PROGRAMMER'S CHOICE

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Entwurfsmuster

Elemente wiederverwendbarer objektorientierter Software

ADDISON-WESLEY

3 Erzeugungsmuster

Entwurfsmuster, die der Erzeugung von Objekten dienen, verstecken den Erzeugungsprozeß. Sie helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Ein klassenbasiertes Erzeugungsmuster verwendet Vererbung, um die Klasse des zu erzeugenden Objekts zu variieren, während ein objektbasiertes Erzeugungsmuster die Erzeugung an ein anderes Objekt delegiert.

Erzeugungsmuster sind vor allem dann von Bedeutung, wenn Systeme beginnen, mehr von Objektkomposition als von Vererbung abzuhängen. Dabei bewegt sich die Konzentration von der Programmierung festgelegten Verhaltens weg. Sie bewegt sich hin zur Definition einer kleineren Menge grundlegender Verhaltenseinheiten, die zu beliebig komplexem Verhalten zusammengesetzt werden können. Deswegen verlangt das Erzeugen von Objekten mit bestimmtem Verhalten mehr als nur das Erzeugen eines Objekts einer einzelnen Klasse.

Es gibt zwei immer wiederkehrende Leitmotive in diesen Mustern. Zum einem kapseln sie alle das Wissen um die konkreten vom System verwendeten Klassen. Zum anderen verstecken sie, wie Exemplare dieser Klassen erzeugt und zusammengefügt werden. Alles, was die Anwendung insgesamt über die Objekte weiß, wird durch die von den abstrakten Klassen definierten Schnittstellen bestimmt. Somit ermöglichen die Erzeugungsmuster zu bestimmen, *was* erzeugt wird, *wer* es erzeugt, *wie* es erzeugt und *wann* es erzeugt wird. Sie ermöglichen es Ihnen, ein System mit Hilfe von »Produktobjekten« zu konfigurieren, die stark in Struktur und Funktionalität variieren können. Die Konfiguration kann statisch (das heißt, zur Übersetzungszeit festgelegt) oder dynamisch sein (das heißt, zur Laufzeit festgelegt).

Mitunter stehen Erzeugungsmuster in Konkurrenz zueinander. Zum Beispiel gibt es Situationen, in denen sowohl ein Prototyp (144) als auch eine abstrakte Fabrik (107) nutzbringend eingesetzt werden können. Manchmal sind die Muster komplementär: Ein Erbauer (119) kann jeweils eines der anderen Muster zum Zusammenbau von Komponenten verwenden. Prototyp (144) kann ein Singleton (157) zu seiner Implementierung verwenden.

Da die Erzeugungsmuster eng zusammenhängen, werden wir alle fünf Muster zusammen betrachten, um ihre Ähnlichkeiten und ihre Unterschiede herauszustellen. Wir werden zudem ein bekanntes Beispiel – den Bau eines Labyrinths für ein Computerspiel – verwenden, um ihre Implementierungen zu illustrieren. Das Labyrinth und das Spiel werden von Muster zu Muster leicht variieren. Manchmal

besteht das Spiel lediglich darin, aus dem Labyrinth herauszufinden. In diesem Fall hat der Spieler vermutlich nur einen lokalen Blick auf das Labyrinth. Manchmal werden die Labyrinthe Probleme und Gefahren enthalten, welche es zu lösen und zu überwinden gilt. Diese Spiele werden möglicherweise eine Karte des bereits untersuchten Teils des Labyrinths anbieten.

Wir werden viele Details ignorieren, welche in einem Labyrinth vorhanden sein können, und ob das Labyrinthspiel einen oder mehrere Spieler kennt. Statt dessen konzentrieren wir uns auf die Erzeugung von Labyrinthen. Wir definieren ein Labyrinth als eine Menge von Räumen. Ein Raum kennt seine Nachbarobjekte, entweder einen weiteren Raum, eine Wand oder eine Tür zu einem anderen Raum.

Die Klassen `Raum`, `Tuer` und `Wand` definieren die Komponenten des in allen unseren Beispielen verwendeten Labyrinths. Wir definieren nur solche Teile dieser Klassen, welche wichtig zum Erzeugen eines Labyrinths sind. Wir werden die Spieler, die Operationen zum Anzeigen und Herumspazieren im Labyrinth, und alle weitere wichtige aber zum Bauen des Labyrinths irrelevante Funktionalität ignorieren.

Die Abbildung 3.1 zeigt die Beziehungen zwischen diesen Klassen.

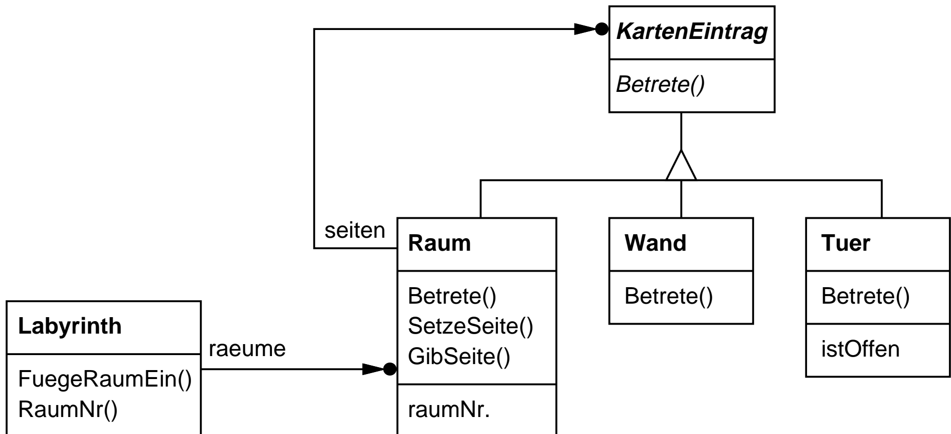

Abbildung 3.1

Jeder Raum besitzt vier Seiten. Wir verwenden den Aufzählungstyp `Richtung` für die C++-Implementierungen, um die nördliche, südliche, östliche und westliche Seite eines Raums anzugeben:

```
enum Richtung { Norden, Sueden, Osten, Westen };
```

Die Smalltalk-Implementierungen verwenden entsprechende Symbole, um diese Richtungen zu repräsentieren.

Die Klasse `KartenEintrag` ist die gemeinsame abstrakte Oberklasse für alle Komponenten eines Labyrinths. Um das Beispiel zu vereinfachen, definiert `KartenEintrag` nur eine Operation, `Betrete`. Ihre Bedeutung hängt davon ab, was man betritt. Wenn Sie einen Raum betreten, ändert sich Ihre Position. Wenn Sie versuchen, eine Tür zu betreten, können zwei Dinge passieren: Wenn die Tür offen ist, gehen Sie in den nächsten Raum. Wenn die Tür geschlossen ist, stoßen Sie sich Ihre Nase.

```
class KartenEintrag {
public:
   virtual void Betrete() = 0;
};
```

Betrete bietet einen einfachen Ausgangspunkt für kompliziertere Spieloperationen. Wenn Sie zum Beispiel in einem Raum sind und »Gehe nach Osten« sagen, kann das Spiel einfach bestimmen, welcher `KartenEintrag` direkt im Osten liegt und dann `Betrete` von ihm aufrufen. Die unterklassenspezifische `Betrete`-Operation bestimmt dann, ob sich Ihre Position geändert hat oder Ihre Nase verletzt wurde. In einem richtigen Spiel könnte `Betrete` das zu bewegende Spielerobjekt als Argument erhalten.

`Raum` ist die konkrete Unterklasse von `KartenEintrag`, welche die zentralen Beziehungen zwischen Komponenten in einem Labyrinth definiert. Sie verwaltet Referenzen zu anderen `KartenEintrag`-Objekten und speichert eine Raumnummer. Die Nummer dient zur Identifizierung von Räumen im Labyrinth.

```
class Raum : public KartenEintrag {
public:
   Raum(int raumNr);

   KartenEintrag* GibSeite(Richtung) const;
   void SetzeSeite(Richtung, KartenEintrag*);

   virtual void Betrete();

private:
   KartenEintrag* _seiten[4];
   int _raumNr;
};
```

Die folgenden Klassen repräsentieren die Wände oder Türen, welche auf jeder Seite eines Raumes auftreten.

```
class Wand : public KartenEintrag {
public:
   Wand();

   virtual void Betrete();
};

class Tuer : public KartenEintrag {
public:
   Tuer(Raum* = 0, Raum* = 0);

   virtual void Betrete();
   Raum* AndereSeite(Raum*);

private:
   Raum* _raum1;
   Raum* _raum2;
   bool _istOffen;
};
```

Wir müssen allerdings mehr als nur die Teile eines Labyrinths bestimmen. Wir definieren weiterhin eine Klasse `Labyrinth`, um eine Sammlung von Räumen zu repräsentieren. `Labyrinth` kann einen bestimmten Raum anhand einer Raumnummer unter Verwendung der `RaumNr`-Operation finden.

```
class Labyrinth {
public:
   Labyrinth();

   void FuegeRaumHinzu(Raum*);
   Raum* RaumNr(int) const;

private:
      // ...
};
```

RaumNr könnte den Raum mittels linearer Suche, einer Hash-Tabelle oder auch nur eines einfachen Arrays ermitteln. Wir werden uns um diese Details aber hier nicht kümmern. Statt dessen werden wir uns auf die Spezifikation der Komponenten im Labyrinth konzentrieren.

Als weitere Klasse definieren wir `LabyrinthSpiel`, welche das Labyrinth erzeugt. Man kann ein Labyrinth einfach durch eine Abfolge von Operationen erzeugen, welche dem Labyrinth Komponenten hinzufügen und sie dann miteinander verbinden. Die folgende Member-Funktion erzeugt zum Beispiel ein Labyrinth, das aus zwei Räumen mit einer Tür dazwischen besteht:

```
Labyrinth* LabyrinthSpiel::ErzeugeLabyrinth() {
   Labyrinth* einLabyrinth = new Labyrinth;
   Raum* raum1 = new Raum(1);
   Raum* raum2 = new Raum(2);
   Tuer* dieTuer = new Tuer(raum1, raum2);

   einLabyrinth->FuegeRaumHinzu(raum1);
   einLabyrinth->FuegeRaumHinzu(raum2);

   raum1->SetzeSeite(Norden, new Wand);
   raum1->SetzeSeite(Osten, dieTuer);
   raum1->SetzeSeite(Sueden, new Wand);
   raum1->SetzeSeite(Westen, new Wand);

   raum2->SetzeSeite(Norden, new Wand);
   raum2->SetzeSeite(Osten, new Wand);
   raum2->SetzeSeite(Sueden, new Wand);
   raum2->SetzeSeite(Westen, dieTuer);

   return einLabyrinth;
}
```

Diese Operation ist ziemlich kompliziert, bedenkt man, daß sie lediglich ein Labyrinth mit zwei Räumen erzeugt. Offenkundig geht es auch einfacher. Zum Beispiel könnte der `Raum`-Konstruktor im voraus die Seiten mit Wänden initialisieren. Aber dies würde den Code nur an eine andere Stelle bewegen. Das eigentliche Problem mit dieser Member-Funktion ist nicht ihre Größe, sondern ihre *Unflexibilität*. Sie schreibt das Labyrinth-layout fest. Eine Veränderung des Layouts bedeutet das Verändern der Member-Funktion, entweder durch Überschreiben oder durch Ändern von Teilen der Implementierung. Die erste Alternative führt zur Reimplementierung der Operation während die zweite Alternative fehleranfällig ist und auch keine Wiederverwendung fördert.

Die Erzeugungsmuster zeigen, wie man den Entwurf *flexibler*, aber nicht notwendig kleiner macht. Insbesondere erleichtern sie es, die Klassen, welche die Komponenten des Labyrinth bestimmen, zu verändern.

Stellen Sie sich vor, Sie wollen ein existierendes Labyrinthlayout für ein neues Spiel wiederverwenden, welches unter anderem verzauberte Labyrinthe enthält. Das verzauberte Labyrinthspiel besitzt neue Arten von Komponenten, etwa `TuerMitZauberspruch`, eine Tür, die nur mit einem Zauberspruch verschlossen und geöffnet werden kann; oder `VerzauberterRaum`, ein Raum der merkwürdige Gegenstände wie magische Schlüssel oder Zaubersprüche enthalten kann. Wie nun kann man `ErzeugeLabyrinth` auf einfache Weise so verändern, daß es Labyrinthe mit diesen neuen Klassen von Objekten erzeugt?

Die größte Hürde für Veränderungen liegt hier in der unflexiblen Programmierung der Klassen, von denen Objekte erzeugt werden sollen. Die Erzeugungsmuster bieten verschiedene Möglichkeiten, explizite Referenzen auf konkrete Klassen aus dem Code, der von ihnen Objekte erzeugen muß, zu entfernen:

- Wenn `ErzeugeLabyrinth` virtuelle Funktionen anstelle von Konstruktoren zum Erzeugen der benötigten Räume, Wände und Türen aufruft, können Sie die Klassen der zu erzeugenden Objekte verändern, indem sie eine Unterklasse von `LabyrinthSpiel` erzeugen und die entsprechenden virtuellen Funktionen überschreiben. Dieser Ansatz ist ein Beispiel für das Fabrikmethodemuster (131).
- Wenn `ErzeugeLabyrinth` ein Objekt als Parameter erhält, das zum Erzeugen von Räumen, Wänden und Türen verwendet wird, dann können Sie die Klassen von Räumen, Wänden und Türen durch das Hereinreichen verschiedener Parameter verändern. Dies ist ein Beispiel für das Abstrakte-Fabrik-Muster (107).
- Wenn `ErzeugeLabyrinth` ein Objekt erhält, das ein neues Labyrinth vollständig unter Verwendung von Operationen zum Hinzufügen von Räumen, Türen und Wänden zum Labyrinth selbst erzeugen kann, dann können Sie Vererbung benutzen, um Teile des Labyrinths oder die Art, wie es gebaut wird, zu verändern. Dies ist ein Beispiel für das Erbauermuster (119).
- Wenn `ErzeugeLabyrinth` mit verschiedenen prototypischen Raum-, Tür- und Wandobjekten parametrisiert wird, welche es kopieren und dem Labyrinth hinzufügen kann, dann können Sie den Aufbau des Labyrinths durch Ersetzen dieser prototypischen Objekte verändern. Dies ist ein Beispiel für das Prototypmuster (144).

Das verbleibende Erzeugungsmuster, Singleton (157), ermöglicht es Ihnen, sicherzustellen, daß es nur ein Labyrinthobjekt pro Spiel gibt und daß alle Spielobjekte direkten Zugriff auf diese Objekte besitzen, ohne auf globale Variablen oder Funktionen zurückgreifen zu müssen. Singleton macht es weiterhin einfach, das Labyrinth zu erweitern oder zu ersetzen, ohne existierenden Code verändern zu müssen.

Abstrakte Fabrik

(Abstract Factory)

Ein objektbasiertes Erzeugungsmuster

Zweck

Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

Auch bekannt als

Kit

Motivation

Stellen Sie sich eine Klassenbibliothek für Benutzungsschnittstellen vor, die mehrere Look-and-Feel-Standards wie Motif oder den Presentation-Manager unterstützt. Unterschiedliche Look-and-Feel-Standards definieren unterschiedliches Aussehen und Verhalten von Widgets, den Interaktionselementen einer Benutzungsschnittstelle, wie Scrollbars, Fenstern und Knöpfen. Um zwischen verschiedenen Look-and-Feel-Standards portierbar zu sein, sollte sich eine Anwendung nicht auf die Widgets eines spezifischen Standards festlegen. Die Erzeugung von Look-and-Feel spezifischen Widgetklassen über die ganze Anwendung zu verteilen macht es schwer, das Look-and-Feel später zu ändern.

Man kann das Problem durch Einführung einer abstrakten WidgetFabrik lösen, die eine Schnittstelle zum Erzeugen jeder grundlegenden Art von Widget deklariert (siehe Abbildung 3.2). Weiterhin gibt es eine abstrakte Klasse für jede Widgetart sowie konkrete Unterklassen, welche die Widgets für den jeweiligen Look-and-Feel-Standard implementieren. Die Schnittstelle der WidgetFabrik besitzt für jede abstrakte Widgetklasse eine Operation, die ein neues Widget zurück gibt. Klienten rufen diese Operationen auf, um Exemplare von Widgets zu erzeugen, ohne dabei die konkreten Klassen zu kennen, die sie benutzen. Somit bleiben sie unabhängig vom aktuellen Look-and-Feel.

Für jeden Look-and-Feel-Standard gibt es eine konkrete Unterklasse von WidgetFabrik. Jede Unterklasse implementiert die Operationen zum Erzeugen des passenden Widgets für ein Look-and-Feel. Die ErzeugeScrollbar-Operation der MotifWidgetFabrik zum Beispiel erzeugt einen Scrollbar für Motif und gibt ihn zurück,

während die entsprechende Operation der PMWidgetFabrik einen Scrollbar für den Presentation-Manager zurückliefert. Klienten erzeugen die Widgets ausschließlich über die Schnittstelle der WidgetFabrik und kennen die Klassen nicht, welche das Widget für ein bestimmtes Look-and-Feel implementieren. Mit anderen Worten, Klienten stützen sich immer nur auf eine durch eine abstrakte Klasse definierte Schnittstelle, nicht aber auf eine bestimmte konkrete Klasse.

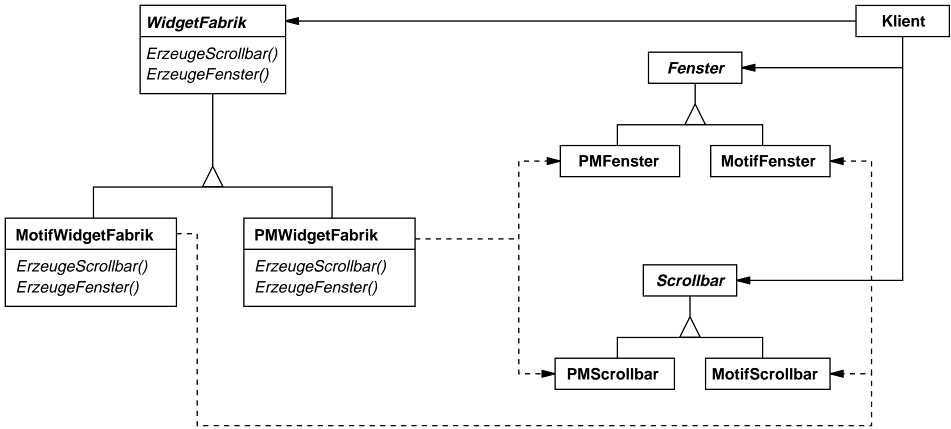

Abbildung 3.2

Eine Widgetfabrik sichert zudem Abhängigkeiten zwischen konkreten Widgetklassen ab. Ein Motif-Scrollbar sollte nur mit einem Motif-Knopf und einem Motif-Texteditor zusammen verwendet werden. Diese Konsistenzbedingung wird automatisch als Konsequenz des Einsatzes einer MotifWidgetFabrik sichergestellt.

