetCurrent: TObject;
    function GetCurrentAndMoveNext: TObject;
    function IsDone: Boolean;
    procedure MoveNext;
  end;

  IAggregate = interface
  ['{46D2376C-8AA2-4B79-AEFD-E9B9992A633B}']
    function CreateIterator: IIterator;
  end;

  IList = interface(IAggregate)
  ['{77F2C46F-775C-4870-AAA6-6102D8F5E852}']
    procedure Add(const AnItem: TObject);
    procedure Remove(const AnItem: TObject);
  end;

  TInterfacedList = class(TInterfacedObject, IList, IAggregate)
  private
    FItems: TObjectList;
  public
    constructor Create;
    destructor Destroy; override;

    procedure Add(const AnItem: TObject);
    procedure Remove(const AnItem: TObject);
    function CreateIterator: IIterator;
  end;

  TInterfacedIterator = class(TInterfacedObject, IIterator)
  private
    FList: TInterfacedList;
    FIndex: Integer;
  public
    constructor Create(const AList: TInterfacedList);
    destructor Destroy; override;

    function GetCurrent: TObject;
    function IsDone: Boolean;
    procedure MoveNext;
    function GetCurrentAndMoveNext: TObject;

  end;

implementation
uses
  SysUtils;
```

```
// TSimpleList
procedure TInterfacedList.Add(const AnItem: TObject);
begin
  FItems.Add(AnItem);
end;

constructor TInterfacedList.Create;
begin
  inherited Create;
  FItems := TObjectList.Create(False);
end;

function TInterfacedList.CreateIterator: IIterator;
begin
  Result := TInterfacedIterator.Create(Self);
end;

destructor TInterfacedList.Destroy;
begin
  FreeAndNil(FItems);
  inherited;
end;

procedure TInterfacedList.Remove(const AnItem: TObject);
begin
  FItems.Remove(AnItem);
end;

// TSimpleIterator
constructor TInterfacedIterator.Create(const AList: TInterfacedList);
begin
  inherited Create;
  FList := AList;
  FList._AddRef;
end;

destructor TInterfacedIterator.Destroy;
begin
  FList._Release;
  inherited;
end;

function TInterfacedIterator.GetCurrent: TObject;
begin
  Result := FList.FItems[FIndex];
end;

function TInterfacedIterator.GetCurrentAndMoveNext: TObject;
begin
  Result := GetCurrent;
  MoveNext;
end;

function TInterfacedIterator.IsDone: Boolean;
begin
  Result := FIndex>=FList.FItems.Count;
end;

procedure TInterfacedIterator.MoveNext;
begin
  Inc(FIndex);
end;
end.
```

Unit ClassBasedFactory

```
interface

type
  TProductClass = class of TAbstractProduct;

  TAbstractProduct = class
  protected
    // reduce visibility of standard constructor
    constructor Create; reintroduce;
  public
    class function CreateInstance: TAbstractProduct; virtual; abstract;
    function Operation: string; virtual; abstract;
  end;

  TProductA = class(TAbstractProduct)
    class function CreateInstance: TAbstractProduct; override;
    function Operation: string; override;
  end;

  TProductB = class(TAbstractProduct)
    class function CreateInstance: TAbstractProduct; override;
    function Operation: string; override;
  end;

implementation

// TProductA
class function TProductA.CreateInstance: TAbstractProduct;
begin
  // Self = TProductA
  Result := Self.Create;
end;

function TProductA.Operation: string;
begin
  Result := 'A';
end;

// TProductB
class function TProductB.CreateInstance: TAbstractProduct;
begin
  // Self = TProductB
  Result := Self.Create;
end;

function TProductB.Operation: string;
begin
  Result := 'B';
end;

// TAbstractProduct

constructor TAbstractProduct.Create;
begin
  inherited
end;

end.
```

Unit MethodPointerBasedCommand

```
interface

type
  TCommandMethod = procedure(AParam: Integer) of object;

  TInvoker = class
  private
    FMethods: array of TCommandMethod;
    function GetIndexOf(const AMethod: TCommandMethod): Integer;
  public
    procedure AddCommandMethod(const AMethod: TCommandMethod);
    procedure RemoveCommandMethod(const AMethod: TCommandMethod);

    procedure InvokeCommandMethods(AParam: Integer);
  end;

implementation

// TInvoker

procedure TInvoker.AddCommandMethod(const AMethod: TCommandMethod);
begin
  if Assigned(AMethod) and (GetIndexOf(AMethod)<0) then
  begin
    SetLength(FMethods, Succ(Length(FMethods)));
    FMethods[High(FMethods)] := AMethod;
  end;
end;

