Eine Pattern-Sprache zur Erzeugung von Objekten

(Java Spektrum 2/2002)
Arno Haase
Arno Haase Consulting
Arno.Haase@Haase-Consulting.com

Einführung

Das Erzeugen von Objekten ist ein Thema, das im Umfeld der Programmiersprache Java oft stiefmütterlich behandelt wird - mit entsprechend unerfreulichen Konsequenzen. Dieser Artikel stellt eine kleine Pattern-Sprache vor, die dieses Versäumnis nachholt.

Patterns und Pattern-Sprachen

Patterns sind zunächst einmal nichts anderes als eine literarische Form, die besonders gut geeignet ist, um Erfahrungen und bewährte Lösungen weiterzugeben.

Patterns bestehen aus drei Teilen. Sie sind *Lösungen* zu *Problemen* in einem *Kontext*. Der Kontext ist in diesem Fall der gleiche, nämlich die Erzeugung von Objekten in Java.

Das Problem ist das Herz eines Patterns. Jeder hat schon einmal ein Stück Software geschrieben, das ihm besonders gut gefallen hat, eine besonders schöne Lösung gefunden. Aber welches Problem löst diese Lösung? Wie lässt es sich verallgemeinern? Welche Hindernisse sind dabei zu überwinden? Eine gut formulierte Problembeschreibung zeigt auf, in welchen Situationen das Pattern anwendbar ist - und in welchen nicht.

Die Lösung schließlich ist - na ja, eben die Lösung. Sie sollte so konkret sein, dass man sie anschließend implementieren kann, und sie sollte auf Vor- und Nachteile hinweisen.

Pattern-Sprachen sind Sammlungen von Patterns zu einem gemeinsamen Thema. Jedes Pattern in einer Pattern-Sprache löst zwar für sich genommen ein Problem, aber sie stehen

ausserdem in Beziehungen zueinander. Wenn man eines der Patterns anwendet, entsteht eine neue Situation, und die Beziehungen in einer Pattern-Sprache geben Hinweise auf andere Patterns, die oft geeignet sind, diese neue Situation weiter zu verbessern.

Die acht Patterns in diesem Artikel bilden eine solche Pattern-Sprache.

Creation Interface

Plane beim Schreiben einer Klasse nicht nur ihre Funktionalität sondern auch die Mechanismen, mit denen Clients Instanzen erhalten.

Problem. Wenn man eine Klasse schreibt, dann denkt man oft zunächst einmal daran, wie Clients mit fertigen Objekten interagieren. Die Klasse soll ja einen Nutzen bringen, und der liegt in ihren Methoden.

Dadurch entstehen dann Klassen, bei denen man Aufwand treiben und Nachforschungen anstellen muss, bevor man eine Instanz hat, die man benutzen kann - frei nach dem Sprichwort “Erst die Arbeit, dann das Vergnügen”.

Oft wird diese Komplexität durch Konventionen und mündliche Tradition aufgefangen, im Stil von: “Ach so, nein, dieser Konstruktor war eigentlich nur für die Verwendung in jenem Kontext gedacht” oder: “Das musste ja schiefgehen, Du hast ja vergessen, das Objekt nach der Erzeugung bei XY zu registrieren”.

Gerade in Java, wo die Virtual Machine hinter einem die nicht mehr benötigten Objekte aufräumt, unterschätzt man leicht die Komplexität der Erzeugung von Objekten, und das grausige Resultat sind allzu oft komplizierte Quelltexte, duplizierter Code und letztenendes subtile Fehler.

Lösung. Behandle die Erzeugung und Bereitstellung von Instanzen als eine zweite Schnittstelle, die jede Klasse zusätzlich zur eigentlichen Funktionalität bereitstellt. Dieses Creation Interface steht gleichberechtigt neben

den eigentlichen Methoden und muss ebenso sorgfältig entworfen und gepflegt werden.

Das Ziel dabei ist, es für Clients so einfach wie möglich zu machen, Objekte zu bekommen, auf denen sie dann Methoden aufrufen können. Das sollte so unkompliziert und wenig fehleranfällig wie nur irgend möglich sein. Insbesondere sollten die Details der eigentlichen Erzeugung von Objekten möglichst gut gekapselt sein, damit Clients sich nicht darum kümmern müssen.

Initializing Constructor

Jeder Konstruktor sollte sicherstellen, dass das Objekt vollständig und sinnvoll initialisiert ist, und andernfalls eine Exception werfen.