Struktur

Abbildung 3.3 zeigt die Struktur des Abstrakte-Fabrik-Musters.

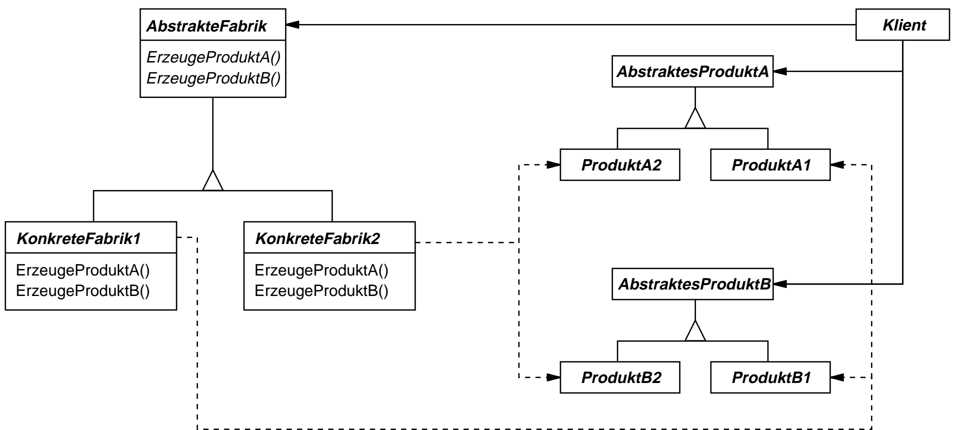

Abbildung 3.3

Anwendbarkeit

Verwenden Sie das Abstrakte-Fabrik-Muster, wenn

- ein System unabhängig davon sein soll, wie seine Produkte[1] erzeugt, zusammengesetzt und repräsentiert werden.
- ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.
- eine Familie von verwandten Produktobjekten entworfen wurde, zusammen verwendet zu werden, und Sie diese Konsistenzbedingung sicherstellen müssen.
- Sie eine Klassenbibliothek von Produkten anbieten möchten, von denen Sie nur die Schnittstellen, nicht aber ihre Implementierungen offenlegen möchten.

Teilnehmer

- **AbstrakteFabrik** (WidgetFabrik)
  - deklariert eine abstrakte Schnittstelle für Operationen, die konkrete Produktobjekte erzeugen.

1. Unter »Produkten« sind hier die vom System erzeugten Objekte zu verstehen. Anm. D.R.

- **KonkreteFabrik** (MotifWidgetFabrik, PMWidgetFabrik)
  - implementiert die Operation zur Erzeugung konkreter Produktobjekte.
- **AbstraktesProdukt** (Fenster, Scrollbar)
  - deklariert eine Schnittstelle für einen bestimmten Typ von Produktobjekten.
- **KonkretesProdukt** (MotifFenster, MotifScrollbar)
  - definiert ein von der entsprechenden konkreten Fabrik zu erzeugendes Produktobjekt.
  - implementiert die AbstraktesProdukt-Schnittstelle.
- **Klient**
  - verwendet nur die Schnittstellen, welche von den AbstrakteFabrik- und Abstraktes-Produkt-Klassen deklariert werden.

Interaktionen

- Normalerweise wird ein einzelnes Exemplar der KonkreteFabrik-Klasse zur Laufzeit erzeugt. Diese konkrete Fabrik erzeugt Produktobjekte, welche spezifische Implementierungen haben. Um verschiedene Produktobjekte zu erzeugen, sollten Klienten unterschiedliche konkrete Fabriken haben.
- Eine AbstrakteFabrik verlagert die Erzeugung von Produktobjekten auf ihre KonkreteFabrik-Unterklassen.

Konsequenzen

Das Abstrakte-Fabrik-Muster hat die folgenden Vorteile und Verbindlichkeiten:

1. *Isolation konkreter Klassen.* Das Abstrakte-Fabrik-Muster ermöglicht es Ihnen, die Klassen von Objekten zu steuern, welche ihre Anwendung erzeugt. Da eine Fabrik für den Prozeß des Erzeugens von Produktobjekten zuständig ist und ihn kapselt, isoliert es Klienten von den Implementierungsklassen. Klienten manipulieren Objekte nur durch ihre abstrakten Schnittstellen. Die Namen von Produktklassen sind in der Implementierung der konkreten Fabrik isoliert; sie erscheinen nicht im Klientencode.
2. *Einfacher Austausch von Produktfamilien.* Die Klasse einer konkreten Fabrik erscheint nur einmal in der Anwendung – genau dort, wo von ihr ein Exemplar erzeugt wird. Dies macht es einfach, die von einer Anwendung benutzte kon-

krete Fabrik auszutauschen. Sie kann verschiedene Produktkonfigurationen einfach durch den Austausch der konkreten Fabrik verwenden. Da eine abstrakte Fabrik eine komplette Familie von Produkten erzeugt, wird die gesamte Produktfamilie auf einmal getauscht. In unserem Benutzungsschnittstellenbeispiel können wir von Motif-Widgets zu Presentation-Manager-Widgets einfach dadurch wechseln, daß wir die entsprechenden Fabrikobjekte austauschen und die Benutzungsschnittstelle erneut erzeugen.

3. *Konsistenzsicherung unter Produkten.* Wenn Produktobjekte einer Familie entworfen werden, um zusammenzuarbeiten, ist es wichtig, daß eine Anwendung nur Objekte einer Familie zur Zeit verwendet. Eine abstrakte Fabrik macht es einfach, dies sicherzustellen.

4. *Schwierige Unterstützung neuer Produkte.* Die Erweiterung abstrakter Fabriken, um neue Arten von Produkte zu produzieren, ist nicht einfach. Dies liegt daran, daß die Schnittstelle einer abstrakten Fabrik die Menge von Produkten, die erzeugt werden können, festlegt. Die Unterstützung neuer Arten von Produkten erfordert es, die Schnittstelle der Fabrik zu erweitern, was dazu führt, die AbstrakteFabrik-Klasse und all ihre Unterklassen zu verändern. Wir diskutieren eine Lösung für dieses Problem im folgenden Abschnitt.

Implementierung

Es gibt verschiedene nützliche Techniken, eine abstrakte Fabrik zu implementieren.

1. *Fabriken als Singletons.* Eine Anwendung braucht üblicherweise genau ein Exemplar einer konkreten Fabrik pro Produktfamilie. Deswegen implementiert man sie am besten als Singleton (157).

2. *Erzeugen von Produkten.* AbstrakteFabrik deklariert lediglich eine Schnittstelle zum Erzeugen von Produkten. Es bleibt den KonkreteFabrik-Unterklassen überlassen, sie tatsächlich zu erzeugen. Üblicherweise definiert man Fabrikmethoden (131) für jedes Produkt. Eine konkrete Fabrik bringt ihre Produkte ins Spiel, indem sie für jedes die entsprechende Fabrikmethode überschreibt. Diese Implementierung ist zwar einfach, erfordert aber eine neue KonkreteFabrik-Unterklasse für jede Produktfamilie, selbst wenn die Produktfamilien sich nur wenig unterscheiden.

   Wenn es viele Produktfamilien geben kann, bietet es sich an, die konkrete Fabrik mit Hilfe des Prototypmusters (144) zu implementieren. Die konkrete Fabrik wird mit einem prototypischen Exemplar eines jeden Produkts aus der Familie initialisiert, und sie erzeugt neue Produkte durch das Klonen ihres Pro-

totypen. Der prototypenbasierte Ansatz vermeidet es, für jede neue Produktfamilie eine neue konkrete Fabrik einführen zu müssen.

Es folgt ein Beispiel zur Implementierung einer prototypenbasierten Fabrik in Smalltalk. Die konkrete Fabrik speichert die zu klonenden Prototypen in einem Dictionary namens `teilKatalog`. Die Methode `erzeuge:` sucht den Prototypen heraus und klont ihn:

```
erzeuge: teilName
   ^ (teilKatalog at: teilName) copy
```

Die konkrete Fabrik verfügt über eine Methode, Prototypen in das Dictionary einzufügen:

```
fuegeTeilHinzu: teilPrototyp mitNamen: teilName
   teilKatalog at: teilName put: teilPrototyp
```

Prototypen werden der Fabrik hinzugefügt, indem man sie mittels eines Symbols identifiziert:

```
eineFabrik fuegeTeilHinzu: einPrototyp mitNamen: #ACMEWidget
```

Sprachen wie Smalltalk oder Objective-C, in denen Klassen Objekte erster Ordnung sind, bieten eine Variante des prototypenbasierten Ansatzes. In diesen Sprachen können Sie eine Klasse als degenerierte Fabrik auffassen, welche genau eine Art von Produkt erzeugt. Sie können *Klassen*, welche die verschiedenen Produkte erzeugen, genau wie Prototypen in den Variablen einer konkreten Fabrik speichern. Auf die Veranlassung der Fabrik hin erzeugen diese Klassen neue Exemplare. Sie definieren eine neue Fabrik durch die Initialisierung eines Exemplars einer konkreten Fabrik mit *Klassen* von Produkten, anstatt weitere Unterklassen zu bilden. Dieser Ansatz basiert auf spezifischen Spracheigenschaften, während der reine Prototypenansatz sprachunabhängig ist.

Die klassenbasierte Version wird ebenso wie die eben diskutierte prototypenbasierte Fabrik in Smalltalk zu einer einzelnen Variable `teilKatalog` führen, welche ein Dictionary ist, dessen Schlüssel der Name der Klasse ist. Statt die zu klonenden Prototypen zu speichern, speichert `teilKatalog` die Klassen der Produkte. Die Methode `erzeuge:` sieht nun folgendermaßen aus:

```
erzeuge: teilName
   ^ (teilKatalog at: teilName) new
```

3. *Definieren von erweiterbaren Fabriken.* AbstrakteFabrik definiert üblicherweise verschiedene Operationen für jede Art von Produkten, die es erzeugen kann. Die Produkttypen sind in den Operationssignaturen festgelegt. Will man eine neue Art von Produkt hinzufügen, so muß man die Schnittstelle von AbstrakteFabrik und die aller Klassen, die davon abhängen, verändern.

   Ein flexiblerer, wenngleich weniger sicherer Entwurf ist es, die objekterzeugenden Operationen um einen Parameter zu erweitern. Dieser Parameter spezifiziert den Typ des Objekts, welches zu erzeugen ist. Dies mag eine Klassenidentifizierung, ein Integer, ein String oder irgendein anderer Wert sein, der den Produkttyp identifiziert. Bei diesem Ansatz benötigt AbstrakteFabrik lediglich eine einzige Erzeuge-Operation mit einem Parameter, welcher den Typ des zu erzeugenden Objekts identifiziert. Dies ist die Technik, welche in den zuvor diskutierten prototypen- und klassenbasierten abstrakten Fabriken verwendet wurde.

   Diese Variante läßt sich in dynamisch typisierten Sprachen wie Smalltalk leichter verwenden als in statisch typisierten Sprachen wie C++. Sie können sie in C++ nur dann benutzen, wenn alle Objekte dieselbe abstrakte Basisklasse haben oder wenn der Klient die verlangten Produktobjekte sicher in den richtigen Typ konvertieren kann. Die Implementierung von Fabrikmethoden (131) zeigt, wie man derart parametrierbare Operationen in C++ implementieren kann.

   Aber selbst, wenn man keine Typkonvertierung benötigt, bleibt ein grundlegendes Problem: Alle an den Klienten zurückgegebenen Produkte haben *dieselbe* abstrakte durch den Rückgabetyp festgelegte Schnittstelle. Der Klient kann die Objekte weder unterscheiden noch sichere Annahmen über die Klassen der Objekte machen. Klienten können keine unterklassenspezifischen Operationen über die abstrakte Schnittstelle verwenden. Es bleibt dem Klient freigestellt, einen Downcast ausführen, zum Beispiel mittels `dynamic_cast` in C++. Dies ist aber nicht immer sinnvoll und sicher, da der Downcast fehlschlagen kann. Dies ist der klassische Nachteil, den man für hochflexible und erweiterbare Schnittstellen in Kauf nehmen muß.

Beispielcode

Wir werden nun das Abstrakte-Fabrik-Muster verwenden, um die zu Beginn des Kapitels diskutierten Labyrinthe zu erzeugen.

Die Klasse `LabyrinthFabrik` ist in der Lage, Komponenten eines Labyrinths zu erzeugen. Sie erstellt Räume, Wände und Türen zwischen den Räumen. Sie kann

von einem Programm verwendet werden, das Pläne für Labyrinthe aus einer Datei liest und das entsprechende Labyrinth erstellt. Oder sie kann von einem Programm verwendet werden, das Labyrinthe zufallsbasiert zusammenbaut. Programme, die Labyrinthe erstellen, erhalten eine `LabyrinthFabrik` als Argument, so daß der Programmierer die Klassen der zu erzeugenden Räume, Wände und Türen festlegen kann.

```
class LabyrinthFabrik {
public:
   LabyrinthFabrik();

   virtual Labyrinth* ErzeugeLabyrinth() const
      { return new Labyrinth; }
   virtual Wand* ErzeugeWand() const
      { return new Wand; }
   virtual Raum* ErzeugeRaum(int n) const
      { return new Raum(n); }
   virtual Tuer* ErzeugeTuer(Raum* raum1, Raum* raum2) const
      { return new Tuer(raum1, raum2); }
};
```

Wie bereits angeführt erzeugt die Member-Funktion `ErzeugeLabyrinth` (Seite 105) ein kleines Labyrinth, das aus zwei Räumen mit einer Tür dazwischen besteht.

`ErzeugeLabyrinth` schreibt die Klassennamen im Code fest, so daß es schwierig wird, Labyrinthe mit anderen Komponenten zu erzeugen.

Es folgt eine Version von `ErzeugeLabyrinth`, die einen Parameter `LabyrinthFabrik` entgegen nimmt und so die Probleme behebt:

```
Labyrinth* LabyrinthSpiel::ErzeugeLabyrinth(
   LabyrinthFabrik& fabrik)
{
   Labyrinth* einLabyrinth = fabrik.ErzeugeLabyrinth();
   Raum* raum1 = fabrik.ErzeugeRaum(1);
   Raum* raum2 = fabrik.ErzeugeRaum(2);
   Tuer* eineTuer = fabrik.ErzeugeTuer(raum1, raum2);

   einLabyrinth->FuegeRaumHinzu(raum1);
   einLabyrinth->FuegeRaumHinzu(raum2);

   raum1->SetzeSeite(Norden, fabrik.ErzeugeWand());
   raum1->SetzeSeite(Osten, eineTuer);
   raum1->SetzeSeite(Sueden, fabrik.ErzeugeWand());
   raum1->SetzeSeite(Westen, fabrik.ErzeugeWand());
```

```
    raum2->SetzeSeite(Norden, fabrik.ErzeugeWand());
    raum2->SetzeSeite(Osten, fabrik.ErzeugeWand());
    raum2->SetzeSeite(Sueden, fabrik.ErzeugeWand());
    raum2->SetzeSeite(Westen, eineTuer);
}
```

Wir können die Klasse `VerzaubertesLabyrinthFabrik` erzeugen, eine Fabrik für verzauberte Labyrinthe, indem wir eine Unterklasse von `LabyrinthFabrik` bilden. `VerzaubertesLabyrinthFabrik` überschreibt die verschiedenen Member-Funktionen und gibt Objekte verschiedener Unterklassen wie zum Beispiel `Raum` und `Wand` zurück.

```
class VerzaubertesLabyrinthFabrik : public LabyrinthFabrik {
public:
    VerzaubertesLabyrinthFabrik();

    virtual Raum* ErzeugeRaum(int n) const
        { return new VerzauberterRaum(n,
            BenoetigterZauberspruch()); }
    virtual Tuer* ErzeugeTuer(Raum* raum1, Raum* raum2) const
        { return new TuerMitZauberspruch(raum1, raum2); }

protected:
    Zauberspruch* BenoetigterZauberspruch() const;
};
```

Nun stellen Sie sich vor, daß wir ein Labyrinthspiel erstellen wollen, bei dem in jedem Raum eine Bombe plaziert werden kann. Wenn die Bombe in die Luft fliegt, beschädigt sie die Wände. Wir erstellen eine Unterklasse von `Raum`, welche vermerkt, ob ein Raum eine Bombe besitzt und ob sie in die Luft gegangen ist. Wir brauchen weiterhin eine Unterklasse von `Wand`, um den den Wänden zugefügten Schaden zu notieren. Wir nennen diese Klassen `RaumMitBombe` und `BombardierbareWand`.

Als letzte Klasse definieren wir `LabyrinthMitBombenFabrik`, eine Unterklasse von `LabyrinthFabrik`, die sicherstellt, daß die Wände Exemplare der Klasse `BombardierbareWand` und die Räume Exemplare der Klasse `RaumMitBombe` sind.

`LabyrinthMitBombenFabrik` braucht dazu nur zwei Funktionen zu überschreiben:

```
Wand* LabyrinthMitBombenFabrik::ErzeugeWand() const {
    return new BombardierbareWand;
}
```

```
Raum* LabyrinthMitBombenFabrik::ErzeugeRaum(int n) const {
   return new RaumMitBombe(n);
}
```

Wollen wir ein einfaches Labyrinth erstellen, das Bomben enthalten kann, so rufen wir einfach `ErzeugeLabyrinth` mit dem Parameter `LabyrinthMitBombenFabrik` auf.

```
LabyrinthSpiel spiel;
LabyrinthMitBombenFabrik fabrik;

spiel.ErzeugeLabyrinth(fabrik);
```

ErzeugeLabyrinth ist gleichermaßen gut in der Lage, mit einem Exemplar von `VerzaubertesLabyrinthFabrik` verzauberte Labyrinthe zu erzeugen.

Es ist bemerkenswert, daß `LabyrinthFabrik` lediglich aus einer Sammlung von Fabrikmethoden besteht. Dies ist die naheliegendste Art und Weise, das Abstrakte-Fabrik-Muster zu implementieren. Es ist weiterhin interessant, daß `LabyrinthFabrik` keine abstrakte Klasse ist; sie fungiert somit gleichermaßen als die abstrakte *und* konkrete Fabrik. Dies ist eine weitere naheliegende Implementierung, um das Abstrakte-Fabrik-Muster einfach anzuwenden. Da die `LabyrinthFabrik` eine konkrete Klasse ist, die vollständig aus Fabrikmethoden besteht, kann man eine neue `LabyrinthFabrik` einfach durch Erstellen einer Unterklasse und Überschreiben der zu ändernden Operation erzeugen.

`ErzeugeLabyrinth` verwendet die `SetzeSeite`-Operation von Räumen, um ihre Seiten zu spezifizieren. Wenn es Räume mit einer `LabyrinthMitBombenFabrik` erzeugt, wird das Labyrinth aus `RaumMitBombe`-Objekten mit `BombardierbareWand`-Seiten bestehen. Wenn `RaumMitBombe` auf eine unterklassenspezifische Operation von `BombardierbareWand` zugreifen muß, so muß sie die Referenzen auf ihre Wände von `Wand*` nach `BombardierbareWand*` konvertieren. Dies ist so lange sicher, wie das Argument tatsächlich auch eine `BombardierbareWand` ist, was genau dann garantiert ist, wenn alle Wände mit einer `LabyrinthMitBombenFabrik` erzeugt werden.