function TInvoker.GetIndexOf(const AMethod: TCommandMethod): Integer;
begin
  Result := High(FMethods);
  while (Result>=Low(FMethods)) and (@FMethods<>@AMethod) do
    Dec(Result);
end;

procedure TInvoker.InvokeCommandMethods(AParam: Integer);
var
  i: Integer;
begin
  for i := Low(FMethods) to High(FMethods) do
    FMethods[i](AParam);
end;

procedure TInvoker.RemoveCommandMethod(const AMethod: TCommandMethod);
var
  i: Integer;
begin
  i := GetIndexOf(AMethod);
  if i>=0 then
  begin
    while i<High(FMethods) do
      FMethods[i] := FMethods[i+1];
    SetLength(FMethods, Pred(Length(FMethods)));
  end;
end;

end.
```

Verwendung der vorgestellten Muster

Die folgenden Quelltexte verdeutlichen den Einsatz der zuvor beschriebenen Implementierungen der Entwurfsmuster. Bei den gezeigten Klassen handelt es sich um DUnit-Tests [DUnit] in der Rolle des Klienten der jeweiligen Muster.

Unit DUnitIteratorTests

```
interface

uses
  TestFrameWork,
  SimpleIterator,
  InterfacedIterator;

type
  TTestSimpleIterator = class(TTestCase)
  private
    FObject1, FObject2, FObject3: TObject;
    FList1, FList2: TSimpleList;
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  public
  published
    procedure TestRobustness;
    procedure TestCompareLists;
  end;

  TTestInterfacedIterator = class(TTestCase)
  private
    FObject1, FObject2, FObject3: TObject;
    FList1, FList2: IList;
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  public
  published
    procedure TestCompareLists;
    procedure TestSimplicity;
  end;

implementation
uses
  SysUtils;
```

```
40  // TTestSimpleIterator
    procedure TTestSimpleIterator.SetUp;
    begin
      inherited;
      FObject1 := TObject.Create;
45    FObject2 := TObject.Create;
      FObject3 := TObject.Create;
      FList1 := TSimpleList.Create;
      FList2 := TSimpleList.Create;

50    FList1.Add(FObject1);
      FList1.Add(FObject2);
      FList2.Add(FObject1);
      FList2.Add(FObject2);
      FList2.Add(FObject3);
55  end;

    procedure TTestSimpleIterator.TearDown;
    begin
      FreeAndNil(FObject1);
60    FreeAndNil(FObject2);
      FreeAndNil(FObject3);
      FreeAndNil(FList1);
      FreeAndNil(FList2);
      inherited;
65  end;

    procedure TTestSimpleIterator.TestCompareLists;
    var
      iter1, iter2: TIterator;
70  begin
      iter1 := FList1.CreateIterator;
      try
        iter2 := FList2.CreateIterator;
        try
75        while not (iter1.IsDone or iter2.IsDone) do
            CheckSame(iter1.GetCurrentAndMoveNext,
                      iter2.GetCurrentAndMoveNext);
          CheckNotEquals(iter1.IsDone, iter2.IsDone);
        finally
80        FreeAndNil(iter2);
        end;
      finally
        FreeAndNil(iter1);
      end;
85  end;

    procedure TTestSimpleIterator.TestRobustness;
    var
      iterator: TIterator;
90    obj: TObject;
    begin
      iterator := FList1.CreateIterator;
      try
        obj := iterator.GetCurrent;
95      FList1.Remove(FObject1);
        CheckFalse(obj = iterator.GetCurrent);
      finally
        iterator.Free;
      end;
100 end;
```

```
// TTestInterfacedIterator
procedure TTestInterfacedIterator.SetUp;
begin
  inherited;
  FObject1 := TObject.Create;
  FObject2 := TObject.Create;
  FObject3 := TObject.Create;

  FList1 := TInterfacedList.Create;
  FList2 := TInterfacedList.Create;

  FList1.Add(FObject1);
  FList1.Add(FObject2);
  FList2.Add(FObject1);
  FList2.Add(FObject2);
  FList2.Add(FObject3);
end;

procedure TTestInterfacedIterator.TearDown;
begin
  FreeAndNil(FObject1);
  FreeAndNil(FObject2);
  FreeAndNil(FObject2);

  // lists are covered by reference counting

  inherited;
end;

procedure TTestInterfacedIterator.TestCompareLists;
var
  iter1, iter2: IIterator;
begin
  iter1 := FList1.CreateIterator;
  iter2 := FList2.CreateIterator;

  while not (iter1.IsDone or iter2.IsDone) do
    CheckSame(iter1.GetCurrentAndMoveNext,
              iter2.GetCurrentAndMoveNext);