Problem. Viele Klassen verlassen sich darauf, dass die Felder ihrer Instanzen bestimmten Bedingungen genügen (sogenannte "Invarianten" erfüllen), was ein Prinzip objektorientierter Entwicklung ist [Meyer] und sich oft in der Praxis bewährt. Wie kann man im Creation Interface sicherstellen, dass diese Annahmen von Anfang an erfüllt sind?

So verlässt sich z.B. Klasse LizenzManager in Listing 1a darauf, dass die Felder sinnvoll initialisiert sind. Das geschieht durch einen gesonderten Aufruf von readLizenzDatei, etwa folgendermaßen:

```
LizenzManager mgr = new LizenzManager ();
try {
  mgr.readLizenzDatei ("license.dat");
}
catch (IOException exc) {...}
```

So weit, so gut - aber was, wenn der Client vergisst, die zusätzliche Methode aufzurufen? Oder wenn beim Lesen der Datei eine Exception auftaucht, so dass das LizenzManager-Objekt nur halb initialisiert ist? Jedenfalls muss sich der Client in erheblicher Weise um die Initialisierung des Objektes kümmern - mit entsprechenden Möglichkeiten, Fehler zu machen.

Lösung. Stelle sicher, dass jeder Konstruktor der Klasse das Objekt vollständig und sinnvoll initialisiert und andernfalls eine Exception wirft. Wenn der Konstruktor exceptionfrei

```
// unsichere Implementierung: Erzeugung
// und Initialisierung sind getrennt
public class LizenzManager {
  private Date _gueltigBis;
  private String _typ;

  // Default-Konstruktor wird vom
  // Compiler generiert, und neue
  // Instanzen werden nicht
  // initialisiert

  public boolean istLizenzGueltig () {
    return _gueltigBis.
        after (new Date ());
  }

  public String getNachricht () {
    return "Lizenz " + _typ +
        " gültig bis " + _gueltigBis;
  }

  public void readLizenzDatei (String
      dateiname) throws IOException
  {
    ... // komplizierter Code zum
        // Einlesen der Lizenzdatei
  }
}
```

Listing 1a: Die Klasse *LizenzManager*

durchläuft, dann kann sich der Client darauf verlassen, dass das Objekt richtig initialisiert ist; andernfalls gibt es zumindest kein falsch oder gar nicht initialisiertes Objekt, das eine Fehlerquelle sein könnte.

Listing 1b zeigt eine entsprechend erweiterte Version der LizenzManager-Klasse. Statt des vom Compiler erzeugten Default-Konstruktors hat sie einen Konstruktor, der einen Dateinamen entgegennimmt und die entsprechende Datei einliest. Wenn es dabei ein Problem gibt, dann wirft eben der Konstruktor eine Exception.

Ein weit verbreiteter Grund, die Initialisierung in einen zweiten Schritt nach der Ausführung des Konstruktors auszulagern, ist eine falsche Scheu, dass Konstruktoren zur Ausführung zu lange brauchen oder Exceptions werfen könnten; für eine solche Scheu gibt es zumindest in Java keinen Grund.

Eine kontrovers diskutierte Frage im Zusammenhang mit diesem Pattern ist, in wie weit der Konstruktor Parameter auf ihre Plausibilität prüfen sollte. Häufig entzündet sich die Dis-

```
// sichere Implementierung: Konstruktor
// stellt die Initialisierung sicher
public class LizenzManager {
  private Date _gueltigBis;
  private String _typ;

  /** Initialisierender Konstruktor.
      Weil kein Default-Konstruktor
     erzeugt wird, ist dies der einzige
      Konstruktor, und das Objekt
      wird initialisiert */
  public LizenzManager (String typ,
      Date gueltigBis)
  {
    _typ = typ;
    _gueltigBis = gueltigBis;
  }

  public boolean istLizenzGueltig () {
    return _gueltigBis.
      after (new Date ());
  }

  public String getNachricht () {
    return "Lizenz " + _typ +
        " gültig bis " + _gueltigBis;
  }

  public void readLizenzDatei (String
      dateiname) throws IOException
  {
    ... // komplizierter Code zum
        // Einlesen der Lizenzdatei
  }
}
```

Listing 1b: Erweiterte Version der *LizenzManager*-Klasse

kussion daran, ob eine Referenz ausdrücklich auf null geprüft werden sollte oder nicht, in der folgenden Klasse Name also die beiden String-Parameter des Konstruktors.

```
public class Name {
  private final String _vorname;
  private final String _nachname;
  public Name (String vorname,
               String nachname) {
    _vorname = vorname;
    _nachname = nachname;
  }
  ...
}
```

Dieser Konstruktor speichert klaglos null-Referenzen als Namen, was eine potentielle Fehlerquelle ist. Andererseits ist es fraglich, ob der zusätzliche Aufwand einer Überprüfung, der den Quelltext länger und unübersichtlicher machen würde, sich hier lohnt.