Dynamisch typisierte Sprachen wie Smalltalk benötigen natürlich keinen Downcast, aber sie können Laufzeitfehler produzieren, wenn sie auf eine `Wand` stoßen, eigentlich aber eine *Unterklasse* von `Wand` erwarten. Die Verwendung von AbstrakteFabrik zum Erzeugen von Wänden hilft diese Laufzeitfehler zu verhindern, indem sichergestellt wird, daß nur bestimmte Wände erzeugt werden können.

Betrachten wir eine Smalltalk-Version einer `LabyrinthFabrik`, die eine einzige `Erzeuge`-Operation besitzt, welche die Art des zu erzeugenden Objekts als Parameter erhält. Weiterhin speichert die konkrete Fabrik die Klassen der Produkte, die sie erzeugt.

Zuerst schreiben wir eine äquivalente Version von `ErzeugeLabyrinth` in Smalltalk:

```
erzeugeLabyrinth: eineFabrik
    | raum1 raum2 eineTuer |
    raum1 := (eineFabrik erzeuge: #raum) nummer: 1.
    raum2 := (eineFabrik erzeuge: #raum) nummer: 2.
    eineTuer := (eineFabrik erzeuge: #tuer)
    von: raum1 nach: raum2.
    raum1 aufSeite: #norden setze: (eineFabrik erzeuge: #wand).
    raum1 aufSeite: #osten setze: eineTuer.
    raum1 aufSeite: #sueden setze: (eineFabrik erzeuge: #wand).
    raum1 aufSeite: #westen setze: (eineFabrik erzeuge: #wand).
    raum2 aufSeite: #norden setze: (eineFabrik erzeuge: #wand).
    raum2 aufSeite: #osten setze: (eineFabrik erzeuge: #wand).
    raum2 aufSeite: #sueden setze: (eineFabrik erzeuge: #wand).
    raum2 aufSeite: #osten setze: eineTuer.
    ^ Labyrinth new
      fuegeRaumHinzu: raum1;
      fuegeRaumHinzu: raum2;
      yourself
```

Wie wir bereits im Implementierungsabschnitt diskutiert haben, benötigt `Labyrinth-Fabrik` lediglich eine einzige Exemplarvariable namens `teilKatalog`, um ein Dictionary bereitzustellen, dessen Schlüssel der Klassenname der jeweiligen Klasse ist. Weiterhin sei ins Gedächtnis gerufen, wie wir die `erzeuge:`-Methode implementiert haben:

```
erzeuge: teilName
    ^ (teilKatalog at: teilName) new
```

Wir können nun eine `LabyrinthFabrik` erstellen und sie zur Implementierung von `erzeugeLabyrinth` verwenden. Wir erzeugen die Fabrik mittels einer Methode `erzeugeLabyrinthFabrik` der Klasse `LabyrinthSpiel`.

```
erzeugeLabyrinthFabrik
    ^ (LabyrinthFabrik new
        fuegeTeilHinzu: Wand namens: #wand;
        fuegeTeilHinzu: Raum namens: #raum;
        fuegeTeilHinzu: Tuer namens #tuer;
        yourself)
```

Man erzeugt eine `LabyrinthMitBombenFabrik` oder eine `VerzaubertesLabyrinth-Fabrik`, indem man verschiedene Klassen mit den jeweiligen Schlüsseln assoziiert. Eine `VerzaubertesLabyrinthFabrik` kann zum Beispiel folgendermaßen erzeugt werden:

```
erzeugeLabyrinthFabrik
   ^ (LabyrinthFabrik new
      fuegeTeilHinzu: Wand namens #wall;
      fuegeTeilHinzu: VerzauberterRaum namens: #raum;
      fuegeTeilHinzu: TuerMitZauberspruch namens: #tuer;
      yourself)
```

Bekannte Verwendungen

InterViews verwendet die »Kit«-Nachsilbe [Lin92], um AbstrakteFabrik-Klassen zu kennzeichnen. Es definiert die abstrakten Fabriken WidgetKit und DialogKit, um Look-and-Feel-spezifische Benutzungsschnittstellenobjekte zu erzeugen. InterViews bietet ebenfalls ein LayoutKit, das je nach gewünschtem Layout verschiedene Kompositionsobjekte generiert. Zum Beispiel benötigt ein konzeptuell horizontales Layout je nach Orientierung des Dokuments (Portrait oder Landschaft) möglicherweise unterschiedliche Kompositionsobjekte.

ET++ [WGM88] verwendet das Abstrakte-Fabrik-Muster, um über verschiedene Fenstersysteme wie X-Windows und SunView hinweg portabel zu sein. Die abstrakte Basisklasse WindowSystem definiert die Schnittstelle zum Erzeugen von Objekten, welche die Ressourcen eines Fenstersystems repräsentieren. Beispiele für diese Operationen sind MakeWindow, MakeFont und MakeColor. Konkrete Unterklassen implementieren die Schnittstellen für ein spezifisches Fenstersystem. Zur Laufzeit erzeugt ET++ ein Exemplar einer konkreten WindowSystem-Unterklasse, welches ihrerseits die konkreten Objekte für Systemressourcen erzeugt.

Verwandte Muster

Die AbstrakteFabrik-Klassen werden oft durch Fabrikmethoden (131) implementiert. Sie können auch mit Hilfe des Prototypmusters (144) implementiert werden.

Eine konkrete Fabrik ist oftmals ein Singleton (157).

Erbauer

(Builder)

Ein objektbasiertes Erzeugungsmuster

Zweck

Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so daß derselbe Konstruktionsprozeß unterschiedliche Repräsentationen erzeugen kann.

Motivation

Ein Einleser für das RTF (Rich Text Format)-Dokumentaustauschformat ist ein Objekt, das Dokumente dieses Formats einlesen und in eine interne Repräsentation umsetzen kann. Es sollte in der Lage sein, RTF in viele verschiedene Textformate konvertieren zu können, so zum Beispiel in reinen ASCII-Text oder in ein interaktiv editierbares Textwidget. Das Problem ist allerdings, daß die Anzahl möglicher Konvertierungen unbeschränkt ist. Deswegen sollte es einfach möglich sein, eine neue Konvertierung einzuführen, ohne den Einleser modifizieren zu müssen.

Eine Lösung besteht darin, die RTFLeser-Klasse mit einem TextKonvertierer-Objekt zu konfigurieren, welches das RTF-Dokument in eine andere Repräsentation konvertiert (siehe Abbildung 3.4). Während der RTFLeser das RTF-Dokument einliest und parst, verwendet es den TextKonvertierer, um die Konvertierung auszuführen. Immer wenn der RTFLeser ein RTF-Token erkennt (entweder einfachen Text oder ein RTF-Steuerwort), stellt es eine Anfrage an den TextKonvertierer, das Token zu konvertieren. Die TextKonvertierer-Objekte sind sowohl für die Ausführung der Datenkonvertierung als auch für die Repräsentation des Tokens in einem bestimmten Format zuständig.

Unterklassen von TextKonvertierer sind Spezialisierungen für unterschiedliche Konvertierungen und Formate. Beispielsweise ignoriert ein ASCIIKonvertierer alle Konvertierungsanfragen bis auf solche, die sich auf reinen Text beziehen. Ein TeXKonvertierer hingegen implementiert alle Anfrageoperationen, um eine TeX-Repräsentation zu erzeugen, die alle stilistischen Informationen des Texts enthält. Ein Text-WidgetKonvertierer wiederum produziert ein komplexes Benutzungsschnittstellenobjekt, das dem Benutzer den Text anzeigt und editieren läßt.

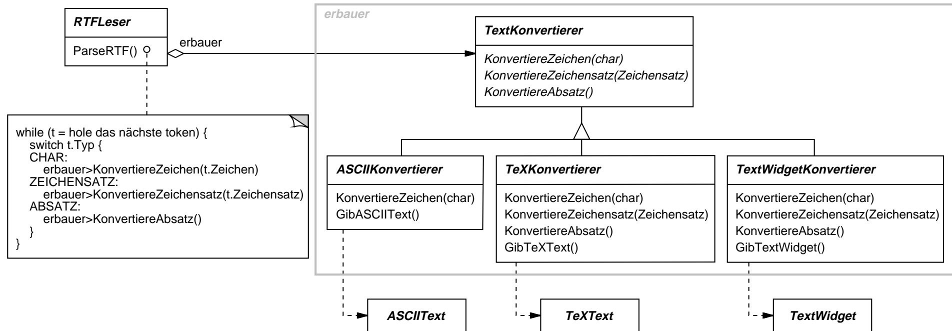

Abbildung 3.4

Jede Art von Konvertierer-Klasse versteckt den Mechanismus zum Erzeugen und Zusammenbauen eines komplexen Objekts hinter einer abstrakten Schnittstelle. Der Konvertierer ist von dem für das Einlesen eines RTF-Dokuments zuständigen Leser abgetrennt.

Das Erbauermuster erfaßt all diese Beziehungen. Jede Konvertierer-Klasse im Muster wird *Erbauer* genannt und jeder Leser *Direktor*. Wendet man das Muster auf das Beispiel an, so trennt das Erbauermuster den Algorithmus zur Interpretierung eines Textformats, also den Parser für RTF-Dokumente, von der Erzeugung und Repräsentation des konvertierten Formats. Dies ermöglicht es uns, den Parsealgorithmus des RTFLesers zum Erzeugen unterschiedlicher Textrepräsentationen von RTF-Dokumenten wiederzuverwenden – man konfiguriert lediglich den RTFLeser mit unterschiedlichen Unterklassen von TextKonvertierer.

Anwendbarkeit

Verwenden Sie das Erbauermuster in folgenden Situationen:

- Der Algorithmus zum Erzeugen eines komplexen Objekts soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden.
- Der Konstruktionsprozeß muß verschiedene Repräsentationen des zu konstruierenden Objekts erlauben.

Struktur

Abbildung 3.5 zeigt die Struktur des Erbauermusters.

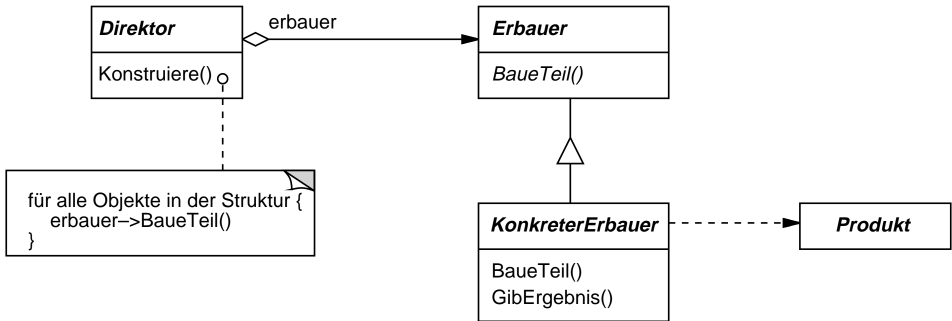

Abbildung 3.5

Teilnehmer

- **Erbauer** (TextKonvertierer)
  - spezifiziert eine abstrakte Schnittstelle zum Erzeugen von Teilen eines Produktobjekts.
- **KonkreterErbauer** (ASCIIKonvertierer, TeXKonvertierer, TextWidgetKonvertierer)
  - konstruiert und fügt Teile des Produkts zusammen, indem es die Erbauerschnittstelle implementiert.
  - definiert und verwaltet die von ihm erzeugte Repräsentation.
  - bietet eine Schnittstelle zum Zurückgeben des Produkts (zum Beispiel GibASCII-Text, GibTextWidget).
- **Direktor** (RTFLeser)
  - konstruiert ein Objekt unter Verwendung der Erbauerschnittstelle.
- **Produkt** (ASCIIText, TeXText, TextWidget)
  - repräsentiert das gerade konstruierte komplexe Objekt. Ein KonkreterErbauer erstellt die interne Repräsentation des Produkts und definiert den Prozeß, durch den es zusammengesetzt wird.
  - schließt Klassen ein, welche die konstituierenden Teile definieren. Dies umfaßt die Schnittstellen, mit denen die Teile zum endgültigen Resultat zusammengefügt werden.

Interaktionen

- Der Klient erzeugt das Direktorobjekt und konfiguriert es mit dem erwünschten Erbauerobjekt.
- Der Direktor informiert den Erbauer, wenn ein Teil des Produkts gebaut werden soll.
- Der Erbauer bearbeitet die Anfragen des Direktors und fügt Teile zum Produkt hinzu.
- Der Klient erhält das Produkt vom Erbauer.

Das Interaktionsdiagramm in Abbildung 3.6 illustriert, wie Erbauer und Direktor mit einem Klienten zusammenarbeiten:

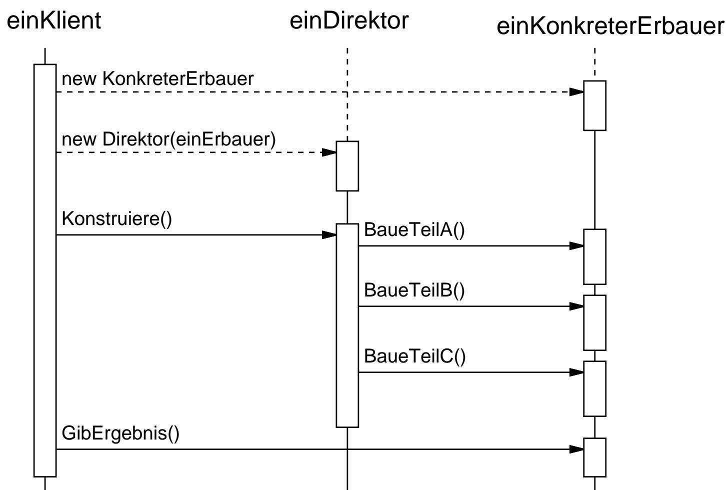

Abbildung 3.6

Konsequenzen

Es folgen zentrale Konsequenzen des Erbauermusters:

1. *Variation der internen Repräsentation eines Produkts.* Das Erbauerobjekt bietet dem Direktor eine abstrakte Schnittstelle zur Konstruktion des Produkts an. Die Schnittstelle ermöglicht es dem Erbauer, die Repräsentation des Produkts, seine interne Struktur und den Konstruktionsprozeß dieser Struktur zu verstekken. Da das Produkt über eine abstrakte Schnittstelle zusammengebaut wird, müssen Sie lediglich eine neue Art von Erbauer definieren, um die interne Repräsentation eines Produkts zu ändern.

2. *Isolierung des Codes zur Konstruktion und Repräsentation.* Das Erbauermuster verbessert die Modularität eines Systems durch Kapselung des Konstruktionsprozesses und der Repräsentation eines komplexen Objekts. Klienten brauchen nichts über die Klassen zu wissen, welche die interne Struktur des Produkts definieren. Diese Klassen erscheinen nicht in der Schnittstelle eines Erbauers.

   Jeder konkrete Erbauer enthält allen Code zur Erzeugung und zur Konstruktion einer bestimmten Produktart. Der Code wird einmal geschrieben. Anschließend können unterschiedliche Direktorobjekte ihn wiederverwenden, um Produktvarianten aus derselben Menge von Komponenten zu bauen. Im anfangs angeführten RTF Beispiel können wir einen Leser für ein anderes Format als RTF definieren, so zum Beispiel einen SGML-Leser. Wir können dabei dieselben TextKonvertierer verwenden, um ASCIIText-, TeXText- und TextWidget-Darstellungen von SGML-Dokumenten zu erzeugen.

3. *Genauere Steuerung des Konstruktionsprozesses*. Im Gegensatz zu den Erzeugungsmustern, die das Produkt in einem Durchgang erzeugen, erzeugt das Erbauermuster das Produkt Schritt für Schritt unter der Steuerung des Direktors. Der Klient holt sich das Produkt vom Erbauer erst nach seiner Fertigstellung. Somit gibt die Erbauerschnittstelle den Konstruktionsprozeß des Produkts mehr als die anderen Erzeugungsmuster wieder. Dies ermöglicht Ihnen eine feinere Steuerung des Konstruktionsprozesses und somit der internen Struktur des resultierenden Produkts.

Implementierung

Üblicherweise gibt es eine abstrakte Erbauerklasse, die eine Operation für jede Komponente definiert, die der Direktor zu erzeugen verlangen könnte. Die Operationen sind per Voreinstellung leer implementiert. Eine KonkreterErbauer-Klasse überschreibt die Operationen für Komponenten, die es erzeugen können möchte.

Es folgen weitere zu bedenkende Implementierungsaspekte:

1. *Konstruktionsschnittstelle*. Erbauer konstruieren ihre Produkte schrittweise. Deswegen muß die Erbauerklassenschnittstelle allgemein genug sein, um die Konstruktion von Produkten aller möglichen konkreten Erbauer zu erlauben.

   Ein zentraler Entwurfsaspekt betrifft das Modell für den Prozeß des Zusammensammelns und Konstruierens. Meistens reicht ein Modell aus, bei dem die Ergebnisse von Konstruktionsanfragen einfach an das Produkt angehängt werden. Im Fall des RTF-Beispiels konvertiert der Erbauer das nächste Token und hängt es an den bis zu diesem Zeitpunkt konvertierten Text an.

Mitunter müssen Sie aber möglicherweise auf Teile des Produkts zugreifen, die Sie bereits zu einem früheren Zeitpunkt konstruiert haben. Im Beispielcodeabschnitt präsentieren wir für das Labyrinthbeispiel die Klasse LabyrinthErbauer, deren Schnittstelle es Ihnen ermöglicht, eine Tür zwischen zwei existierenden Räumen einzufügen. Ein anderes Beispiel sind Baumstrukturen wie zum Beispiel Parsebäume, die von unten her (bottom-up) aufgebaut werden. In einem solchen Fall übergibt der Erbauer dem Direktor die Kindobjektknoten, also die Wurzelknoten eines Teilbaums. Der Direktor gibt sie dem Erbauer zur Konstruktion der Elternobjektknoten zurück.

2. *Keine abstrakte Produktklasse.* Im allgemeinen Fall unterscheiden sich die von konkreten Erbauern erzeugten Produkte so sehr, daß man durch die Einführung einer gemeinsamen Oberklasse für die unterschiedlichen Produkte kaum etwas gewinnen kann. Im RTF-Beispiel ist es eher unwahrscheinlich, daß die ASCIIText- und TextWidget-Objekte eine gemeinsame Schnittstelle besitzen. Sie benötigen sie auch gar nicht. Da der Klient üblicherweise den Direktor mit den ihn interessierenden konkreten Erbauern konfiguriert, weiß er auch, welche konkrete Unterklasse von Erbauer gerade benutzt wird und kann seine Produkte somit entsprechend handhaben.

3. *Leere Methoden als Defaultimplementierung in der Erbaueroberklasse.* Die Baue-Operationen sind in C++ absichtlich nicht als rein virtuelle (pure virtual) Member-Funktionen deklariert. Sie sind statt dessen als leere Operationen definiert, so daß Klienten nur jene Operationen zu überschreiben brauchen, an denen sie interessiert sind.

