Entwurfsmuster für die Praxis

2. Auflage
Aktuell zu PHP 5.3

PHP Design Patterns

O'REILLY®

Stephan Schmidt

Inhalt

KAPITEL 4
Erzeugungsmuster

Nachdem Sie nun wissen, welche Möglichkeiten Ihnen PHP für die objektorientierte Programmierung bietet und welche Regeln Sie befolgen sollten, wenn Sie Software entwickeln, erfahren Sie in den folgenden Kapiteln, wie einzelne Design Patterns im Detail angewandt werden. Den Anfang machen dabei die erzeugenden Entwurfsmuster.

Erzeugungsmuster werden verwendet, um Objekte zu konstruieren. »Dazu bietet PHP doch den new-Operator«, mögen Sie jetzt sagen. »Wozu sollen also Entwurfsmuster gut sein, die etwas erledigen, was PHP bereits mit sich bringt?« Damit haben Sie teilweise auch recht: Die eigentliche Instanziierung der Objekte wird immer mithilfe des `new`-Operators erfolgen, eine andere Möglichkeit haben Sie in PHP nicht. In den vorherigen Kapiteln haben Sie die Regel kennengelernt, dass Sie immer gegen eine Schnittstelle statt einer konkreten Implementierung programmieren sollten. Jedoch haben Sie immer an irgendeiner Stelle eine konkrete Implementierung der Schnittstelle mit dem `new`-Operator instanziiert und somit wieder gegen diese Regel verstoßen (zum Glück hat das nur noch niemand gemerkt).

In diesem Kapitel werden Sie nun erfahren, wie Sie die Verwendung des `new`-Operators und somit auch die Abhängigkeit von einer konkreten Implementierung aus Ihrem Code verbannen. Durch den Einsatz des *Factory-Method-Patterns* werden Sie ein Objekt erzeugen, ohne dass Sie den Namen der Klasse kennen, und mithilfe des *Abstract-Factory-Patterns* sogar eine ganze Familie von Objekten instanziieren, ohne dabei Klassennamen anzugeben. Mithilfe des *Prototype-Patterns* verwenden Sie bestehende Objekte als Vorlage für die Erzeugung neuer Instanzen. Zu Beginn dieses Kapitels werden Sie jedoch erst das *Singleton-Pattern* anwenden, mit dem Sie die Anzahl der möglichen Instanzen einer Klasse beschränken können.

Haben Sie also geglaubt, Sie wüssten durch die Verwendung des `new`-Operators schon alles, was es zur Erzeugung von Objekten zu wissen gibt? Wenn dem so ist, sollten Sie die folgenden Seiten aufmerksam studieren und werden am Ende des Kapitels staunen, wie komfortabel die Erzeugung von Objekten doch sein kann.

Das Singleton-Pattern

PHP 5 erlaubt Ihnen, zu kontrollieren, wer auf welche Eigenschaften oder Methoden Ihrer Klassen und Objekte Zugriff erhält, indem Sie die Schlüsselwörter `public`, `private` oder `protected` verwenden. PHP ermöglicht Ihnen sogar zu beschränken, was beim Ableiten von Ihren Klassen überschrieben werden darf oder sogar muss. Eines ermöglicht Ihnen PHP jedoch nicht: Sie können nicht einschränken, wer wie viele Instanzen Ihrer Klasse erzeugen darf. Und genau hier kommt das *Singleton-Pattern* ins Spiel.

Das Singleton-Pattern ist eines der einfachsten Design Patterns, aber dennoch eines der am meisten genutzten. Vielleicht haben auch Sie dieses Muster schon eingesetzt, ohne zu wissen, dass es sich dabei um ein Entwurfsmuster handelt.

Motivation

In der Beispielanwendung des letzten Kapitels haben Sie den Debugging-Code aus der `RentalCompany`-Klasse herausgelöst und ihn in neuen Klassen gekapselt. Dadurch kann der Debugging-Code in Zusammenarbeit mit verschiedenen Klassen verwendet werden. Bei Erzeugung einer `RentalCompany`-Instanz übergeben Sie einen Debugger einfach im Konstruktor:

```
$debugger = new DebuggerEcho();
$rentalCompany = new RentalCompany($debugger);
```

Wenn Sie nun auch noch einen Debugger für Kunden und Autos verwenden wollten, könnte der Code zum Beispiel so aussehen:

```
$debugger2 = new DebuggerEcho();
$stephan = new Customer($debugger2, 1, 'Stephan Schmidt');

$debugger3 = new DebuggerEcho();
$bmw = new Car($debugger3, 'BMW', 'blau');
```

Sie haben hier also schon drei Instanzen derselben Debugger-Klasse erzeugt und damit ca. dreimal so viel Speicher verbraucht. Die Autovermietung wird sicher mehr als ein Auto vermieten und sicher auch mehr als einen Kunden bedienen. Der Speicherverbrauch wird also mit der Anzahl der vermieteten Autos und der Anzahl an Kunden stetig weiter ansteigen. Wenn Sie sich den Debugger genauer ansehen, fällt Ihnen sehr schnell auf, dass es eigentlich nicht nötig ist, für jeden Kunden und jedes Auto einen eigenen Debugger zu verwenden, schließlich wird dieser nur benutzt, um die übergebene Debug-Meldung auszugeben. Er weiß nichts über das Objekt, das ihn verwendet.