Die Frage halte ich nicht für sehr wichtig. Meiner Ansicht nach lohnt sich die Überprüfung zunächst nur dort, wo nicht klar ist, ob null ein erlaubter Parameter ist oder nicht. In allen übrigen Fällen kann man sie ja "nachrüsten", wenn der Bedarf doch noch entsteht, und hat bis dahin einfacheren und klareren Quelltext. Für eine ausführliche Diskussion, wie man die Initialisierung einer Klasse wirklich wasserdicht absichern kann, siehe Item 24 in [Bloch].

Final Attribute

Objekte, deren Zustand sich über ihre Lebensdauer nicht ändert, sind einfacher zu handhaben. Das lässt sich dadurch ausdrücken, dass man Attribute final macht.

Problem. Wenn ein Objekt mit einem Initializing Constructor erzeugt und sinnvoll initialisiert wurde, muss die restliche Implementierung immer noch sicherstellen, dass auch nach jeder Änderung das Objekt in einem sinnvollen Zustand ist. In einem nebenläufigen Kontext muss man ausserdem noch aufpassen, dass Änderungen ausreichend synchronisiert sind. Beides macht eine Klasse zumindest etwas schwieriger zu verstehen.

Das kann man manchmal umgehen, indem man für einen Teil oder alle Felder der Klasse nach der Initialisierung überhaupt keine Änderungen mehr zulässt - wenn man etwas nicht ändern kann, dann kann man auch keine Probleme bei seiner Änderung haben.

Diese Eigenschaft eines Feldes wird dadurch besonders wertvoll, dass andere Programmierer sie kennen und sich darauf verlassen können; wie kann man sie im Quelltext ausdrücken?

Lösung. Deklariere das Feld bzw. die Felder als final. Dadurch stellt der Compiler sicher, dass das Feld genau einmal initialisiert und anschließend nie mehr verändert wird. Das Feld ist also unveränderlich, und diese Eigenschaft ist im Quelltext dokumentiert und wird vom Compiler garantiert.

Das folgende Fragment zeigt als Beispiel eine Person-Klasse.

```
public class Person {
  private final String _name;
  private final Date _geburtstag;

  // Der Konstruktor initialisiert die
  //final Felder
  public Person (String name,
                 final Date geburtstag) {
    _name = name;
    _geburtstag =
      new Date (geburtstag.getTime ());
  }
  ...
}
```

Man sieht auf den ersten Blick, dass sich die Felder nach der Initialisierung nicht mehr ändern können.

Die Java-Sprachspezifikation erlaubt es, ein final Feld im Konstruktor zu initialisieren. Dabei überprüft der Compiler, dass das Feld unter allen denkbaren Umständen in jedem Konstruktor initialisiert wird, dass also z.B. die Initialisierung nicht durch eine Exception übersprungen wird, die später im Konstruktor gefangen wird[1].

Wenn ein Konstruktor einen beliebigen anderen Syntaxfehler enthält, dann neigen die Java-Compiler dazu, zusätzlich als Fehler auszugeben, dass die final-Felder nicht initialisiert wurden, so dass manchmal der eigentliche Fehler von irreführenden Meldungen begraben wird. Da hilft nur, sich nicht vom Compiler einschüchtern zu lassen und nach der eigentlichen Ursache zu suchen.

Bei dem Date-Feld mit dem Geburtstag im Beispiel garantiert das Schlüsselwort final genau wie beim Namen, dass die Referenz nicht geändert werden kann, das hindert aber einen Client nicht daran, sich die Referenz zu merken und später den Wert des Date-Objektes zu ändern. Wenn man das verhindern will, kann man z.B. wie im Beispiel der Person-Klasse eine Kopie des hereingereichten Objektes speichern.

1. Der Compiler führt dabei keine vollständige Analyse der verschiedenen Pfade durch den Konstruktor durch sondern verbietet manche komplizierteren Konstrukte, obwohl sie eigentlich funktionieren würden. Für Details siehe [JLS], §16.8.