Beispielcode

Wir werden eine Variante der `ErzeugeLabyrinth` Member-Funktion (Seite 105) definieren, die einen Erbauer der Klasse `LabyrinthErbauer` als Argument entgegennimmt.

Die Klasse `LabyrinthErbauer` definiert die folgende Schnittstelle zum Bau von Labyrinthen:

```
class LabyrinthErbauer {
public:
    virtual void BaueLabyrinth() {}
    virtual void BaueRaum(int raumNr) {}
    virtual void BaueTuer(int vonRaumNr, int nachRaumNr) {}

    virtual Labyrinth* GibLabyrinth() { return 0; }
```

```
protected:
   LabyrinthErbauer();
};
```

Über diese Schnittstelle können drei Dinge erzeugt werden: (1) das Labyrinth, (2) Räume mit einer bestimmten Raumnummer und (3) Türen zwischen numerierten Räumen. Die Operation `GibLabyrinth` gibt das Labyrinth an den Klienten zurück. Unterklassen von `LabyrinthErbauer` überschreiben diese Operation, um das von ihnen gebaute Labyrinth zurückzugeben.

Alle das Labyrinth erzeugenden Operationen von `LabyrinthErbauer` sind defaultmäßig leer implementiert. Sie sind nicht als rein virtuell deklariert, um es abgeleiteten Klassen zu ermöglichen, nur die sie interessierenden Operationen überschreiben zu müssen.

Da wir nun über die `LabyrinthErbauer`-Schnittstelle verfügen, können wir die `ErzeugeLabyrinth`-Member-Funktion ändern, so daß sie diesen Erbauer als Parameter annimmt.

```
Labyrinth* LabyrinthSpiel::ErzeugeLabyrinth(
   LabyrinthErbauer& erbauer)
{
   erbauer.BaueLabyrinth();

   erbauer.BaueRaum(1);
   erbauer.BaueRaum(2);
   erbauer.BaueTuer(1, 2);

   return erbauer.GibLabyrinth();
}
```

Vergleichen Sie diese Version von `ErzeugeLabyrinth` mit dem Original. Beachten Sie dabei, wie der Erbauer die interne Repräsentation des Labyrinths versteckt – das heißt, die Klassen, welche die Räume, Türen und Wände definieren – und wie diese Teile zusammengefügt werden, um das endgültige Labyrinth zu erstellen. Von außen kann man erkennen, daß es Klassen zur Repräsentation von Räumen und Türen gibt. Von Wänden aber fehlt jede Spur. Dies erleichtert es, die Repräsentation des Labyrinths zu ändern, da kein Klient von `LabyrinthErbauer` geändert werden muß.

Wie die anderen Erzeugungsmuster auch, kapselt das Erbauermuster die Erzeugung von Objekten. Das geschieht in diesem Fall mittels der von `LabyrinthErbauer` definierten Schnittstelle. Dies bedeutet, daß wir `LabyrinthErbauer` wiederverwen-

den können, um unterschiedliche Labyrintharten zu bauen. Die Operation ErzeugeKomplexesLabyrinth ist ein Beispiel dafür:

```
Labyrinth* LabyrinthSpiel::ErzeugeKomplexesLabyrinth(
   LabyrinthErbauer& erbauer)
{
   erbauer.BaueRaum(1);
   // ...
   erbauer.BaueRaum(1001);

   return erbauer.GibLabyrinth();
}
```

Beachten Sie, daß LabyrinthErbauer das Labyrinth nicht selbst erzeugt. Seine Hauptaufgabe besteht lediglich darin, eine Schnittstelle zum Erzeugen von Labyrinthen zu definieren. Es definiert die leeren Implementierungen von Operationen hauptsächlich aus Bequemlichkeitsgründen. Unterklassen von LabyrinthErbauer vollbringen die eigentliche Arbeit.

Die Unterklasse StandardLabyrinthErbauer stellt eine Implementierung dar, die einfache Labyrinthe zusammenbaut. Es merkt sich das im Bau befindliche Labyrinth in der Variablen _aktuellesLabyrinth.

```
class StandardLabyrinthErbauer : public LabyrinthErbauer {
public:
   StandardLabyrinthErbauer();

   virtual void BaueLabyrinth();
   virtual void BaueRaum(int raumNr);
   virtual void BaueTuer(int vonRaumNr, int nachRaumNr);

   virtual Labyrinth* GibLabyrinth();

private:
   Richtung GemeinsameWand(Raum*, Raum*);
   Labyrinth* _aktuellesLabyrinth;
};
```

GemeinsameWand ist eine Hilfsoperation, die die Richtung der gemeinsamen Wand zwischen zwei Räumen bestimmt.

Der Konstruktor von StandardLabyrinthErbauer initialisiert einfach _aktuelles-Labyrinth.

```
StandardLabyrinthErbauer::StandardLabyrinthErbauer() {
   _aktuellesLabyrinth = 0;
}
```

BaueLabyrinth erzeugt ein `Labyrinth`, das mittels weiterer Operationen zusammengefügt und am Ende über `GibLabyrinth` an den Klienten zurückgeben wird.

```
void StandardLabyrinthErbauer::BaueLabyrinth() {
   _aktuellesLabyrinth = new Labyrinth;
}

Labyrinth* StandardLabyrinthErbauer::GibLabyrinth() {
   Labyrinth* labyrinth = _aktuellesLabyrinth;
   return labyrinth;
}
```

Die Operation `BaueRaum` erzeugt einen Raum und baut die Wände um ihn herum:

```
void StandardLabyrinthErbauer::BaueRaum(int raumNr) {
   if (!_aktuellesLabyrinth->RaumNr(raumNr)) {
      Raum* raum = new Raum(raumNr);
      _aktuellesLabyrinth->FuegeRaumHinzu(raum);

      raum->SetzeSeite(Norden, new Wand);
      raum->SetzeSeite(Sueden, new Wand);
      raum->SetzeSeite(Osten, new Wand);
      raum->SetzeSeite(Westen, new Wand);
   }
}
```

Um eine Tür zwischen zwei Räumen zu bauen, sucht `StandardLabyrinthErbauer` beide Räume im Labyrinth sowie ihre gemeinsame Wand heraus:

```
void StandardLabyrinthErbauer::BaueTuer(int raumNr1,
   int raumNr2)
{
   Raum* raum1 = _aktuellesLabyrinth->RaumNr(raumNr1);
   Raum* raum2 = _aktuellesLabyrinth->RaumNr(raumNr2);
   Tuer* tuer = new Tuer(raum1, raum2);

   raum1->SetzeSeite(GemeinsameWand(raum1, raum2), tuer);
   raum2->SetzeSeite(GemeinsameWand(raum2, raum1), tuer);
}
```

Klienten können nun zur Erzeugung eines Labyrinths ErzeugeLabyrinth zusammen mit StandardLabyrinthErbauer verwenden:

```
Labyrinth* labyrinth;
LabyrinthSpiel spiel;
StandardLabyrinthErbauer erbauer;

spiel.ErzeugeLabyrinth(erbauer);
labyrinth = erbauer.GibLabyrinth();
```

Wir hätten alle Operationen von StandardLabyrinthErbauer in die Schnittstelle von Labyrinth aufnehmen und das Labyrinth sich selbst bauen lassen können. Dadurch, daß wir die Schnittstelle von Labyrinth kleiner machen, ist die Klasse aber leichter zu verstehen und zu verändern. Zudem ist StandardLabyrinthErbauer ohnehin leicht von Labyrinth zu trennen. Am wichtigsten aber ist, daß die Trennung der zwei Klassen uns die Einführung beliebiger LabyrinthErbauer-Klassen ermöglicht, die jeweils unterschiedliche Klassen für die Räume, Wände und Türen verwenden können.

Eine eher exotische Variante von LabyrinthErbauer ist ZaehlenderLabyrinthErbauer. Dieser Erbauer erzeugt überhaupt kein Labyrinth. Er ermittelt lediglich die Anzahl der unterschiedlichen Komponenten, die im Fall eines normalen Labyrinth-Erbauers erzeugt worden wären.

```
class ZaehlenderLabyrinthErbauer : public LabyrinthErbauer {
public:
    ZaehlenderLabyrinthErbauer();

    virtual void BaueLabyrinth();
    virtual void BaueRaum(int);
    virtual void BaueTuer(int, int);
    virtual void FuegeWandHinzu(int, Richtung);

    void GibAnzahl(int&, int&) const;

private:
    int _tueren;
    int _raeume;
};
```

Der Konstruktor initialisiert die Zähler, und die überschriebenen LabyrinthErbauer-Operationen erhöhen sie entsprechend.

```
ZaehlenderLabyrinthErbauer::ZaehlenderLabyrinthErbauer() {
   _raeume = _tueren = 0;
}

void ZaehlenderLabyrinthErbauer::BaueRaum(int) {
   _raeume++;
}

void ZaehlenderLabyrinthErbauer::BaueTuer(int, int) {
   _tueren++;
}

void ZaehlenderLabyrinthErbauer::GibAnzahl(int& raeume,
   int& tueren) const
{
   raeume = _raeume;
   tueren = _tueren;
}
```

Ein Klient könnte ein Exemplar von `ZaehlenderLabyrinthErbauer` folgendermaßen benutzen:

```
int raeume, tueren;
LabyrinthSpiel spiel;
ZaehlenderLabyrinthErbauer erbauer;

spiel.ErzeugeLabyrinth(erbauer);
erbauer.GibAnzahl(raeume, tueren);

cout << "Das Labyrinth verfügt über "
   << raeume << " Räume und "
   << tueren << "Türen." << endl;
```

Bekannte Verwendungen

Die RTF-Konvertierer-Anwendung stammt aus ET++ [WGM88]. Sein Direktor zur Textkonstruktion verwendet einen Erbauer, um einen im RTF-Format gespeicherten Text zu bearbeiten.

Erbauer ist ein bekanntes Muster in Smalltalk-80 [Par90]:

- Die Klasse Parser im Übersetzersubsystem ist ein Direktor, der einen ProgramNodeBuilder als Argument entgegennimmt. Ein Parser-Objekt informiert sein ProgramNodeBuilder-Objekt jedesmal, wenn es ein syntaktisches Kon-

strukt erkannt hat. Wenn der Parser fertig ist, fragt er den Erbauer nach dem aufgebauten Parsebaum und gibt ihn an den Klienten zurück.

- ClassBuilder ist ein Erbauer, den Klassen verwenden, um Unterklassen von sich selbst zu erzeugen. In diesem Fall ist Class sowohl der Direktor als auch das Produkt.
- ByteCodeStream ist ein Erbauer, der eine übersetzte Methode als Bytearray erzeugt. ByteCodeStream ist eine unübliche Anwendung des Erbauermusters, weil das komplexe von ihm erzeugte Objekt als Bytearray und nicht als normales Smalltalk-Objekt kodiert ist. Die Schnittstelle von ByteCodeStream ist allerdings typisch für einen Erbauer. Es wäre somit einfach, ByteCodeStream durch eine andere Klasse zu ersetzen, die Programme zum Beispiel als zusammengesetzte Objekte repräsentieren.

Das Service-Configurator-Framework des Adaptive-Communications-Environment verwendet einen Erbauer, um Komponenten des Netzwerkservices zu konstruieren. Die Komponenten werden zur Laufzeit in einen Server eingebunden [SS94]. Die Komponenten werden mittels einer Konfigurationssprache beschrieben, die von einem LALR(1) Parser eingelesen wird. Die semantischen Aktionen des Parser führen Operationen auf dem Erbauer aus, welche der Servicekomponente Informationen hinzufügen. In diesem Fall stellt der Parser den Direktor dar.

Verwandte Muster

Das Abstrakte-Fabrik-Muster ist dem Erbauermuster in der Hinsicht ähnlich, daß es ebenfalls komplexe Objekte konstruieren kann. Der Hauptunterschied ist, daß das Erbauermuster sich auf den schrittweisen Konstruktionsprozeß eines komplexen Objekts konzentriert. Die Betonung des Abstrakte-Fabrik-Musters liegt auf Familien von Produktobjekten (ob nun einfach oder komplex). Erbauer gibt das Produkt als letzten Schritt zurück, während das Abstrakte-Fabrik-Muster das Produkt unmittelbar zurückgibt.

Erbauer bauen oftmals Komposita (239).

Fabrikmethode

(Factory Method)

Ein klassenbasiertes Erzeugungsmuster

Zweck

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

Auch bekannt als

Virtueller Konstruktor

Motivation

Frameworks verwenden abstrakte Klassen, um die Beziehungen zwischen Objekten zu definieren und zu verwalten. Ein Framework ist oft auch für die Erzeugung dieser Objekte zuständig.

Stellen Sie sich ein Framework für Anwendungen vor, die dem Benutzer mehrere Dokumente auf einmal präsentieren können. Zwei zentrale Abstraktionen dieses Frameworks sind die Klassen Anwendung und Dokument. Beide Klassen sind abstrakt, und Klienten müssen Unterklassen von ihnen bilden, um ihre anwendungsspezifischen Implementierungen einzubringen. Um beispielsweise eine Zeichenanwendung zu erstellen, definieren wir die Klassen ZeichenAnwendung und ZeichenDokument. Die Klasse Anwendung ist für die Verwaltung von Dokumenten zuständig und erzeugt sie auf Verlangen – beispielsweise wenn der Benutzer Öffnen oder Neu in einem Menü auswählt.

Da die jeweilige Dokumentunterklasse, von der Objekte zu erzeugen sind, anwendungsspezifisch ist, kann Anwendung diese Unterklasse nicht vorhersagen – sie weiß lediglich, *wann* ein neues Dokument erzeugt werden soll, nicht aber *welche Art* von Dokument zu erzeugen ist. Dies führt zu einem Dilemma: Das Framework muß Objekte erzeugen, kennt aber nur ihre abstrakten Oberklassen, von denen es keine Objekte erzeugen kann.

Das Fabrikmethodemuster bietet eine Lösung. Es kapselt das Wissen um die zu erzeugende Dokument-Unterklasse und lagert es aus dem Framework aus (siehe Abbildung 3.7).

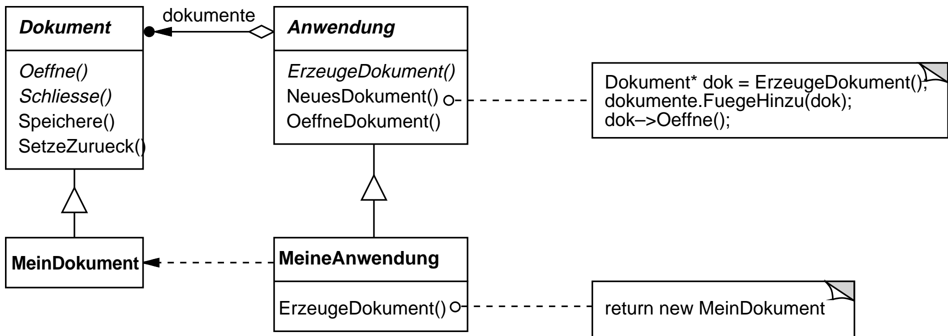

Abbildung 3.7

Anwendungs-Unterklassen überschreiben eine abstrakte ErzeugeDokument-Operation von Anwendung, so daß sie ein Exemplar der passenden Dokument-Unterklasse zurückgibt. Sobald einmal ein Objekt einer Unterklasse von Anwendung erzeugt ist, kann sie anwendungsspezifische Dokumente erzeugen, ohne deren exakte Klasse zu kennen. Wir nennen ErzeugeDokument eine *Fabrikmethode*, weil sie für die »Herstellung« eines Objekts zuständig ist.

Anwendbarkeit

Verwenden Sie das Fabrikmethodemuster, wenn

- eine Klasse die Klassen von Objekten, die sie erzeugen muß, nicht im voraus kennen kann.
- eine Klasse möchte, daß ihre Unterklassen die von ihr zu erzeugenden Objekte festlegen.
- Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen und Sie das Wissen lokalisieren wollen, an welche Hilfsunterklasse die Zuständigkeiten delegiert werden.

Struktur

Abbildung 3.8 zeigt die Musterstruktur.

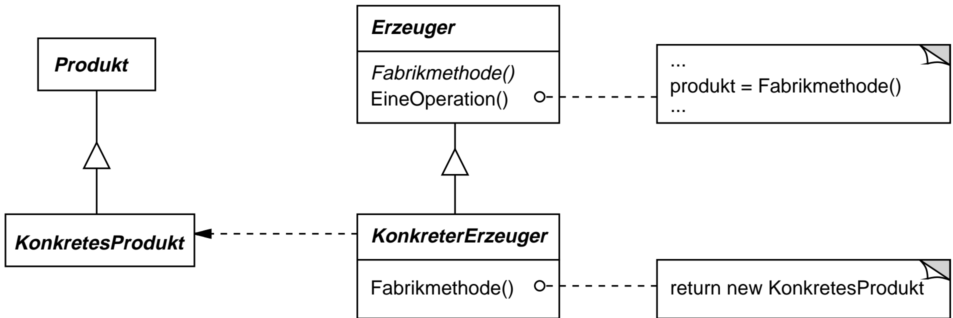

Abbildung 3.8

Teilnehmer

- **Produkt** (Dokument)
  - definiert die Klasse des von der Fabrikmethode erzeugten Objekts.
- **KonkretesProdukt** (MeinDokument)
  - implementiert die Produktschnittstelle.
- **Erzeuger** (Anwendung)
  - deklariert die Fabrikmethode, die ein Objekt des Typs Produkt zurückgibt. Der Erzeuger kann möglicherweise eine Defaultimplementierung der Fabrikmethode definieren, die ein vordefiniertes KonkretesProduktObjekt erzeugt.
  - kann die Fabrikmethode aufrufen, um ein Produktobjekt zu erzeugen.
- **KonkreterErzeuger** (MeineAnwendung)
  - überschreibt die Fabrikmethode, so daß sie ein Exemplar von KonkretesProdukt zurückgibt.

Interaktionen

- Der Erzeuger verläßt sich darauf, daß Unterklassen die Fabrikmethode definieren, so daß sie ein Exemplar der passenden konkreten Produktklasse zurückgeben.

Konsequenzen

Fabrikmethoden verhindern es, daß Sie anwendungsspezifische Klassen in Frameworkcode einbinden müssen. Der Code befaßt sich nur mit der Produktschnittstelle; er kann somit mit jeder benutzerdefinierten KonkretesProdukt-Klasse arbeiten.

Ein möglicher Nachteil von Fabrikmethoden ist, daß Klienten potentiell die Erzeugerklasse ableiten müssen, nur um ein bestimmtes KonkretesObjekt-Exemplar erzeugen zu können. Die Bildung von Unterklassen ist unproblematisch, wenn der Klient die Erzeugerklasse sowieso ableiten muß. Ist dies nicht der Fall, so muß er ausschließlich der Fabrikmethode wegen mit einem weiteren Evolutionsast seiner Software zurechtkommen.