  CheckNotEquals(iter1.IsDone, iter2.IsDone);
end;

procedure TTestInterfacedIterator.TestSimplicity;
begin
  with FList1.CreateIterator do
    while not IsDone do
      CheckNotNull(GetCurrentAndMoveNext);
end;

initialization
  TestFramework.RegisterTest(TTestSimpleIterator.Suite);
  TestFramework.RegisterTest(TTestInterfacedIterator.Suite);

end.
```

Unit DUnitFactoryTests

```
interface

uses
  TestFrameWork,
  ClassBasedFactory;

type
  TTestFactory = class(TTestCase)
  private
    FFactoryA, FFactoryB: TProductClass;
  protected
    procedure SetUp; override;
  published
    procedure TestInstantiation;
  end;


implementation
uses
  SysUtils;

// TTestFactory

procedure TTestFactory.SetUp;
begin
  inherited;
  FFactoryA := TProductA;
  FFactoryB := TProductB;
end;

procedure TTestFactory.TestInstantiation;
var
  product: TAbstractProduct;
begin
  product := FFactoryA.CreateInstance;
  try
    CheckEquals(TProductA, product.ClassType);
    CheckEquals('A', product.Operation);
  finally
    FreeAndNil(product);
  end;
end;

initialization
  TestFramework.RegisterTest(TTestFactory.Suite);

end.
```

Unit DUnitCommandTests

```
interface

uses
  TestFrameWork,
  MethodPointerBasedCommand;

type
  TTestCommand = class(TTestCase)
  private
    FInvoker: TInvoker;
    procedure Expects42(Param: Integer);
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestSimpleCall;
  end;


implementation
uses
  SysUtils;

// TTestFactory


procedure TTestCommand.SetUp;
begin
  inherited;
  FInvoker := TInvoker.Create;
end;

procedure TTestCommand.TearDown;
begin
  FreeAndNil(FInvoker);
  inherited;
end;

procedure TTestCommand.Expects42(Param: Integer);
begin
  CheckEquals(42, Param);
end;

procedure TTestCommand.TestSimpleCall;
begin
  FInvoker.AddCommandMethod(Self.Expects42);
  FInvoker.InvokeCommandMethods(42);
end;

initialization
  TestFramework.RegisterTest(TTestCommand.Suite);

end.
```

Literaturverzeichnis

[ALEXANDER1997] Alexander, C., Ishikawa, S., et al. (1977). *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press.

[ALEXANDRESCU2001] Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley.

[BOOCH1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings

[BORLAND2002#1] Borland (2002). *Delphi Sprachreferenz*. Langen, Borland GmbH.

[BORLAND2002#2] Borland (2002). *Delphi Entwicklerhandbuch*. Langen, Borland GmbH.

[BUSCHMANN1996] Buschmann, F., Meunier, R., et al. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons.

[COMSpec] Microsoft (1995) *The Component Object Model Specification* unter http://www.microsoft.com/com/resources/comdocs.asp

[Delphi] *Delphi* unter http://www.borland.com/delphi/

[DUnit] *DUnit: An Xtreme testing framework for Borland Delphi programs* unter http://dunit.sourceforge.net/

[ELLIS1990] Ellis, M. A. und Stroustrup, B. (1990). *The Annotated C++ Reference Manual*, Addison-Wesley.

[FOWLER1999] Fowler, M., Beck, K., et al. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

[FOWLER2001] Fowler, M. (2001) *Reducing Coupling* unter http://www.martinfowler.com/ieeeSoftware/coupling.pdf

[FOWLER2002] Fowler, M., Rice, D., et al. (2002). *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional.

[GAMMA1996] Gamma, E., Helm, R., et al. (1996). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software* Addison-Wesley.

[HEJLSBERG2003] *Delegates, Components, and Simplexity* unter http://www.artima.com/intv/simplexity.html

[J2EEPatterns] *Core J2EE Patterns* unter http://java.sun.com/blueprints/corej2eepatterns/Patterns/

[KLEINER2003] Kleiner, M., Rothen, S., et al. (2003). *Patterns konkret*, Software & Support.

[KOFLER1993] Kofler, T. (1993). *Robust Iterators in ET++* Structured Programming 14(2): 62-85.

[LINK2005] Link, J. (2005). *Softwaretests mit JUnit*, Dpunkt Verlag.

[RIEL1996] Riel, A. J. (1996). *Object-Oriented Design Heuristics*, Addison-Wesley.

[SCHMIDT2000] Schmidt, D., Stal, M., et al. (2000). *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*, John Wiley & Sons.

[WIRTH1971] Wirth, N. (1971). *The Programming Language Pascal.* Acta Informatica. 1.