Static Factory Method

Oft bieten statische create-Methoden eine bessere Kapselung der Erzeugung von Instanzen als die Konstruktoren.

Problem. Wenn man Objekte erzeugen will, dann ist meist der erste Gedanke, sie direkt mit new zu erzeugen. So auch im folgenden Beispiel, wo verschiedene Arten von Personen erzeugt werden.

```
Person boss = new Manager ("Bill Gates");
Person mitarbeiter =
  new Mitarbeiter ("John Smith");
```

Dieser Code ist leicht zu verstehen und er funktioniert. Wenn Clients aber auf diese Weise Objekte erzeugen, werden sie mit mehreren Implementierungsdetails konfrontiert:

- Es gibt eine Vererbungshierarchie, und Manager und Mitarbeiter sind verschiedene Klassen.
- Es wird tatsächlich jeweils ein neues Objekt erzeugt und z.B. kein Cache verwendet.

Das ist aus der Perspektive des Creation Interface keine gute Lösung. Wie kann man also die Erzeugung der Objekte besser kapseln?

Lösung. Stelle statische create-Methoden bereit, die die Implementierungsdetails der Objekterzeugung kapseln. Die Namen der Methoden sind eine gute Möglichkeit, den Quelltext zusätzlich zu dokumentieren.

Dann kann sich Client-Code, der Objekte erzeugt, auf die Absicht konzentrieren, ohne sich um Implementierungsdetails zu kümmern.

Im Beispiel hat dann die Klasse Person zwei statische Methoden für die beiden Arten, Personen zu erzeugen.

```
public class Person {
  ...
  public static Person createManager
        (String name) {
    return new Manager (name);
  }
```

```
  public static Person createMitarbeiter
        (String name) {
    return new Mitarbeiter (name);
  }
  ...
}
```

Clients, die ein Person-Objekt haben wollen, können sich damit ganz auf ihre Absicht konzentrieren, ohne sich um technische Details zu kümmern.

```
Person boss = Person.
    createManager ("Bill Gates");
Person mitarbeiter = Person.
    createMitarbeiter ("John Smith");
```

Der einzige Nachteil dieses Patterns ist, dass die create-Methoden zusätzlicher Quelltext sind, der gelesen und verstanden werden muss, was bei sehr einfachen Klassen manchmal ins Gewicht fällt. Sehr oft überwiegt aber der Vorteil des klareren und ausdrucksstärkeren Creation Interfaces.

Dieses Pattern ist eine besonders einfache Form von Factory. Eine ganze Reihe von komplizierteren Patterns zur Erzeugung von Objekten in spezielleren Situationen wird z.B. in [GOF] beschrieben.

Checked Raw Creation

Wenn es durch äußere Rahmenbedingungen nötig ist, einen öffentlichen Default-Konstruktor bereitzustellen, der nötige Initialisierungen nicht vornehmen kann, dann schaffe zumindest eine Methode, die überprüft, dass das Objekt sinnvoll initialisiert ist.

Problem. Es gibt Situationen, in denen eine Klasse einen öffentlichen Default-Konstruktor haben muss, ohne dass dieser Konstruktor die Objekte sinnvoll initialisieren könnte. Ein typisches Beispiel sind viele Persistenz-Bibliotheken, die Objekte per Reflection erzeugen und anschließend die Attribute einzeln setzen.

Das führt dazu, dass der Quelltext der Klasse fälschlich suggeriert, man könne einfach durch Aufruf des Default-Konstruktors gültige Instanzen erhalten; das Creation Interface der Klasse führt in die Irre.

Die Lösung kann nicht darin bestehen, den Missstand zu beklagen oder auf die Verwendung der entsprechenden Bibliotheken zu verzichten; was soll man also tun?

Lösung. Stelle eine Methode bereit, die überprüft, ob das Objekt vollständig initialisiert ist.

In einer Situation, wie sie hier beschrieben ist, sind pragmatische Lösungen angebracht. Zunächst einmal sollte man also den Default-Konstruktor (und ggf. öffentliche set-Methoden etc.) mit einem klaren und deutlichen Kommentar versehen, dass er ausschließlich zur Verwendung durch das Framework gedacht ist.