Es folgen zwei weitere Konsequenzen des Fabrikmethodemusters:

1. *Spezialisierungmöglichkeiten für Unterklassen.* Die Erzeugung von Objekten innerhalb einer Klasse mittels einer Fabrikmethode ist immer flexibler als das direkte Erzeugen eines Objekts. Die Fabrikmethode bietet Unterklassen die Möglichkeit, eine erweiterte Version eines Objekts einzuführen.

   Im Dokumentbeispiel könnte die Dokument-Klasse eine Fabrikmethode namens ErzeugeDateiDialog definieren, die einen vordefinierten Dateidialog zum Öffnen eines existierenden Dokuments erzeugt. Eine Dokument-Unterklasse kann einen anwendungsspezifischen Dateidialog durch Überschreiben dieser Fabrikmethode definieren. In diesem Fall ist die Fabrikmethode nicht abstrakt, sondern bietet eine sinnvolle Defaultimplementierung.

2. *Verbindung paralleler Klassenhierarchien.* In den bisher betrachteten Beispielen wird die Fabrikmethode nur von Erzeugern aufgerufen. Dies muß aber nicht immer so sein. Klienten können Fabrikmethoden ebenfalls als sinnvoll erachten, insbesondere im Fall von parallelen Klassenhierarchien.

   Parallele Klassenhierarchien ergeben sich, wenn eine Klasse Teile seiner Zuständigkeiten an eine abgetrennte Klasse delegiert. Stellen Sie sich grafische Objekte vor, die interaktiv manipuliert werden können; das heißt, sie können mittels Maus gestreckt, bewegt oder rotiert werden. Die Implementierung dieser Interaktionen ist nicht immer einfach. Man muß oftmals Informationen speichern und aktualisieren, die den Zustand der Manipulation zu einem bestimmten Zeitpunkt festhalten. Dieser Zustand wird lediglich während der Manipulation gebraucht; somit braucht er nicht im grafischen Objekt aufbewahrt zu werden. Weiterhin verhalten sich verschiedene grafische Objekte unterschiedlich, wenn der Benutzer sie manipuliert. Beispielsweise hat das Strecken

einer Linie den Effekt des Bewegens eines Endpunkts, während das Strecken eines Textobjekts möglicherweise zur Änderung seines Zeilenabstands führt.

Unter diesen Bedingungen ist es sinnvoller, ein abgetrenntes Manipulator-Objekt zu verwenden, das die Interaktion implementiert und jeglichen benötigten manipulationsspezifischen Zustand verwaltet. Unterschiedliche grafische Objekte verwenden unterschiedliche Manipulator-Unterklassen, um bestimmte Interaktionsmöglichkeiten zu bieten. Die resultierende Manipulator-Klassenhierarchie verläuft zumindest teilweise parallel zur Klassenhierarchie der grafischen Objekte (siehe Abbildung 3.9).

Die GrafischesObjekt-Klasse bietet eine ErzeugeManipulator-Fabrikmethode, die es Klienten ermöglicht, ein zum grafischen Objekt passendes Manipulator-Objekt zu erzeugen.

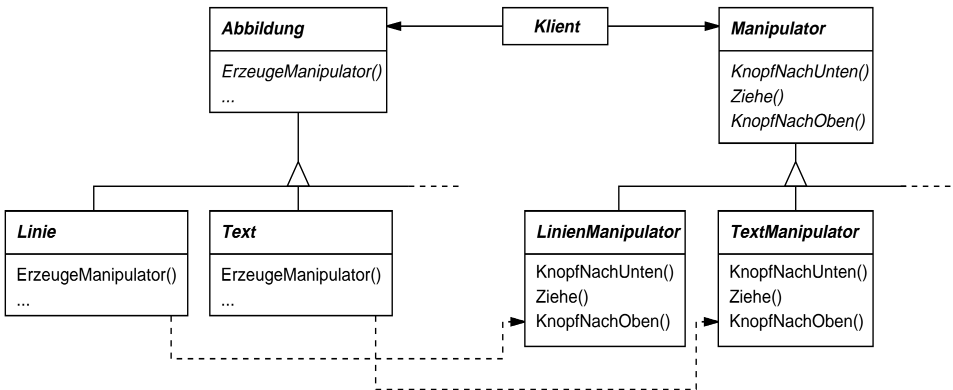

Abbildung 3.9

Die GrafischesObjekt-Unterklassen überschreiben diese Methode, so daß sie ein Exemplar der für sie richtigen Manipulator-Unterklasse zurückgeben. Alternativ kann die GrafischesObjekt-Klasse ErzeugeManipulator so implementieren, daß sie ein Objekt einer vordefinierten Manipulator-Klasse zurückgibt. Die Unterklassen können einfach diese Voreinstellung erben. Jene GrafischesObjekt-Klassen, die dies tun, benötigen keine angepaßte Manipulator-Unterklasse – somit sind die Klassenhierarchien nur teilweise parallel strukturiert.

Beachten Sie, wie die Fabrikmethode die Verbindung zwischen den zwei Klassenhierarchien definiert. Sie lokalisiert das Wissen, welche Klassen zueinander gehören.

Implementierung

Ziehen Sie die folgenden Aspekte bei der Anwendung des Fabrikmethodemusters in Betracht:

1. *Zwei größere Variationen.* Die zwei wichtigsten Variationen des Fabrikmethodemusters sind (1) der Fall, wenn die Erzeugerklasse abstrakt ist und keine Implementierung der von ihr deklarierten Fabrikmethode bietet, und (2) der Fall, wenn die Erzeugerklasse konkret ist und eine Defaultimplementierung für die Fabrikmethode bietet. Es ist auch möglich, wenngleich eher selten, über eine abstrakte Klasse zu verfügen, die eine Defaultimplementierung bietet.

   Der erste Fall *verlangt*, daß Unterklassen eine Implementierung definieren, weil es keine sinnvolle Voreinstellung gibt. Es umgeht das Dilemma, Objekte nicht vorhersehbare Klassen erzeugen zu müssen. Im zweiten Fall benutzt der konkrete Erzeuger die Fabrikmethode hauptsächlich aus Flexibilitätsgründen. Er folgt einer Regel, die besagt, daß man Objekte in einer separaten Operation erzeugen soll, so daß Unterklassen den Erzeugungscode überschreiben können. Diese Regel stellt sicher, daß Entwickler von Unterklassen, falls notwendig, die Klassen der von ihren Oberklassen erzeugten Objekte ändern können.

2. *Parametrierbare Fabrikmethoden.* Eine weitere Variation des Musters ermöglicht es der Fabrikmethode, mehrere Arten von Produkten zu erzeugen. Die Fabrikmethode erhält einen Parameter, der die Art des zu erzeugenden Objekts bestimmt. Alle von der Fabrikmethode erzeugten Objekte teilen die Produktschnittstelle. Im Dokumentbeispiel unterstützt die Anwendung möglicherweise unterschiedliche Arten von Dokumenten. Sie übergeben dann ErzeugeDokument einen zusätzlichen Parameter, um die Art des zu erzeugenden Dokuments festzulegen.

   Das Unidraw-Framework für grafische Editoren [VL90] verwendet diesen Ansatz zur Rekonstruktion von auf der Festplatte gespeicherten Objekten. Unidraw definiert eine Klasse `Creator` mit einer Fabrikmethode `Create`, die einen Klassenidentifizierer als Argument annimmt. Der Klassenidentifizierer spezifiziert die Klasse, von der Objekte zu erzeugen sind. Wenn Unidraw ein Objekt auf der Festplatte abspeichert, schreibt es zuerst den Klassenidentifizierer, gefolgt von den Exemplarvariablen. Wenn es das Objekt von der Festplatte rekonstruiert, liest es zuerst den Identifizierer.

   Ist der Klassenidentifizierer einmal gelesen, ruft das Framework `Create` auf und übergibt dabei den Identifizierer als Parameter. `Create` sucht den Konstruktor der entsprechenden Klasse heraus und benutzt ihn, um das Objekt zu erzeugen. Zum Schluß ruft `Erzeuge` die `Read`-Operation des Objekts auf, welche die

auf der Festplatte verbliebenen Daten einliest und die Exemplarvariablen des Objekts mit ihnen initialisiert.

Eine parametrierbare Fabrikmethode besitzt die folgende allgemeine Form, wobei `MeinProdukt` und `DeinProdukt` Unterklassen von `Produkt` sind:

```
class Erzeuger {
public:
   virtual Produkt* Erzeuge(ProduktId);
};

Produkt* Erzeuger::Erzeuge(ProduktId id) {
   if (id == MEINS) return new MeinProdukt;
   if (id == DEINS) return new DeinProdukt;
   // Wiederholung für verbleibende Produkte...

   return 0;
}
```

Das Überschreiben einer parametrierbaren Fabrikmethode ermöglicht es Ihnen, die von einem Erzeuger produzierten Produkte einfach und gezielt zu erweitern oder zu ändern. Sie können neue Identifizierer für neue Arten von Produkten einführen, oder Sie können existierende Identifizierer an andere Produkte binden.

Beispielsweise könnte eine Unterklasse von `MeinErzeuger` die Unterklassen `MeinProdukt` und `DeinProdukt` austauschen und eine neue Unterklasse `IhrProdukt` unterstützen:

```
Produkt* MeinErzeuger::Erzeuge(ProduktId id) {
   if (id == DEINS) return new MeinProdukt;
   if (id == MEINS) return new DeinProdukt;
   // Die Produkte wurden vertauscht

   if (id == IHRS) return new IhrProdukt;

   return Erzeuger::Erzeuge(id);
   // wird aufgerufen, wenn alles andere fehlschlägt
}
```

Beachten Sie, daß der letzte Arbeitsschritt dieser Operation im Aufruf von `Erzeuge` der Oberklasse besteht. Dies liegt daran, daß `MeinErzeuger::Erzeuge` nur `MEINS`, `DEINS` und `IHRS` anders als die Oberklasse behandelt. Andere Klassen interessieren es nicht. Somit *erweitert* `MeinErzeuger` die Arten erzeugbarer Produk-

te. Es gibt die Zuständigkeit zum Erzeugen aller Produktarten bis auf einige wenige an ihre Oberklasse weiter.

3. *Sprachspezifische Varianten und Aspekte.* Unterschiedliche Sprachen führen von allein zu weiteren interessanten Variationen und Fragestellungen.

   Smalltalk-Programme verwenden oft eine Methode, die die Klasse des zu erzeugenden Objekts zurückgibt. Eine Fabrikmethode des Erzeugers kann diesen Rückgabewert dazu verwenden, ein Produkt zu erzeugen, und ein KonkreterErzeuger kann diese Klasse speichern oder sogar berechnen. Das Ergebnis ist eine noch einmal spätere Ermittlung des Typs von KonkretesProdukt, von dem ein Objekt erzeugt werden soll.

   Eine Smalltalk-Version des Dokumentbeispiels definiert möglicherweise eine Methode `dokumentKlasse` der Klasse `Anwendung`. Die Methode `dokumentKlasse` gibt die korrekte Dokument-Klasse zum Erzeugen von Dokumenten zurück. Die Implementierung von `dokumentKlasse` in `MeineAnwendung` gibt die Klasse `MeinDokument` zurück. Somit ergibt sich für die Klasse `Anwendung`:

```
klientenMethode
    dokument := self dokumentKlasse new

dokumentKlasse
    self subclassResponsibility
```

   In der Klasse MeineAnwendung definieren wir:

```
dokumentKlasse
    ^ MeinDokument
```

   Diese Methode gibt die Klasse `MeinDokument` zurück, von der `Anwendung` Exemplare erzeugt.

   Ein noch flexiblerer Ansatz, vergleichbar parametrierbaren Fabrikmethoden, besteht darin, die zu Klasse zu erzeugender Objekte als eine Klassenvariable von `Anwendung` zu speichern. Man muß dann keine Unterklasse von `Anwendung` bilden, um das Produkt zu variieren.

   In C++ sind Fabrikmethoden immer virtuelle Funktionen, die meistens sogar als rein virtuell deklariert werden. Sie müssen hierbei allerdings aufpassen, die Fabrikmethoden nicht aus dem Konstruktor des Erzeugers heraus aufzurufen – die Fabrikmethoden der Unterklasse KonkreterErzeuger sind zu diesem Zeitpunkt noch nicht verfügbar.

   Sie können dies vermeiden, indem Sie vorsichtigerweise auf Produkte ausschließlich durch Zugriffsoperationen zugreifen, die das Objekt auf Verlangen

erzeugen. Statt das konkrete Produkt im Konstruktor zu erzeugen, initialisiert der Konstruktor es lediglich zu 0. Die Zugriffsoperation gibt das Objekt zurück, testet allerdings vorher, ob das Produkt existiert. Falls dies nicht der Fall ist, erzeugt es das Produkt erst einmal. Diese Technik wird mitunter als **verzögerte Initialisierung** (lazy initialization) bezeichnet. Der folgende Code zeigt eine typische Implementierung:

```
class Erzeuger {
public:
                Produkt* GibProdukt();

protected:
            virtual Produkt* ErzeugeProdukt();

private:
                Produkt* _produkt;
};

Produkt* Erzeuger::GibProdukt() {
   if (_produkt == 0) {
      _produkt = ErzeugeProdukt();
   }
   return _produkt;
}
```

4. *Verwendung von Templates zur Vermeidung von Unterklassen.* Ein weiteres potentielles Problem von Fabrikmethoden besteht darin, daß Sie gezwungen sein könnten, eine Unterklasse lediglich zur Erzeugung der passenden Produktobjekte zu erstellen. In C++ bietet sich eine weitere Möglichkeit, dieses Problem zu umgehen. Dabei führt man eine templatebasierte Unterklasse von Erzeuger ein, die mit der Produktklasse parametrierbar wird:

```
class Erzeuger {
public:
   virtual Produkt* ErzeugeProdukt() = 0;
};

template<class DasProdukt>
class StandardErzeuger : public Erzeuger {
public:
   virtual Produkt* ErzeugeProdukt();
};

template<class DasProdukt>
Produkt* StandardErzeuger<DasProdukt>::ErzeugeProdukt() {
```

```
        return new DasProdukt;
    }
```

Unter Verwendung dieses Templates gibt der Klient nur noch die Produktklasse an – es muß keine Unterklasse von Erzeuger erstellt werden.

```
class MeinProdukt : public Produkt {
public:
    MeinProdukt();
    // ...
};

StandardErzeuger<MeinProdukt> meinProdukt;
```

5. *Namenskonventionen.* Es hat sich bewährt, Namenskonventionen zu verwenden, die klarstellen, daß Sie Fabrikmethoden verwenden. Beispielsweise deklariert das Mac-App-Application-Framework für den Apple Macintosh [App89] die eine Fabrikmethode definierende abstrakte Operation immer als `Klasse* DoMakeKlasse()`, wobei `Klasse` die Produktklasse darstellt.

Beispielcode

Die Funktion `ErzeugeLabyrinth` (Seite 90) erzeugt ein Labyrinth und gibt es zurück. Ein Nachteil dieser Funktion ist, daß sie die Klassen des Labyrinths, der Räume, Türen und Wände fest codiert. Wir führen Fabrikmethoden ein, um Unterklassen die Komponenten auswählen zu lassen.

Zuerst definieren wir Fabrikmethoden in `LabyrinthSpiel`, um Labyrinth-, Raum-, Wand- und Türobjekte zu erzeugen:

```
class LabyrinthSpiel {
public:
    Labyrinth* ErzeugeLabyrinth();

    // Die Fabrikmethoden:
    virtual Labyrinth* ErzeugeLabyrinth() const
        { return new Labyrinth; }
    virtual Raum* ErzeugeRaum(int raumNr) const
        { return new Raum(raumNr); }
    virtual Wand* ErzeugeWand() const
        { return new Wand; }
    virtual Tuer* ErzeugeTuer(Raum* raum1, Raum* raum2) const
        { return new Tuer(raum1, raum2); }
};
```

Jede Fabrikmethode gibt eine Labyrinthkomponente eines gegebenen Typs zurück. `LabyrinthSpiel` bietet Defaultimplementierungen, welche die einfachsten Arten von Labyrinthen, Räumen, Wänden und Türen zurückgeben.

Wir können nun `ErzeugeLabyrinth` so umschreiben, daß es diese Fabrikmethoden verwendet:

```
Labyrinth* LabyrinthSpiel::ErzeugeLabyrinth() {
   Labyrinth* einLabyrinth = ErzeugeLabyrinth();

   Raum* raum1 = ErzeugeRaum(1);
   Raum* raum2 = ErzeugeRaum(2);
   Tuer* dieTuer = ErzeugeTuer(raum1, raum2);

   einLabyrinth->FuegeRaumHinzu(raum1);
   einLabyrinth->FuegeRaumHinzu(raum2);

   raum1->SetzeSeite(Norden, ErzeugeWand());
   raum1->SetzeSeite(Osten, dieTuer);
   raum1->SetzeSeite(Sueden, ErzeugeWand());
   raum1->SetzeSeite(Westen, ErzeugeWand());

   raum2->SetzeSeite(Norden, ErzeugeWand());
   raum2->SetzeSeite(Osten, ErzeugeWand());
   raum2->SetzeSeite(Sueden, ErzeugeWand());
   raum2->SetzeSeite(Westen, dieTuer);

   return einLabyrinth;
}
```

Unterschiedliche Spiele können Unterklassen von `LabyrinthSpiel` bilden, um Teile des Labyrinths zu spezialisieren. Diese Unterklassen können einige oder alle Fabrikmethoden neu definieren, um die Produkte zu variieren. Beispielsweise kann ein `LabyrinthMitBombenSpiel` die Raum- und Wandprodukte neu definieren, um so die bombardierbaren Versionen zurückzugeben:

```
class LabyrinthMitBombenSpiel : public LabyrinthSpiel {
public:
   LabyrinthMitBombenSpiel();

   virtual Wand* ErzeugeWand() const
      { return new BombardierbareWand; }

   virtual Raum* ErzeugeRaum(int raumNr) const
      { return new RaumMitBombe(raumNr); }
};
```

Ein Variante VerzaubertesLabyrinthSpiel kann so definiert werden:

```
class VerzaubertesLabyrinthSpiel : public LabyrinthSpiel {
public:
   VerzaubertesLabyrinthSpiel();

   virtual Raum* ErzeugeRaum(int raumNr) const
      { return new VerzauberterRaum(raumNr,
          BenoetigterZauberspruch()); }
   virtual Tuer* ErzeugeTuer(Raum* raum1, Raum* raum2) const
      { return new TuerMitZauberspruch(raum1, raum2); }

protected:
   Zauberspruch* BenoetigterZauberspruch() const;
};
```

Bekannte Verwendungen

Fabrikmethoden werden in Klassenbibliotheken und Frameworks durchgängig eingesetzt. Das einführende Dokumentbeispiel ist ein typischer Anwendungsfall in MacApp und ET++ [WGM88]. Das Manipulator-Beispiel stammt aus Unidraw.