Damit ist aber noch nicht sichergestellt, dass das Framework die Objekte auch vollständig und sinnvoll initialisiert; es könnte ja immerhin sein, dass die Konfiguration des Frameworks einen Fehler hat oder einfach nicht auf dem selben Stand ist wie die Klassen. Man kann die Sicherheit deutlich vergrößern, indem man eine öffentliche checkInit-Methode bereitstellt, die überprüft, ob sich das Objekt in einem gültigen Zustand befindet und andernfalls eine Exception wirft, wie das Beispiel in Listing 2 zeigt.

Diese Methode kann man dann z.B. auf jedem Objekt aufrufen, das vom Framework erzeugt und für fertig initialisiert erklärt wurde.

Wenn Instanzen auch direkt durch Clients erzeugt werden sollen und nicht nur durch das Framework, dann muss man natürlich zusätzlich zum Default-Konstruktor weitere Initializing Constructors oder Static Factory Methods als Creation Interface bereitstellen. Wenn man an dieser Stelle in die Falle der Bequemlichkeit tappt, macht man aus einer kleinen Unschönheit eine subtile Fehlerquelle.

Default Init Value

Wenn viele Clients bei der Erzeugung eines Objektes für einen Parameter den gleichen Wert verwenden, dann schaffe eine Möglichkeit, diesen Wert als Default zu verwenden.

Problem. Oft gibt man einer Klasse zunächst einen einzigen Konstruktor[2], der die volle

```
// hat aus technischen Gründen einen
// Default-Konstruktor, der die
// Initialisierung umgeht. Die
// checkInit-Methode erlaubt
// die nachträgliche Überprüfung der
// Initialisierung
public class Person {
  private String _name;
  private Date _geburtstag;

  /** ACHTUNG: NUR FÜR DAS FRAMEWORK */
  public Person () {}

  public static Person createPerson
    (String name, Date geburtstag)
  {
    Person result = new Person ();
    result._name = name;
    result._geburtstag = geburtstag;
    return result;
  }

  /** prüft, ob das Objekt vollständig
      initialisiert ist, und wirft
      andernfalls eine Exception. */
  public void checkInit () {
    if (_name == null)
      throw new InvalidStateException
          ("Name");
    if (_geburtstag == null)
      throw new InvalidStateException
          ("Geburtstag");
  }

  ...
}
```

Listing 2: Pattern *Checked Raw Creation*

Kontrolle über die Initialisierung eines Objektes erlaubt.

Mit der Zeit zeigt sich dann oft, dass eine Reihe von Clients sich für einen oder mehrere der Parameter nicht interessieren und einen Default-Wert verwenden.

Dieser Default-Wert ist eigentlich eine Eigenschaft der Klasse, die auf diese Weise quer durch den Client-Code verstreut ist. Wie kann man ihn in das Creation Interface der Klasse aufnehmen?

2. Dieses Pattern ist in gleicher Weise auf Konstruktoren und Static Factory Methods anwendbar. Der Einfachheit halber werden nur Konstruktoren erwähnt, die Übertragung ist aber durchweg offensichtlich.

Lösung. Gib der Klasse einen zusätzlichen Konstruktor, der die Default-Werte kennt. Die Parameterliste dieses Konstruktors ist entsprechend kürzer, und er ruft den ursprünglichen Konstruktor mit den Default-Werten auf.

Die Klasse java.util.GregorianCalendar hat z.B. einen Konstruktor, der eine TimeZone und eine Locale bekommt. Da man für diese Parameter aber in der Praxis oft die eingestellten Default-Werte verwendet, gibt es Konstruktoren, die genau das tun (s. Listing 3).

```
// vereinfachter Quelltext aus dem JDK.
// Die beiden Parameter des
// Konstruktors sind optional
public class GregorianCalendar {
  /** vollständiger Konstruktor */
  public GregorianCalendar (TimeZone
      zone, Locale locale)
  {
    ... // Init mit diesen Werten
  }

  /** Default-Wert für Locale */
  public GregorianCalendar (TimeZone
      zone)
  {
    this (zone, Locale.getDefault ());
  }

  /** Default-Wert für TimeZone */
  public GregorianCalendar (Locale
      locale)
  {
    this (TimeZone.getDefault (),
          locale);
  }

  /** Default für TimeZone und Locale*/
  public GregorianCalendar () {
    this (TimeZone.getDefault(),
          Locale.getDefault ());
  }

  ...
}
```

Listing 3: Pattern *Default Init Value*

Auf diese Weise ist die Information, welche Werte als Default verwendet werden, Teil des Creation Interface geworden, und Client-Code wird einfacher und besser lesbar.

Dieses Pattern lässt sich leicht von Konstruktoren auf Static Factory Methods übertragen. Dadurch ergeben sich sogar noch weitere Möglichkeiten:

- Mehrere Defaultwerte. Im Gegensatz zu Konstruktoren sind Static Factory Methods benannt, so dass es problemlos möglich ist, mehrere Default-Werte für den gleichen Parameter zu unterstützen und die Unterscheidung sogar noch durch sprechende Namen zu dokumentieren.
- Flexibilität. In Konstruktoren muss der Aufruf eines anderen Konstruktors das erste Statement sein, oder der Compiler weist ihn ab. Statische Methoden haben diese Einschränkung nicht, was manchmal den Quelltext vereinfacht.
- Caching. Es kommt vor, dass eine einzige Default-Instanz für alle Clients reicht, z.B. wenn sie nicht veränderbar ist. Wenn Clients direkt den Konstruktor aufrufen, wird jedes Mal eine neue Instanz erzeugt, während eine Static Factory Method bei jedem Aufruf das selbe, zwischengespeicherte Objekt liefern können.

Simple Global Instance

Wenn es ausgezeichnete globale Instanzen einer Klasse gibt, bei denen nichts dagegen spricht, ihre Lebensdauer fest an die Klasse zu koppeln, dann lege für sie public static final Felder an, die bei der Deklaration initialisiert werden.

Problem. Die meisten Klassen sind dazu gedacht, beliebig oft instanziiert zu werden, wobei dann alle Instanzen gleichberechtigt nebeneinander stehen. Manchmal gibt es aber ausgezeichnete Instanzen, z.B. weil sie besonders häufig verwendet werden. Im häufigen Extremfall der “Enumerations”[3] sind solche globalen Instanzen sogar die einzigen, Clients sollen nur diese Instanzen verwenden und überhaupt keine neuen Instanzen erzeugen.

3. Enumeration ist eine gängige Bezeichnung für eine Klasse, von der es nur eine feste Menge von Instanzen gibt. Der Name kommt daher, dass solche Klassen das "enum"-Konstrukt aus C und C++ ersetzen. Für Einzelheiten siehe z.B. [Henney] oder [Bloch].

Auf jeden Fall gehört der Zugriff auf solche Instanzen ins Creation Interface.

Lösung. Erzeuge für jede der Instanzen ein public static final Feld, das bei seiner Deklaration initialisiert wird. Dadurch haben Clients den denkbar einfachsten Zugriff auf diese Instanzen, und die Namen der Felder dokumentieren die Instanzen. Ganz nebenbei garantiert der ClassLoader auch noch, dass die Initialisierung thread-sicher erfolgt.

Das folgende Beispiel ist ein vereinfachter Auszug aus der Klasse java.util.Locale, in der einige häufige Locales als fertige Instanzen bereitgestellt werden.

```
public class Locale {
  static public final Locale GERMANY =
    new Locale ("de","DE","");
  static public final Locale UK =
    new Locale ("en","GB","");
  static public final Locale US =
    new Locale ("en","US","");
  ... // eine Reihe weiterer Locales

  public Locale (String language,
                 String country,
                 String variant) {
    ... // Objekt mit diesen Parameter
        // initialisieren
  }

  ...
}
```

In diesem Falle steht es Clients frei, ob sie eine der fertigen Instanzen verwenden wollen oder eine neue erzeugen wollen. Wenn das nicht gewünscht ist, dann lässt es sich einfach dadurch verhindern, dass man alle Konstruktoren private macht.

Die globalen Instanzen stehen allen Threads über die gesamte Laufzeit des Programms zur Verfügung, so dass alle Änderungen an den Objekten schwer abzuschätzende Folgen haben. Deshalb sind Simple Global Instances sehr häufig unveränderlich.

Die static final Felder müssen direkt bei ihrer Deklaration initialisiert werden. Am einfachsten ist das, wenn die Klasse einen Initializing Constructor hat, der die Objekte vollständig initialisiert. Außerdem darf dieser Konstruktor

keine Checked Exception werfen, sonst ist der Quelltext nicht compilierbar.

Ansonsten kann man die Instanzen mit einem statischen Initialisierungs-Block "nach-initialisieren", der z.B. mögliche Exceptions fängt. Dazu ist es manchmal nötig, einen privaten Default-Konstruktor einzuführen, mit dem man die Felder initialisiert, und eine private Init-Methode, die anschließend aus einem statischen Initialisierungs-Block aufgerufen wird.