Die Klasse View im Smalltalk-80 MVC-Framework besitzt eine Methode default-Controller, die ein Controller-Objekt erzeugt, so daß diese Methode wie eine Fabrikmethode aussieht [Par90]. Unterklassen von View spezifizieren allerdings die Klasse ihres Default-Controllers durch die Definition der Methode defaultControllerClass, welche die Klasse zurückgibt, von der defaultController Exemplare erzeugt. Somit ist defaultControllerClass die eigentliche Fabrikmethode, das heißt jene Methode, die Unterklassen überschreiben sollten.

Ein eher abgehobenes Beispiel in Smalltalk-80 ist die Fabrikmethode parserClass, die von Behavior definiert wird (Behavior ist die Oberklasse aller Objekte, die Klassen repräsentieren). Dies ermöglicht es einer Klasse, einen maßgeschneiderten Parser für ihren Quelltext zu verwenden. Beispielsweise kann ein Klient eine Klasse SQLParser definieren, um den Quelltext einer Klasse mit eingebetteten SQL-Befehlen zu analysieren. Die Klasse Behavior implementiert parserClass so, daß es die standardmäßige Smalltalk-Parserklasse zurückgibt. Eine Klasse, die SQL-Befehle einbetten kann, überschreibt diese Methode (als eine Klassenmethode) und gibt die SQLParser-Klasse zurück.

Das Orbix ORB-System von IONA Technologies [ION94] benutzt Fabrikmethoden, um ein Proxyobjekt (siehe Proxy (254)) des passenden Typs zu generieren, wann immer ein Objekt eine Referenz auf ein Objekt in einem anderen Prozeß

verlangt. Fabrikmethode macht es einfach, das Defaultproxy durch ein anderes Proxy zu ersetzen, das zum Beispiel Caching auf der Klientenseite verwendet.

Verwandte Muster

Das Abstrakte-Fabrik-Muster wird oft mittels Fabrikmethoden implementiert. Das Beispiel aus dem Motivationsabschnitt des Abstrakte-Fabrik-Muster beschreibt ebenfalls das Fabrikmethodemuster.

Fabrikmethoden werden üblicherweise innerhalb von Schablonenmethoden (366) aufgerufen. Im obigen Dokumentbeispiel stellt NeuesDokument eine Schablonenmethode dar.

Prototypen (144) benötigen keine Unterklasse von Erzeuger. Sie verlangen allerdings oftmals eine Initialisiere-Operation der Produktklasse. Erzeuger verwendet Initialisiere zur Initialisierung des Objekts. Fabrikmethoden benötigen keine solche Operation.

Prototyp

(Prototype)

Ein objektbasiertes Erzeugungsmuster

Zweck

Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen.

Motivation

Sie können einen Editor für Musikpartituren erstellen, indem Sie ein allgemeines Framework für grafische Editoren anpassen und neue Objekte hinzufügen, welche die Noten, Pausen und Notenlinien repräsentieren. Das Editor-Framework besitzt vielleicht eine Palette von kleinen Werkzeugen, um Musikobjekte der Partitur hinzuzufügen. Die Palette dürfte weiterhin Werkzeuge zum Auswählen, Bewegen sowie für weitere Manipulationsmöglichkeiten der Musikobjekte besitzen. Die Benutzer klicken auf das Werkzeug für Viertelnoten und benutzen es, um die Viertelnoten der Partitur hinzuzufügen. Oder Sie verwenden das Bewegungswerkzeug, um eine Note auf den Notenlinien auf oder ab zu bewegen, wobei Sie seine Tonhöhe verändern.

Nehmen wir an, daß das Framework eine abstrakte Klasse GrafischesObjekt für grafische Komponenten wie die Noten und Notenlinien bietet. Weiterhin bietet es eine abstrakte Klasse Werkzeug zur Definition von Werkzeugen wie denen in der Palette. Das Framework definiert weiterhin eine Unterklasse GrafischesWerkzeug für Werkzeuge, die grafische Objekte erzeugen und dem Dokument hinzufügen können.

Die Klasse GrafischesWerkzeug stellt ein Problem für den Frameworkentwickler dar. Die Klassen für Noten und Notenlinien sind anwendungsspezifisch, die Klasse GrafischesWerkzeug gehört aber zum Framework. Die Klasse GrafischesWerkzeug weiß nicht, wie die Exemplare unserer Musikklassen zu erzeugen sind, die der Partitur hinzugefügt werden sollen. Wir könnten für jede Art von Musikobjekt eine Unterklasse von GrafischesWerkzeug bilden, was aber zu vielen Unterklassen führen würde, die sich nur in der Art des zu erzeugenden Musikobjekts unterscheiden. Wir wissen, daß Objektkomposition eine flexible Alternative zur Unterklassenbildung ist. Die Frage ist, wie das Framework Objektkomposition ver-

wenden kann, um Exemplare von GrafischesWerkzeug mit der Klasse der zu erzeugenden grafischen Objekte zu parametrieren.

Die Lösung besteht darin, GrafischesWerkzeug ein neues grafisches Objekt mittels Kopieren oder »Klonen« eines Exemplars einer GrafischesObjekt-Unterklasse erzeugen zu lassen (siehe Abbildung 3.10). Wir nennen dieses Exemplar *Prototyp*. GrafischesWerkzeug wird mit dem Prototyp parametrisiert, den es klonen und dem Dokument hinzufügen soll. Wenn alle Unterklassen von GrafischesObjekt eine Klone-Operation anbieten, kann GrafischesWerkzeug jede Art von GrafischesObjekt klonen.

Somit ist in unserem Musikeditor jedes Werkzeug zum Erzeugen eines Musikobjekts ein Exemplar von GrafischesWerkzeug, das mit einem anderen Prototypen initialisiert wird. Jedes GrafischesWerkzeug-Exemplar produziert ein Musikobjekt, indem es seinen Prototypen klont und das geklonte Objekt der Partitur hinzufügt.

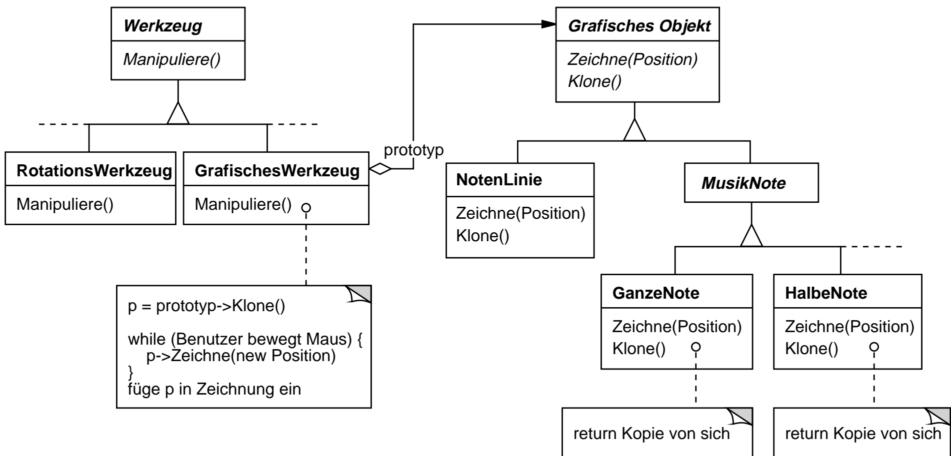

Abbildung 3.10

Wir können das Prototypmuster sogar dazu verwenden, die Anzahl der Klassen noch weiter zu senken. Wir verfügen über Klassen für ganze und für halbe Noten. Dies ist möglicherweise unnötig. Statt dessen könnten sie Exemplare derselben Klasse sein, die mit unterschiedlichen Bitmaps und unterschiedlicher Tondauer initialisiert werden. Ein Werkzeug zum Erzeugen ganzer Noten wird somit zu einem GrafischesWerkzeug-Objekt, dessen Prototyp eine MusikNote ist, die so initialisiert wurde, daß sie eine ganze Note darstellt. Diese kann zu einer drastischen Reduzierung der Klassenanzahl im System führen. Das Hinzufügen einer neuen Art von Note zum Musikeditor wird ebenfalls einfacher.

Anwendbarkeit

Verwenden Sie das Prototypmuster, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, *und*

- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, beispielsweise durch dynamisches Laden, *oder*
- um zu vermeiden, eine Klassenhierarchie von Fabriken zu erstellen, die parallel zur Klassenhierarchie der Produkte verläuft, *oder*
- wenn Exemplare einer Klasse nur wenige unterschiedliche Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonen statt die Objekte einer Klasse jedesmal von Hand mit dem richtigen Zustand zu erzeugen.

Struktur

Abbildung 3.11 zeigt die Struktur des Prototypmusters.

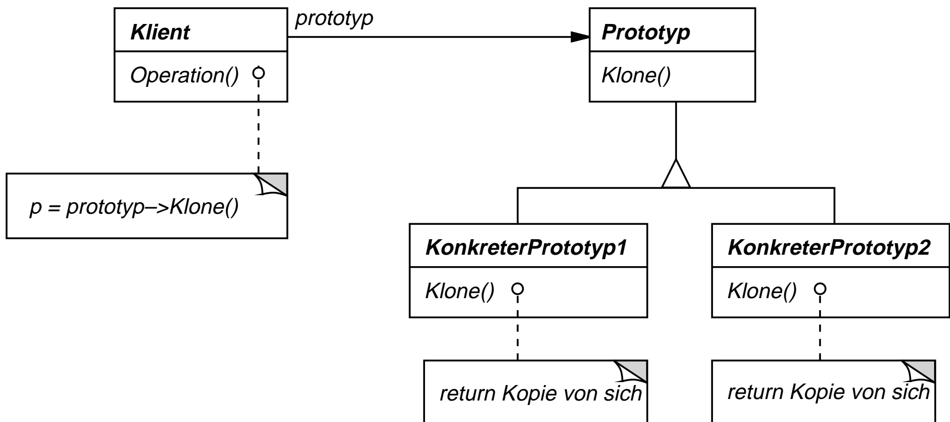

Abbildung 3.11

Teilnehmer

- **Prototyp** (GrafischesObjekt)
  - deklariert eine Schnittstelle, um sich selbst zu klonen.
- **KonkretesProdukt** (NotenLinie, GanzeNote, HalbeNote)
  - implementiert eine Operation, um sich selbst zu klonen.

- **Klient** (GrafischesWerkzeug)
  - erzeugt ein neues Objekt, indem es einem Prototyp befiehlt, sich selbst zu klonen.

Interaktionen

- Ein Klient befiehlt einem Prototyp, sich selbst zu klonen.

Konsequenzen

Das Prototypmuster hat viele derselben Konsequenzen, welche das Abstrakte-Fabrik-Muster (107) und das Erbauermuster (119) haben: Es versteckt die konkreten Produktklassen vor dem Klienten und reduziert dadurch die Anzahl der dem Klienten bekannten Namen. Weiterhin ermöglichen diese Muster es einem Klienten, ohne Modifikation mit anwendungsspezifischen Klassen zu arbeiten.

Zusätzliche Möglichkeiten des Prototypmusters sind im Folgenden aufgeführt.

1. *Hinzufügen und Entfernen von Produkten zur Laufzeit.* Prototypen ermöglichen es Ihnen, eine neue Produktklasse in ein System einfach dadurch einzubinden, daß sie ein prototypisches Exemplar beim Klienten registrieren. Dies ist etwas flexibler als die anderen Erzeugungsmuster, weil ein Klient Prototypen zur Laufzeit installieren und entfernen kann.

2. *Spezifikation neuer Objekte durch Variation von Werten.* Hochdynamische Systeme ermöglichen es Ihnen, neues Verhalten durch die Objektkomposition zu definieren – indem Sie beispielsweise Werte für die Variablen eines Objektes spezifizieren – und nicht, indem Sie neue Klassen definieren. Sie definieren effektiv neue Arten von Objekten durch das Erzeugen von Objekten existierender Klassen und die Registrierung der Objekte als Prototypen für Klientenobjekte. Ein Klient kann neues Verhalten durch die Delegation von Zuständigkeiten an den Prototypen ausüben.

   Diese Art von Entwurf ermöglicht es Benutzern, neue »Klassen« ohne Programmierung zu definieren. Tatsächlich gleicht das Klonen eines Prototypen dem Erzeugen eines Objekts einer Klasse. Das Prototypmuster ist in der Lage, die von einem System benötigte Anzahl an Klassen deutlich zu reduzieren. In unserem Musikeditor kann eine einzige GrafischesWerkzeug-Klasse eine unbegrenzte Vielfalt von Musikobjekten erzeugen.

3. *Spezifikation neuer Objekte durch Variation der Struktur.* Viele Anwendungen konstruieren Objekte aus Teilen und Subteilen. Editoren für den Entwurf elektri-

scher Schaltkreise konstruieren beispielsweise Schaltkreise mit Hilfe von Teilschaltkreisen.[1] Aus Bequemlichkeitsgründen ermöglichen diese Anwendungen es Ihnen oft, komplexe, benutzerdefinierte Strukturen zu erzeugen, etwa um einen bestimmten Teilschaltkreis immer wieder zu verwenden.

Das Prototypmuster unterstützt dies ebenfalls. Wir fügen diesen Teilschaltkreis einfach als einen Prototyp zur Palette verfügbarer Schaltkreiselemente hinzu. Solange das den zusammengesetzten Schaltkreis repräsentierende Objekt das Klonen als tiefes Kopieren implementiert, können Schaltkreise mit unterschiedlichen Strukturen als Prototypen verwendet werden.

4. *Verringerte Unterklassenbildung.* Das Fabrikmethodemuster (131) produziert oft eine Hierarchie von Erzeugerklassen, die parallel zur Produktklassenhierarchie verläuft. Das Prototypmuster ermöglicht es Ihnen, zur Erzeugung eines neuen Objekts einen Prototypen zu klonen, statt eine Fabrikmethode aufzurufen. Somit benötigen Sie überhaupt keine Erzeugerklassenhierarchie. Dieser Vorteil kommt hauptsächlich in Sprachen wie C++ zum Tragen, die Klassen nicht als Objekte erster Ordnung behandeln. Sprachen wie Smalltalk oder Objective-C, die dies tun, ziehen hieraus einen geringeren Vorteil, da Sie immer ein Klassenobjekt als Erzeuger verwenden können. Klassenobjekte spielen in diesen Sprachen bereits die Rolle von Prototypen.

5. *Dynamisches Konfigurieren einer Anwendung mit Klassen.* Manche Laufzeitumgebungen ermöglichen es Ihnen, Klassen dynamisch zu einer Anwendung hinzuzuladen. Das Prototypmuster ist der Schlüssel zum Ausbeuten solcher Möglichkeiten in einer Sprache wie C++.

   Eine Anwendung, die Exemplare einer dynamisch geladenen Klasse erzeugen möchte, wird nicht in der Lage sein, den Konstruktur dieser Klassen statisch zu referenzieren. Statt dessen erzeugt die Laufzeitumgebung beim Laden jeder Klasse automatisch ein Exemplar und registriert dieses Exemplar bei einem Prototypenverwalter (siehe Implementierungsabschnitt). Die Anwendung kann dann den Prototypenverwalter nach Exemplaren der gerade neu geladenen Klassen fragen, Klassen die ursprünglich gar nicht in das Programm eingebunden waren. Das ET++-Application-Framework [WGM88] verfügt über ein Laufzeitsystem, das diese Methode verwendet.

Die Hauptverpflichtung des Prototypmusters besteht darin, daß jede Unterklasse von Prototyp die Operation Klone implementieren muß. Dies kann eine schwie-

1. Solche Anwendungen verwenden auch das Kompositionsmuster (239) und das Dekorierermuster (199).

rige Aufgabe sein. Beispielsweise ist das Hinzufügen von Klone schwierig, wenn die in Betracht gezogenen Klassen bereits existieren. Die Implementierung von Klone kann schwierig sein, wenn ihre interne Repräsentation Objekte umfaßt, die keine Kopiermöglichkeiten bieten oder über zirkuläre Referenzen verfügen.

Implementierung

Das Prototypmuster ist besonders nützlich in statisch typisierten Programmiersprachen wie C++, in denen Klassen keine Objekte sind und nur wenig oder gar keine Typinformation zur Laufzeit verfügbar ist. In Sprachen wie Smalltalk oder Objective-C, die Prototypen vergleichbarer Objekte zur Erzeugung von Exemplaren einer Klasse bieten, nämlich Klassenobjekte, ist es weniger wichtig. Dieses Muster ist in prototyp-basierten Sprachen wie Self [US87], in denen alle Erzeugung von Objekten durch das Klonen eines Prototyps geschieht, von Haus aus vorhanden.

Beachten Sie die folgenden Aspekte, wenn Sie das Prototypmuster implementieren.

1. *Verwendung eines Prototypenverwalters.* Wenn die Anzahl von Prototypen eines Systems nicht von vorneherein festgelegt ist (das heißt, sie können dynamisch erzeugt und gelöscht werden), sollten Sie die verfügbaren Prototypen in einer Registratur vermerken. In diesem Fall verwalten Klienten die Prototypen nicht selbst, sondern speichern sie in der Registratur ab und fordern sie wieder an. Ein Klient holt sich einen Prototypen von der Registratur, bevor er ihn klont. Wir nennen diese Registratur einen **Prototypenverwalter**.

   Ein Prototypenverwalter verwendet einen assoziativen Speicher, der den zu einem Schlüssel passenden Prototypen zurückgibt. Er besitzt Operationen zum Registrieren eines Prototypen bezüglich eines Schlüssels und zum Auflösen der Registrierung. Klienten können die Registratur zur Laufzeit ändern oder sie durchsuchen. Somit können die Klienten das System ohne Schreiben von Code erweitern oder sich einen Überblick verschaffen.

2. *Implementierung der Klone-Operation.* Der schwierigste Aspekt am Prototypmuster ist die korrekte Implementierung der Klone-Operation. Sie ist insbesondere dann sehr trickreich, wenn die Objektstrukturen zirkuläre Referenzen enthalten.

   Die meisten Sprachen unterstützen das Klonen von Objekten zumindest teilweise. Zum Beispiel verfügt Smalltalk über eine Implementierung von `copy`, die von allen Unterklassen von `Object` geerbt wird. C++ bietet einen Kopierkonstruktor. Aber all diese Hilfsmittel lösen das Problem »flaches versus tiefes Ko-

pieren« nicht [GR83]. Kurzgefaßt stellt dieses Problem die Frage, ob das Klonen eines Objekts zum Klonen seiner Exemplarvariablen führt oder ob das Originalobjekt und der Klon dieselben Variablen miteinander teilen.

Eine flache Kopie ist einfach und oftmals ausreichend. Sie wird aus diesem Grund von Smalltalk als die Defaulteinstellung angeboten. Der standardmäßige Kopierkonstruktor in C++ kopiert die Exemplarvariablen, was bedeutet, daß im Fall von Objektreferenzen die referenzierten Objekte von Kopie und Original gemeinsam genutzt werden. Üblicherweise aber verlangt das Klonen von Prototypen mit komplexen Strukturen das Ausführen einer tiefen Kopie, weil der Klon und das Original unabhängig voneinander sein müssen. Sie müssen deswegen sicherstellen, daß die Komponenten des Klons wiederum Klons der Komponenten des Prototypen sind. Sie werden durch das Klonen gezwungen, zu entscheiden, was, wenn überhaupt, gemeinsam genutzt wird.