Wenn man wie hier beschrieben die Instanzen direkt bei der Deklaration initialisiert, dann gibt man damit die Kontrolle über den Erzeugungszeitpunkt der Instanzen aus der Hand; sie werden einfach dann instanziiert, wenn die Klasse geladen wird.

Das ist häufig kein Problem. Wenn die Initialisierung aber z.B. auf andere globale Funktionalität wie etwa eine Logging-Komponente zugreift, die zuerst initialisiert werden muss, dann kann diese Vorgehensweise unübersichtlich und fehleranfällig werden. In solchen Fällen bietet Managed Global Instance umfangreichere Kontrolle über die Erzeugung.

Managed Global Instance

Wenn eine Klasse ausgezeichnete globale Instanzen hat, deren Erzeugung nicht unmittelbar an das Laden der Klasse gekoppelt werden soll, dann verstecke die Instanzen hinter einer Static Factory Method.

Problem. Manchmal gehören zu Klassen globale Instanzen, von denen man im Creation Interface nichts weiter als ihre Existenz veröffentlichen möchte und andere Einzelheiten wie den Zeitpunkt ihrer Erzeugung oder die Frage, ob sie über die gesamte Laufzeit der Anwendung gleich bleiben, lieber kapselt.

Ein möglicher Grund ist, dass die Instanz sehr speicherintensiv ist und nur dann erzeugt werden soll, wenn sie tatsächlich benötigt wird; oder die Erzeugung ist zeitintensiv und soll nebenläufig im Hintergrund erfolgen.

Ein weiterer möglicher Grund ist, dass zur Erzeugung der Instanz andere globale Ressourcen benötigt werden, die vorher initialisiert werden müssen. Oder die Instanz ist zwar global, kann aber von außen initialisiert werden wie z.B. die aktuell in der Nutzerschnittstelle verwendete Sprache.

Allen diesen Szenarien ist gemeinsam, dass es globale Instanzen gibt, dass man aber mehr Kontrolle über ihre Erzeugung braucht als bei Simple Global Instances.

Lösung. Verstecke die Erzeugung der Instanzen hinter einer Static Factory Method. Wenn ein statisches Feld zum Speichern der Referenz verwendet wird, dann mache es private.

Dadurch bleibt der - eventuell komplizierte - Mechanismus zum Erzeugen oder die Möglichkeit, die Instanz während der Laufzeit des Programms auszutauschen, vor Clients versteckt, und Änderungen daran betreffen Clients weit weniger.

Auch die Verwendung von Pooling und andere Maßnahmen, die die tatsächliche Zahl der Instanzen verändern, lassen sich mit weit weniger Aufwand nachträglich durchführen.

Ein Beispiel sind Factories für Geschäftsobjekte, bei denen man häufig eine gewisse Kontrolle über den Zeitpunkt der Erzeugung haben möchte. Listing 4 zeigt als Beispiel eine ver-

```
// Eine Factory, die die einzige
// Instanz als Managed Global Instance
// verwaltet
public class VertragsManagerFactory {
  private static VertragsManager
      _instance = null;

  public static synchronized
      VertragsManager getInstance ()
  {
    if  (_instance == null) {
      _instance=new VertragsManager ();
    }
    return _instance;
  }
}
```

Listing 4: Pattern *Managed Global Instance*

einfachte Factory für ein VertragsManager-Objekt, die die VertragsManager-Instanz erst beim ersten Zugriff erzeugt.

Die statische Zugriffsmethode ist synchronized, weil sonst ein zweiter Thread, der sie während der Initialisierung aufruft, einen zweiten VertragsManager erzeugen oder sogar einen erst teilweise initialisierten VertragsManager

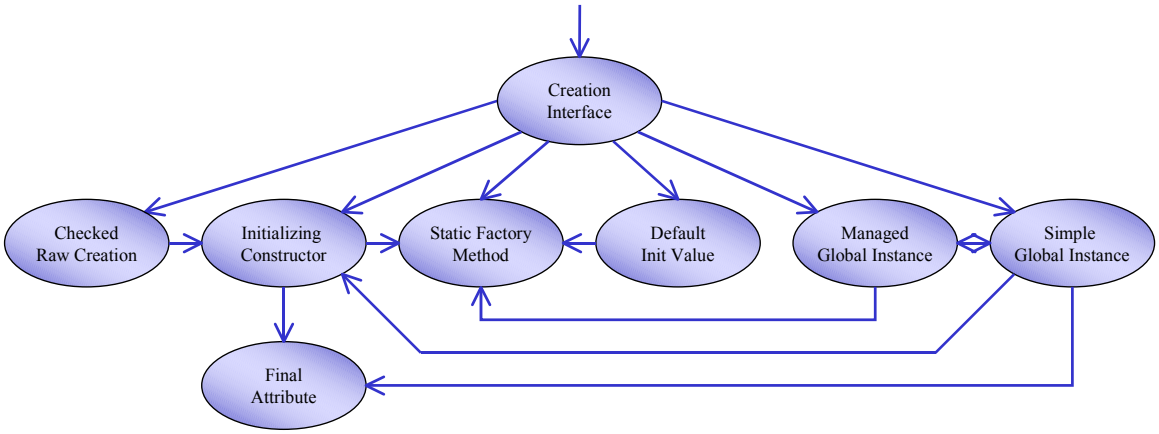

Abb. 1: Die Patterns und ihre Beziehungen. *Creation Inteface* ist der Einstiegspunkt in die Pattern-Sprache, und von dort aus gelangt man entlang der Pfeile zu den anderen Patterns.

zu Gesicht bekommen könnte[4]. Immer wenn die Erzeugung von Instanzen in einem nebenläufigen Kontext erfolgt, ist es nötig, die Synchronisation sicherzustellen. Nur wenn die Instanziierung beim Laden der Klasse ausgeführt wird, sorgt der ClassLoader für die Synchronisation.

Managed Global Instance bietet weitergehende Kontrolle über die Erzeugung und Verwaltung der globalen Instanzen als Simple Global Instance, allerdings zum Preis eines höheren Implementierungsaufwandes und einer schwieriger zu verstehenden Klasse.

Im Gegensatz z.B. zu C++ berücksichtigt Java Abhängigkeiten bei der Reihenfolge der Initialisierung statischer Variablen, so dass Simple Global Instance für viele Fälle ausreicht.

Die Pattern-Sprache

Jedes der acht Patterns, die in diesem Artikel vorgestellt wurden, löst ein Problem und ist für sich genommen nützlich. Zusätzlichen Wert bekommen sie aber noch dadurch, dass sie als Pattern-Sprache in Beziehung zueinander stehen.

Abbildung 1 gibt einen Überblick über diese Pattern-Sprache. Jedes Oval steht für ein Pattern, und jeder der Pfeile bedeutet eine "Nachfolger"-Beziehung: Wenn man ein Pattern verwendet, dann zeigen die Pfeile auf andere Patterns, die man vielleicht in der gleichen Situation zusätzlich gut gebrauchen kann. Der Einstiegspunkt in die Pattern-Sprache ist Creation Interface, das den allgemeinen Rahmen absteckt, in dem die anderen Patterns leben.

Durch diese Art der Darstellung und überhaupt das Konzept der Pattern-Sprachen kann man ein ganzes Themengebiet - hier die Erzeugung von Objekten - durch eine Reihe von Patterns abdecken, ohne dass die einzelnen Patterns überfrachtet sind. Das Diagramm hilft dabei, einen Überblick zu bekommen, und dient als Gedächtnisstütze.

Fazit

Das Erzeugen von Objekten erfordert genau wie die eigentliche Funktionalität aktive Beachtung und Planung. Klassen sollten ihren Clients ein gut entworfenes Creation Interface zur Verfügung stellen, das das Erzeugen und Verwalten von Instanzen einfach und wenig fehlerträchtig macht. Genau dabei hilft die hier vorgestellte Pattern-Sprache.

4. Das ist auf den ersten Blick wenig intuitiv und hängt mit dem aktuellen Speichermodell von Java zusammen. Versuche, den kleinen Performance-Overhead durch Double Checked Locking wegzuoptimieren, scheitern in Java [DoubleCheckedLocking]. Für eine andere Möglichkeit zur Performance-Optimierung siehe [Bloch].

Literatur und Links

[Bloch] Joshua Bloch, Effective Java, Addison-Wesley 2001

[DoubleChecking] http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[GOF] Gamma, Helm, Johnson, Vlissides, Design Patterns, Addison-Wesley 1995

[Henney] Kevlin Henney, Patterns of Value, Java Report Feb. 2000

[JLS] Gosling, Joy, Steele, Bracha, The Java Language Specification, 2nd ed., Addison-Wesley 2000

[Meyer] Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall 1997