Wenn die Objekte im System Lade- und Speicheroperationen bieten, dann können Sie diese für eine Defaultimplementierung der Klone-Operation verwenden, indem Sie einfach das Objekt speichern und sofort wieder einladen. Die Speicheroperation legt das Objekt in einem Zwischenspeicher ab, und die Ladeoperation erzeugt ein Duplikat bei der Rekonstruktion des Objekts aus dem Speicher.

3. *Initialisierung geklonter Objekte.* Während manche Klienten mit dem geklonten Objekt völlig zufriedengestellt sind, verlangen andere Klienten die Initialisierung von Teilen oder dem gesamten inneren Zustand mit Werten ihrer Wahl. Sie können diese Werte üblicherweise nicht der Klone-Operation direkt übergeben, weil ihre Anzahl zwischen den Klassen der Prototypen variiert. Manche Prototypen benötigen möglicherweise mehrere Initialisierungsparameter, während andere Prototypen keine benötigen. Die Übergabe von Parametern in der Klone-Operation schließt eine einheitliche Schnittstelle zum Klonen aus.

   Vielleicht definieren die Klassen ihrer Prototypen ja schon Operationen zum Zurücksetzen wichtiger Teile des Zustands. Wenn dem so ist, können Klienten diese Operationen direkt nach dem Klonen verwenden. Wenn nicht, müssen sie vielleicht eine `Initialisiere`-Operation einführen (siehe Beispielcodeabschnitt), die die Initialisierungsparameter als Argumente entgegennimmt und den internen Zustand des geklonten Objekts entsprechend setzt. Besondere Aufmerksamkeit erfordern Klone-Operationen, die tiefe Kopien anfertigen – die Kopien müssen möglicherweise gelöscht werden, entweder explizit oder innerhalb von `Initialisiere`, bevor Sie sie erneut initialisieren können.

Beispielcode

Wir werden eine Unterklasse `LabyrinthPrototypFabrik` der Klasse `LabyrinthFabrik` (Seite 129) definieren. `LabyrinthPrototypFabrik` wird mit den Prototypen jener Objekte initialisiert, die es erzeugen soll, so daß wir keine Unterklassen bilden müssen, nur um die Klassen der von ihr erzeugten Räume und Wände zu ändern.

`LabyrinthPrototypFabrik` erweitert die Schnittstelle von `LabyrinthFabrik` mit einem Konstruktor, der die Prototypen als Argument erhält:

```
class LabyrinthPrototypFabrik : public LabyrinthFabrik {
public:
   LabyrinthPrototypFabrik(Labyrinth*, Wand*, Raum*, Tuer*);

   virtual Labyrinth* ErzeugeLabyrinth() const;
   virtual Raum* ErzeugeRaum(int) const;
   virtual Wand* ErzeugeWand() const;
   virtual Tuer* ErzeugeTuer(Raum*, Raum*) const;

private:
   Labyrinth* _prototypLabyrinth;
   Raum* _prototypRaum;
   Wand* _prototypWand;
   Tuer* _prototypTuer;
};
```

Der neue Konstruktor initialisiert einfach seine Prototypen:

```
LabyrinthPrototypFabrik::LabyrinthPrototypFabrik(
   Labyrinth* labyrinth, Wand* wand, Raum* raum, Tuer* tuer)
{
   _prototypLabyrinth = labyrinth;
   _prototypWand = wand;
   _prototypRaum = raum;
   _prototypTuer = tuer;
}
```

Die Member-Funktionen zum Erzeugen von Wänden, Räumen und Türen gleichen einander: Jede klont einen Prototypen und initialisiert ihn dann. Es folgen die Definitionen von `ErzeugeWand` und `ErzeugeTuer`:

```
Wand* LabyrinthPrototypFabrik::ErzeugeWand() const {
   return _prototypWand->Klone();
}
```

```
Tuer* LabyrinthPrototypFabrik::ErzeugeTuer(
   Raum* raum1, Raum* raum2) const
{
   Tuer* tuer = _prototypTuer->Klone();
   tuer->Initialisiere(raum1, raum2);
   return tuer;
}
```

Wir können `LabyrinthPrototypFabrik` zum Erzeugen eines prototypischen Labyrinths oder eines Defaultlabyrinths verwenden, indem wir es einfach mit Prototypen der grundlegenden Komponenten des Labyrinths initialisieren:

```
LabyrinthSpiel spiel;
LabyrinthPrototypFabrik einfacheLabyrinthFabrik(
   new Labyrinth, new Wand, new Raum, new Tuer);

Labyrinth* labyrinth = spiel.ErzeugeLabyrinth(
   einfacheLabyrinthFabrik);
```

Um den Labyrinthtyp zu ändern, initialisieren wir `LabyrinthPrototypFabrik` mit einer anderen Menge an Prototypen. Der folgende Aufruf erzeugt ein Labyrinth mit Prototypen für `BombardierbareTuer` und `RaumMitBombe`:

```
LabyrinthPrototypFabrik bombardierbareLabyrinthFabrik(
   new Labyrinth, new BombardierbareWand,
   new RaumMitBombe, new Tuer);
```

Ein Objekt, das als Prototyp genutzt werden kann, wie zum Beispiel ein Exemplar von `Wand`, muß die Klone-Operation unterstützen. Es muß ebenso über einen Kopierkonstruktor zum Klonen verfügen. Es benötigt möglicherweise eine separate Operation zur erneuten Initialisierung seines internen Zustands. Wir fügen deswegen die Operation `Initialisiere` der Klasse `Tuer` hinzu, um Klienten die Initialisierung geklonter Räume zu ermöglichen.

Vergleichen Sie die folgende Definition von `Tuer` mit derjenigen auf Seite 104:

```
class Tuer : public KartenEintrag {
public:
   Tuer();
   Tuer(const Tuer&);

   virtual void Initialisiere(Raum*, Raum*);
   virtual Tuer* Klone() const;
   virtual void Betrete();
```

```
   Raum* AndereSeiteVon(Raum*);

private:
   Raum* _raum1;
   Raum* _raum2;
};

Tuer::Tuer(const Tuer& andereTuer) {
   _raum1 = andereTuer._raum1;
   _raum2 = andereTuer._raum2;
}

void Tuer::Initialisiere(Raum* raum1, Raum* raum2) {
   _raum1 = raum1;
   _raum2 = raum2;
}

Tuer* Tuer::Klone() const {
   return new Tuer(*this);
}
```

Die Unterklasse `BombardierbareWand` muß die Klone-Operation überschreiben und einen entsprechenden Kopierkonstruktor implementieren.

```
class BombardierbareWand : public Wand {
public:
   BombardierbareWand();
   BombardierbareWand(const BombardierbareWand&);

   virtual Wand* Klone() const;
   bool IstBeschaedigt();

private:
   bool _istBeschaedigt;
};

BombardierbareWand::BombardierbareWand(
   const BombardierbareWand& andereWand) : Wand(andereWand)
{
   _istBeschaedigt = andereWand._istBeschaedigt;
}

Wand* BombardierbareWand::Klone() const {
   return new BombardierbareWand(*this);
}
```

Obwohl `BombardierbareWand::Klone` einen Zeiger vom Typ `Wand*` zurückgibt, gibt seine Implementierung einen Zeiger auf ein neues Exemplar einer Unterklasse, nämlich vom Typ `BombardierbareWand*` zurück. Wir definieren `Klone` solcherart in der Basisklasse, damit die Klienten nichts von den konkreten Unterklassen der Prototypen wissen müssen, die sie klonen. Klienten sollten auf dem Rückgabewert der Klone-Operation niemals einen Downcast zum erwünschten Typ ausführen müssen.

In Smalltalk können Sie die von `Object` geerbte Standardmethode `copy` wiederverwenden, um beliebige Prototypen von `KartenEintrag` Unterklassen zu klonen. Sie können eine `LabyrinthFabrik` verwenden, um die von Ihnen benötigten Prototypen zu produzieren. Beispielsweise können Sie einen Raum erzeugen, indem sie den Namen `#raum` übergeben. Die `LabyrinthFabrik` besitzt ein Dictionary, das Namen auf Prototypen abbildet. Seine `erzeuge:` Methode sieht folgendermaßen aus:

```
erzeuge: teilName
   ^ (teilKatalog at: teilName) copy
```

Wenn angemessene Methoden zur Initialisierung der `LabyrinthFabrik` mit Prototypen gegeben sind, können Sie ein einfaches Labyrinth mit dem folgenden Code erzeugen.

```
ErzeugeLabyrinth fuer:
   (LabyrinthFabrik new
   mit: Tuer new namens: #tuer;
   mit: Wand new namens: #wand;
   mit: Raum new namens: #raum;
   yourself)
```

Hierbei sähe die Definition der in `ErzeugeLabyrinth` verwendeten Klassenmethode `fuer:` so aus:

```
fuer: eineFabrik
   | raum1 raum2 |
   raum1 := (eineFabrik erzeuge: #raum) koordinate: 1@1.
   raum2 := (eineFabrik erzeuge: #raum) koordinate: 2@1.
   tuer := (eineFabrik erzeuge #tuer) von: raum1 nach: raum2.

   raum1
      aufSeite: #norden setze: (eineFabrik erzeuge: #wand);
      aufSeite: #osten setze: tuer;
      aufSeite: #sueden setze: (eineFabrik erzeuge: #wand);
      aufSeite: #westen setze: (eineFabrik erzeuge: #wand);
```

```
raum2
   aufSeite: #norden setze: (eineFabrik erzeuge: #wand);
   aufSeite: #osten setze: (eineFabrik erzeuge: #wand);
   aufSeite: #sueden setze: (eineFabrik erzeuge: #wand);
   aufSeite: #westen setze: tuer;
^ Labyrinth new
   fuegeRaumHinzu: raum1;
   fuegeRaumHinzu: raum2;
   yourself
```

Bekannte Verwendungen

Das vermutlich erste Beispiel des Prototypmusters ist in Ivan Sutherlands Sketchpad-System [Sut63] zu finden. Die erste weithin bekannte Anwendung des Musters in einer objektorientierten Sprache geschah in ThingLab, bei dem Benutzer ein zusammengesetztes Objekt erstellen und dann zu einem Prototypen machen konnten, indem Sie es in einer Bibliothek wiederverwendbarer Objekte installierten [Bor81]. Goldberg und Robson erwähnen Prototypen als Muster [GR83]. Allerdings gibt Coplien [Cop92] eine sehr viel umfassendere Beschreibung. Er beschreibt dem Prototypmuster verwandte C++-Idiome und nennt viele Beispiele und Variationen.

Etgdb ist ein auf ET++ basierendes Debugger-Frontend, das eine Point-and-Click-Benutzungsschnittstelle für verschiedene zeilenorientierte Debugger bietet. Zu jedem Debugger gibt es eine entsprechende DebuggerAdaptor-Unterklasse. Beispielsweise paßt GdbAdaptor etgdb an die Befehlssyntax von GNUs gdb an, während SunDbxAdaptor dasselbe für Suns dbx debugger tut. Etgdb verfügt nicht über eine fest codierte Menge von DebuggerAdaptor-Klassen, sondern es liest den Namen des zu verwendenden Adapters aus einer Umgebungsvariablen ein, sucht in einer globalen Tabelle nach einem Prototypen mit dem angegebenen Namen und klont den Prototypen. Neue Debugger können zu etgdb hinzugefügt werden, in dem man es mit dem DebuggerAdaptor für den Debugger zusammen linkt.

Die »interaction technique library« (Bibliothek für Interaktionstechniken) in ModeComposer speichert die Prototypen von Objekten, die unterschiedliche Interaktionstechniken unterstützen [Sha90]. Jede vom ModeComposer erzeugte Interaktionstechnik kann als Prototyp genutzt werden, indem man ihn in die Bibliothek einfügt. Das Prototypmuster ermöglicht es ModeComposer, eine unbegrenzte Anzahl von Interaktionstechniken zu unterstützen.

Das eingangs diskutierte Musikeditorbeispiel basiert auf dem Unidraw Zeichen-Framework [VL90].

Verwandte Muster

Wie am Ende des Kapitels diskutiert wird, konkurrieren das Prototypmuster und das Abstrakte-Fabrik-Muster (107) in verschiedener Hinsicht miteinander. Sie können auch zusammen angewendet werden. Eine abstrakte Fabrik könnte eine Menge von Prototypen speichern, die geklont und zurückgegeben werden.

Sich stark auf das Kompositions- (239) und Dekorierermuster (199) abstützende Entwürfe können ebenfalls oft vom Prototypmuster profitieren.

Singleton

Ein objektbasiertes Erzeugungsmuster

Zweck

Sichere ab, daß eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

Motivation

Bei manchen Klassen ist es wichtig, daß es genau ein Exemplar gibt. Obwohl es in einem System viele Drucker geben kann, sollte es nur einen Druckerspooler geben. Es sollte nur ein Dateisystem und nur eine Fensterverwaltung geben. Ein digitaler Filter besitzt einen A/D-Konvertierer. Ein Buchhaltungssystem dient während es arbeitet genau einer Firma.

Wie stellen wir sicher, daß eine Klasse über genau ein Exemplar verfügt und daß einfach auf dieses Exemplar zugegriffen werden kann? Eine globale Variable ermöglicht den Zugriff auf ein Objekt, verhindert aber nicht das Erzeugen mehrerer Exemplare.

Es ist besser, die Klasse selbst für die Verwaltung ihres einzigen Exemplars zuständig zu machen. Die Klasse kann durch Abfangen von Befehlen zur Erzeugung neuer Objekte sicherstellen, daß kein weiteres Exemplar erzeugt wird, und sie kann die Zugriffsmöglichkeit auf das Exemplar anbieten. Dies ist die Essenz des Singletonmusters.

Anwendbarkeit

Verwenden Sie das Singletonmuster, wenn

- es genau ein Exemplar einer Klasse geben und es für Klienten an einem wohldefinierten Punkt zugreifbar sein muß.
- das einzige Exemplar durch Bildung von Unterklassen erweiterbar sein soll und Klienten in der Lage sein sollen, das erweiterte Exemplar ohne Modifikation ihres Codes verwenden zu können.

Struktur

Abbildung 3.12 zeigt die Struktur des Singletonmusters.

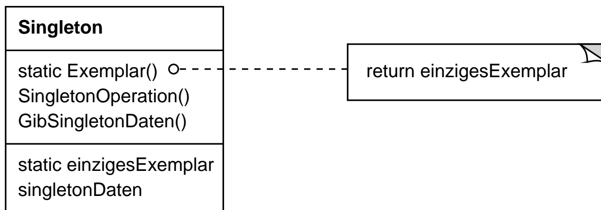

Abbildung 3.12

Teilnehmer

- **Singleton**
  - definiert eine Exemplaroperation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen. Exemplar ist eine Klassenoperation, also eine Klassenmethode in Smalltalk und eine statische Member-Funktion in C++.
  - ist potentiell für die Erzeugung seines einzigen Exemplars zuständig.

Interaktionen

Klienten greifen auf ein Singletonexemplar ausschließlich durch die Exemplar-Operation der Singletonklasse zu.

Konsequenzen

Das Singletonmuster besitzt mehrere Vorteile:

1. *Zugriffskontrolle auf das Exemplar.* Da die Singletonklasse sein einziges Exemplar kapselt, verfügt es über eine strikte Kontrolle darüber, wie und wann die Klienten auf das Exemplar zugreifen können.

2. *Eingeschränkter Namensraum.* Das Singletonmuster ist eine Verbesserung gegenüber globalen Variablen. Es vermeidet die Überfrachtung des Namensraums mit globalen Variablen, welche die Singletonexemplare speichern.

3. *Verfeinerung von Operationen und Repräsentation.* Die Singletonklasse kann abgeleitet und spezialisiert werden. Zudem ist es einfach, eine Anwendung mit einem Exemplar dieser erweiterten Klasse zu konfigurieren. Sie können sogar die Anwendung mit einem Exemplar der benötigten Klasse zur Laufzeit konfigurieren.

4. *Variable Anzahl von Exemplaren.* Sollten Sie Ihre Meinung ändern und doch mehr als ein Exemplar der Singletonklasse benötigen, so macht das Muster es Ihnen auch hierbei leicht. Sie können weiterhin denselben Ansatz dazu verwenden, die Anzahl der von der Anwendung benutzten Exemplare zu steuern. Sie müssen dafür nur jene Operation ändern, die den Zugriff auf das Singletonexemplar ermöglicht.

5. *Flexibler als Klassenoperationen.* Eine andere Möglichkeit, die Funktionalität eines Singletons zusammenzufassen, besteht in der Verwendung von Klassenoperationen, also statischen Member-Funktionen in C++ oder Klassenmethoden in Smalltalk. Beide Sprachtechniken erschweren die Änderung eines Entwurfs, möchte man doch über mehr als ein Exemplar einer Klasse verfügen. In C++ sind statische Member-Funktionen zudem niemals virtuell, so daß Unterklassen sie nicht polymorph überschreiben können.

Implementierung

Es folgen einige Implementierungsaspekte, die man bei der Anwendung des Singletonmusters bedenken sollte:

1. *Garantie eines einzigen Exemplars.* Das Singletonmuster macht das einzige Exemplar zu einem normalen Exemplar seiner Klasse. Die Klasse ist so geschrieben, daß nur ein einziges Exemplar jemals erzeugt werden kann. Üblicherweise versteckt man die das Exemplar erzeugende Operation hinter einer Klassenoperation, die garantiert, daß nur ein Exemplar erzeugt wird. Diese Operation kann auf die Variable, die das einzige Exemplar enthält, zugreifen, und sie stellt sicher, daß die Variable mit dem einzigen Exemplar initialisiert ist, bevor sie ihren Wert zurückgibt. Dieser Ansatz stellt sicher, daß ein Singleton vor seiner ersten Benutzung erzeugt und initialisiert wird.

   Sie können die Klassenoperation in C++ mit einer statischen Member-Funktion `Exemplar` der Klasse `Singleton` definieren. `Singleton` definiert weiterhin eine statische Member-Variable `_exemplar`, die einen Zeiger auf sein einziges Exemplar enthält.

   Die Klasse `Singleton` ist folgendermaßen deklariert:

   ```
   class Singleton {
   public:
      static Singleton* Exemplar();

   protected:
      Singleton();
   ```

```
private:
      static Singleton* _exemplar;
};
```

Die entsprechende Implementierung sieht so aus:

```
Singleton* Singleton::_exemplar = 0;

Singleton* Singleton::Exemplar() {
   if (_exemplar == 0) {
      _exemplar = new Singleton;
   }
   return _exemplar;
}
```

Klienten greifen auf das Singleton ausschließlich durch die `Exemplar` Member-Funktion zu. Die Variable `_exemplar` wird mit 0 initialisiert, und die statische Member-Funktion `Exemplar` gibt seinen Wert zurück, wobei sie ihn mit dem einzigen Exemplar initialisiert, wenn er auf 0 steht. `Exemplar` verwendet verzögerte Initialisierung (lazy initialization); der zurückgegebene Wert wird nicht erzeugt und gespeichert, bis das erste Mal darauf zugegriffen wird.

Beachten Sie, daß der Konstruktor geschützt ist, das heißt als `protected` deklariert ist. Versucht ein Klient, ein Singleton direkt zu erzeugen, so ergibt sich zur Übersetzungszeit ein Fehler. Dies stellt sicher, daß nur ein Exemplar erzeugt wird.

Da `_exemplar` ein Zeiger auf ein Singletonobjekt ist, kann die `Exemplar` Member-Funktion ihm weiterhin einen Zeiger auf ein Exemplar einer Unterklasse von Singleton zuweisen. Wir werden im Beispielcodeabschnitt ein Beispiel vorführen.

Ein letzte Bemerkung zur C++-Implementierung: Es ist nicht ausreichend, das Singleton als globales oder statisches Objekt zu definieren und sich dann auf die automatische Initialisierung zu verlassen. Dafür gibt es drei Gründe:

a. Wir können nicht garantieren, daß insgesamt nur ein Exemplar eines statischen Objekts erzeugt werden wird.

b. Wir verfügen möglicherweise nicht über genügend Informationen, jedes Singleton zur statischen Initialisierungszeit zu erzeugen. Ein Singleton benötigt möglicherweise Werte, die erst später während des Programmablaufs berechnet werden.

c. C++ definiert über Übersetzungseinheiten hinweg keine Reihenfolge, in der die Konstruktoren globaler Objekte aufgerufen werden [ES90]. Dies führt dazu, daß keine Abhängigkeiten zwischen den Singletons existieren dürfen. Gäbe es sie, so wären Fehler unvermeidlich.

Ein zusätzlicher, wenngleich weniger wichtiger Nebeneffekt des globalen/statischen Objektansatzes besteht darin, daß alle Singletons erzeugt werden, ob sie nun benutzt werden oder nicht. Die Verwendung einer statischen Member-Funktion vermeidet all diese Probleme.

In Smalltalk wird die Funktion, die das einzige Exemplar zurückgibt, als eine Klassenmethode der Singletonklasse implementiert. Um sicherzustellen, daß nur ein Exemplar erzeugt wird, überschreibt man die `new`-Operation. Somit besitzt die resultierende Singletonklasse beispielsweise die folgenden zwei Klassenmethoden. Hierbei ist `EinzigesExemplar` eine Klassenvariable, die nirgendwo anders verwendet wird:

```
new
   self error: 'kann kein Objekt erzeugen'

default
   EinzigesExemplar isNil ifTrue:[EinzigesExemplar := super new].
   ^ EinzigesExemplar
```

2. *Ableiten der Singletonklasse.* Das Hauptproblem besteht nicht so sehr in der Definition der Unterklasse, sondern in der Installation seines einzigen Exemplars, so daß Klienten es verwenden können. Im wesentlichen muß die Variable, die das Singletonexemplar referenziert, mit einem Exemplar der Unterklasse initialisiert werden. Die einfachste Technik besteht in der Bestimmung des zu verwendenden Singletons in der Operation `Exemplar` der Singletonklasse. Ein Beispiel im Beispielcodeabschnitt zeigt, wie man diese Technik mittels Umgebungsvariablen implementiert.

   Eine weitere Möglichkeit, die Singletonunterklasse auszuwählen, besteht darin, die Implementierung von `Exemplar` aus seiner Oberklasse (zum Beispiel `LabyrinthFabrik`) zu entfernen und in die Unterklasse einzufügen. Dies ermöglicht es einem C++-Programmierer, die Klasse des Singletons erst während des Bindens zu wählen (zum Beispiel durch das Binden einer Objektcodedatei mit einer anderen Implementierung). Die Klasse bleibt aber weiterhin vor den Klienten des Singletons versteckt.

   Dieser Ansatz legt die Wahl der Singletonklasse zur Bindezeit fest. Somit ist es schwer, die Singletonklasse zur Laufzeit zu wählen. Die Verwendung von be-

dingten Anweisungen zur Bestimmung der Unterklasse ist flexibler, legt aber die Menge möglicher Singletonklassen im Code fest. Keiner der Ansätze ist für alle Fälle flexibel genug.

Ein flexiblerer Ansatz verwendet eine *Registratur für Singletons*. Statt die Exemplaroperation die Anzahl möglicher Singletonklassen definieren zu lassen, melden sich die Singletonexemplare über ihren Namen bei einer wohlbekannten Registratur an.

Die Registratur bildet von stringbasierten Namen auf Singletons ab. Wenn `Exemplar` ein Singleton benötigt, konsultiert es die Registratur und fragt mittels des Namens nach dem Singleton. Die Registratur sucht das entsprechende Singleton heraus (sofern es existiert) und gibt es zurück. Dieser Ansatz befreit `Exemplar` vom Wissen um alle möglichen Singletonklassen oder Exemplare. Es verlangt einzig eine gemeinsame Schnittstelle aller Singletonklassen. Diese enthält auch die Operationen für die Registratur:

```
class Singleton {
public:
   static void Registriere(char* name, Singleton*);
   static Singleton* Exemplar();

protected:
   static Singleton* Suche(const char* name);

private:
   static Singleton* _exemplar;
   static Liste<NameSingletonPaar>* _registratur;
};
```

Registriere registriert das Singletonexemplar unter dem gegebenen Namen. Um die Registratur einfach zu halten, speichern wir die Exemplare in einer Liste von `NameSingletonPaar`-Objekten. Jedes `NameSingletonPaar` bildet einen Namen auf ein Singleton ab. Die Operation `Suche` sucht auf Basis eines übergebenen Namen das Singleton heraus. Wir machen dabei die Annahme, daß eine Umgebungsvariable den Namen der gewünschten Singletons spezifiziert.

```
Singleton* Singleton::Exemplar() {
   if (_exemplar == 0) {
      const char* exemplarName = getenv("SINGLETON");
      // Benutzer oder Umgebung setzen die Variable
      // beim Hochfahren
```

```
        _exemplar = Suche(exemplarName);
        // Suche gibt 0 zurück, wenn es noch kein Singleton gibt
    }
    return _exemplar;
}
```

Zu welchem Zeitpunkt registrieren sich Singletonklassen selbst? Eine Möglichkeit ist ihr Konstruktor. Beispielsweise könnte eine Unterklasse `MeinSingleton` das folgende machen:

```
MeinSingleton::MeinSingleton() {
    //...
    Singleton::Registriere("MeinSingleton", this);
}
```

Natürlich wird der Konstruktor nicht aufgerufen werden, bis jemand ein Objekt der Klasse erzeugt, was seinerseits genau jenes Problem darstellt, das das Singletonmuster zu lösen versucht! In C++ können wir das Problem umgehen, indem wir ein statisches Exemplar von `MeinSingleton` definieren. Wir können zum Beispiel in der Datei, in der `MeinSingleton` implementiert wird, folgendes schreiben:

```
static MeinSingleton dasSingleton;
```

Die Singletonklasse ist somit nicht länger für das Erzeugen des Singletons zuständig. Seine primäre Aufgabe besteht darin, die gewünschten Singletonobjekte im System verfügbar zu machen. Der Ansatz, statische Objekte zu verwenden, hat immer noch einen potentiellen Nachteil – es müssen nämlich Exemplare von allen möglichen Singletonunterklassen erzeugt werden, da sie andernfalls nicht registriert werden.

Beispielcode

Stellen Sie sich vor, daß wir eine wie auf Seite 129 beschriebene `LabyrinthFabrik` zum Aufbau von Labyrinthen definieren wollen. `LabyrinthFabrik` definiert eine Schnittstelle zum Aufbau von unterschiedlichen Teilen des Labyrinths. Unterklassen können die Operationen neu definieren, so daß sie Exemplare spezialisierter Produktklassen zurückgeben (zum Beispiel `BombardierbareWand`-Objekte statt einfachen `Wand`-Objekten).

Hierbei ist wichtig zu bemerken, daß die Labyrinthanwendung nur ein Exemplar der Labyrinthfabrik benötigt. Dieses Exemplar sollte für jeglichen Code verfügbar sein, der Teile des Labyrinths konstruiert. Hier kommt das Singletonmuster ins

Spiel. Indem wir die LabyrinthFabrik zur Singletonklasse machen, können wir auch das Labyrinthobjekt allgemein zugreifbar machen, ohne uns auf globale Variablen abstützen zu müssen.

Um das Beispiel zu vereinfachen, nehmen wir an, daß wir LabyrinthFabrik niemals ableiten werden (die Alternative werden wir in Kürze betrachten). In C++ machen wir es zu einer Singletonklasse, indem wir eine statische Exemplar-Operation und eine statische _exemplar Member-Variable hinzufügen, um das einzige Exemplar zu halten. Wir müssen weiterhin den Konstruktor schützen, um eine zufällige Erzeugung von Objekten, die zu mehr als einem Exemplar führen könnte, zu verhindern.

```
class LabyrinthFabrik {
public:
   static LabyrinthFabrik* Exemplar();

   // existierende Schnittstelle folgt hier
   // ...

protected:
   LabyrinthFabrik();

private:
   static LabyrinthFabrik* _exemplar;
};
```

Die entsprechende Implementierung sieht so aus:

```
LabyrinthFabrik* LabyrinthFabrik::_exemplar = 0;

LabyrinthFabrik* LabyrinthFabrik::Exemplar() {
   if (_exemplar == 0) {
      _exemplar = new LabyrinthFabrik;
   }
   return _exemplar;
}
```

Lassen Sie uns nun betrachten, was passiert, wenn es Unterklassen der LabyrintFabrik gibt und die Anwendung entscheiden muß, welche zu verwenden ist. Wir wählen die Art des Labyrinths durch eine Umgebungsvariable aus und fügen Code hinzu, der ein Objekt der passenden LabyrinthFabrik-Unterklasse auf Basis des Werts dieser Umgebungsvariablen erzeugt. Die Exemplar-Operation stellt einen guten Ort für diesen Code dar, weil sie die LabyrinthFabrik bereits erzeugt:

```
LabyrinthFabrik* LabyrinthFabrik::Exemplar() {
   if (_exemplar == 0) {
      const char* labyrinthStil = getenv("LABYRINTHSTIL");

      if (strcmp(labyrinthStil, "mitbomben") == 0) {
         _exemplar = new LabyrinthMitBombenFabrik;
      }
      else if (strcmp(labyrinthStil, "verzaubert") == 0) {
         _exemplar = new VerzaubertesLabyrinthFabrik;
      }
      // ... weitere mögliche Unterklassen
      else {
         _exemplar = new LabyrinthFabrik;
      }
   }
   return _exemplar;
}
```

Beachten Sie, daß `Exemplar` jedesmal modifiziert werden muß, wenn Sie ein neue Unterklasse der `LabyrinthFabrik` definieren. Dies stellt vermutlich kein Problem für diese Anwendung dar, sähe aber im Fall einer in einem Framework definierten abstrakten Fabrik schon anders aus.

Eine mögliche Lösung besteht in der Verwendung des im Implementierungsabschnitts beschriebenen Registraturansatzes. Dynamisches Binden kann sich hierbei ebenfalls als nützlich herausstellen – es würde die Anwendung davon abhalten, alle nicht benötigten Unterklassen laden zu müssen.

Bekannte Verwendungen

Ein Beispiel des Singletonmusters in Smalltalk-80 [Par90] ist die Menge von Änderungen des Codes, was als `ChangeSet current` verfügbar ist. Ein subtileres Beispiel ist die Beziehung zwischen Klassen und ihren **Metaklassen**. Eine Metaklasse ist die Klasse einer Klasse, und jede Metaklasse besitzt genau ein Exemplar. Metaklassen haben keine Namen (außer über einen indirekten Weg: durch den Namen ihrer einzigen Exemplare), aber sie verwalten ihre Exemplare und erzeugen üblicherweise keine weiteren Exemplare.

InterViews, eine Klassenbibliothek zur Erstellung von Benutzungsschnittstellen [LCI+92], verwendet das Singletonmuster unter anderem zum Zugriff auf die einzigen Exemplare seiner Klassen Session und WidgetKit. Session definiert die Hauptschleife der Anwendung zum Dispatch von Ereignissen, speichert und lädt die Datenbank an stilistischen Voreinstellungen des Benutzers und verwaltet die

Verbindungen zu einem oder mehreren physikalischen Bildschirmen. WidgetKit ist eine abstrakte Fabrik (107) zur Definition des Look-and-Feels von Benutzungsschnittstellenwidgets. Die `WidgetKit::instance()`-Operation bestimmt die jeweilige Unterklasse von WidgetKit, die unter Verwendung der von Session definierten Umgebungsvariable erzeugt wird. Eine ähnliche Operation von Session legt fest, ob monochrome oder Farbbildschirme unterstützt werden, und sie konfiguriert das einzige Session-Exemplar dementsprechend.

Verwandte Muster

Viele Muster können unter Verwendung des Singletonmusters implementiert werden, so zum Beispiel das Abstrakte-Fabrik-Muster (107), das Erbauermuster (119) und das Prototypmuster (144).

3.1 Diskussion der Erzeugungsmuster

Es gibt zwei bekannte Möglichkeiten, ein System mit den Klassen von ihm erzeugter Objekte zu parametrieren. Eine Möglichkeit besteht darin, eine Unterklasse der Klasse zu erstellen, welche die Objekte erzeugt; dies entspricht der Verwendung des Fabrikmethodemusters. Der Nachteil dieses Ansatzes besteht darin, daß man möglicherweise eine neue Unterklasse erzeugen muß, nur um die Klasse des Produkts zu ändern. Solche Änderungen können sich kaskadenförmig fortpflanzen. Wenn beispielsweise der Erzeuger eines Produkts selbst von einer Fabrikmethode erzeugt wird, so müssen Sie wiederum seinen Erzeuger spezialisieren.

Die andere Möglichkeit, ein System zu parametrieren, besteht in der Anwendung von Objektkomposition: Definieren Sie ein Objekt, das für das Wissen um die Klasse von Produktobjekten zuständig ist, und machen Sie es zu einem Parameter des Systems. Dies ist ein zentraler Aspekt des Abstrakte-Fabrik-Musters (107), des Erbauermusters (119) und des Prototypmusters (144). Alle drei führen zur Erzeugung eines neuen »Fabrikobjekts«, dessen Zuständigkeit im Erzeugen von Produktobjekten liegt. Das Abstrakte-Fabrik-Muster verwendet ein Fabrikobjekt, das Objekte mehrerer verschiedener Klassen erzeugen kann. Erbauer verwendet ein Fabrikobjekt, das ein komplexes Produkt unter Verwendung eines ebenso komplexen Protokolls inkrementell konstruiert. Prototyp verwendet ein Fabrikobjekt, das ein Produkt durch Kopieren eines Prototypobjekts erzeugt. In diesem Fall sind Fabrikobjekt und Prototyp dasselbe Objekt, da der Prototyp für die Rückgabe des Produkts zuständig ist.

Stellen Sie sich das zum Prototypmuster beschriebene Framework für Zeicheneditoren vor. Es gibt verschiedene Wege, ein GrafischesWerkzeug-Objekt mit der Klasse seines Produkts zu parametrieren:

- Die Anwendung des Fabrikmethodemusters führt zur Erzeugung einer Unterklasse von GrafischesWerkzeug für jede GrafischesObjekt-Klasse in der Palette. GrafischesWerkzeug verfügt dann über eine NeuesGrafischesObjekt-Operation, die jede GrafischesWerkzeug-Unterklasse überschreibt.

- Die Anwendung des Abstrakte-Fabrik-Musters führt zur einer Klassenhierarchie von GrafischeFabrik-Objekten, eines für jede GrafischesObjekt-Unterklasse. Jede Fabrik erzeugt in diesem Fall genau ein Produkt: KreisFabrik erzeugt Kreise, LinienFabrik erzeugt Linien usw. Ein GrafischesWerkzeug wird mit einer Fabrik zum Erzeugen der passenden GrafischesObjekt-Klasse parametrisiert.

- Die Anwendung des Prototypmusters führt zur Implementierung der Klone-Operation für jede Unterklasse von GrafischesObjekt. Jedes GrafischesWerkzeug-Objekt ist mit einem Prototypen der von ihm erzeugten grafischen Objekte parametrisiert.

Es hängt von vielen Faktoren ab, welches Muster am besten geeignet ist. Im Fall unseres Zeicheneditor-Frameworks ist das Fabrikmethodemuster das anfangs am einfachsten zu verwendende Muster. Es ist einfach, eine neue Unterklasse von GrafischesWerkzeug zu definieren. Die Exemplare von GrafischesWerkzeug werden nur dann erzeugt, wenn die Palette definiert wird. Der wesentliche Nachteil ist, daß die GrapischesWerkzeug-Klassen sich fortpflanzen, aber keine von Ihnen sonderlich viel tut.

Das Abstrakte-Fabrik-Muster bietet keine große Verbesserung der Situation, da es eine genauso große GrafischeFabrik-Klassenhierarchie benötigt. Eine abstrakte Fabrik wäre der Fabrikmethode nur dann vorzuziehen, wenn es bereits eine GrafischeFabrik-Klassenhierarchie gäbe – entweder, weil der Übersetzer es automatisch anbietet (wie in Smalltalk oder Objective-C) oder weil es in anderen Teilen des Systems gebraucht wird.

Insgesamt betrachtet ist das Prototypmuster wohl die beste Lösung für das Zeicheneditor-Framework, weil es nur die Implementierung einer Klone-Operation für jede GrafischesObjekt-Klasse verlangt. Dies reduziert die Anzahl von Klassen. Zudem kann Klone für andere Zwecke als die reine Erzeugung von Objekten verwendet werden (zum Beispiel für die Implementierung eines Dupliziere-Menüeintrags).

Eine Fabrikmethode macht einen Entwurf leichter anpaßbar und nur wenig komplizierter. Andere Entwurfsmuster benötigen neue Klassen, während eine Fabrikmethode nur eine neue Operation verlangt. Entwickler verwenden Fabrikmethoden oft als Standardlösung zum Erzeugen von Objekten. Fabrikmethoden werden nicht benötigt, wenn die zu Klasse zu erzeugender Objekte sich nie ändert oder wenn die Erzeugung der Objekte an einem Ort stattfindet, den Unterklassen leicht überschreiben können, so zum Beispiel in einer Initialisierungsoperation.

Entwürfe, die das Abstrakte-Fabrik-Muster, das Prototypmuster oder das Erbauermuster verwenden, sind sogar noch flexibler als Entwürfe, die das Fabrikmethodemuster benutzen. Sie sind aber auch komplexer. Oft beginnen Entwürfe mit der Verwendung von Fabrikmethoden und entwickeln sich dann zur Anwendung der anderen Erzeugungsmuster hin, wenn die Entwickler feststellen, daß Sie mehr Flexibilität benötigen. Die Kenntnis mehrerer Entwurfsmuster gibt Ihnen eine größere Auswahl an Möglichkeiten beim wechselseitigen Abwägen von Entwurfskriterien.