Es wäre also ideal, wenn Sie für jedes Auto und jeden Kunden immer denselben Debugger verwenden könnten und somit eine Menge Speicher sparen und Instanziierungen vermeiden. Wenn die Anwendung allerdings wächst, werden die Debugger an den verschiedensten Stellen instanziiert, sodass Sie nie sicher sein können, welches das erstes Debugger-Objekt ist, das Sie erzeugt haben.

Sie benötigen also einen zentralen Zugriffspunkt auf unseren Debugger.

Zweck des Patterns

Diesen zentralen Zugriffspunkt werden Sie im Folgenden mithilfe des Singleton-Patterns bereitsstellen:

Das Singleton-Pattern sichert ab, dass von einer Klasse nur eine Instanz existieren kann, und stellt einen globalen Zugriffspunkt auf diese Instanz bereit.

Um dies nun auf das Debugger-Problem anzuwenden, müssen Sie die folgenden Schritte befolgen:

1. Bereitstellen des zentralen Zugriffspunkts auf die Instanz der Debugger-Klasse.
2. Dieser zentrale Zugriffspunkt muss immer Zugriff auf dasselbe Objekt bieten, egal wie oft er verwendet wird.
3. Verhindern, dass auf irgendeinem anderen Weg es ermöglicht wird, eine zweite Instanz der Klasse zu erstellen.

Implementierung

Wenn Sie den Debugger direkt an der Stelle im Quellcode erzeugen, an der Sie ihn verwenden möchten, haben Sie dort keine Möglichkeit festzustellen, ob es bereits eine Instanz der Klasse gibt. Um die Instanziierung des Debuggers zentral an einer Stelle vorzunehmen, lagern Sie diesen Code in eine eigene Methode aus. Da für die Instanziierung des Objekts keine weiteren Informationen nötig sind, verwenden Sie dafür eine statische Methode (also eine Methode, die direkt auf der Klasse aufgerufen werden kann, ohne eine Instanz erzeugen zu müssen):

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerEcho implements Debugger {

    public static function getInstance() {
        $debugger = new DebuggerEcho();
        return $debugger;
    }

    public function debug($message) {
        echo "{$message}\n";
    }
}
```

In der statischen Methode `getInstance()` erstellen Sie also eine neue Instanz der Klasse `DebuggerEcho` und geben diese anschließend zurück. Statt im Code der Klasse mithilfe des `new`-Operators einen Debugger zu erzeugen, können Sie dafür nun die neue Methode verwenden:

```
use de\phpdesignpatterns\util\debug\DebuggerEcho;

$debuggerObi = DebuggerEcho::getInstance();
```

```
$debuggerObi->debug('Nutze die Macht, Luke!');

$debuggerVader = DebuggerEcho::getInstance();
$debuggerVader->debug('Luke, ich bin dein Vater!');

if ($debuggerObi === $debuggerVader) {
    echo "Die beiden Debugger sind dasselbe Objekt.\n";
} else {
    echo "Die beiden Debugger sind *NICHT* dasselbe Objekt.\n";
}
```

Sie erzeugen nun das Objekt zwar nicht mehr an der Stelle im Quellcode, an der es benötigt wird, viel gewonnen haben Sie dennoch nicht, denn bei jedem Aufruf der Methode wird ein neuer Debugger erstellt. Dies zeigt Ihnen auch die Ausgabe des Beispiels:

```
Nutze die Macht, Luke!
Luke, ich bin dein Vater!
Die beiden Debugger sind *NICHT* dasselbe Objekt.
```

Die Methode sollte also etwas mehr Logik mitbringen und beim Aufruf die folgenden Schritte durchlaufen:

1. Beim Aufruf überprüfen, ob bereits ein Debugger erstellt wurde.
2. Falls nicht, einen neuen Debugger mithilfe des `new`-Operators erstellen und diesen speichern.
3. Den gespeicherten Debugger zurückgeben.

Um dasselbe Objekt mehrfach verwenden zu können, müssen Sie es also innerhalb der `getInstance()`-Methode speichern. Würden Sie die `getInstance()`-Methode auf einem Objekt aufrufen, könnten Sie eine Objekteigenschaft dazu verwenden. Da Sie allerdings die Methode statisch aufrufen, funktioniert dies nicht. Stattdessen müssen Sie eine statische Klasseneigenschaft verwenden, um die erstellte Instanz zu speichern:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerEcho implements Debugger {

    private static $instance = null;

    public static function getInstance() {
        if (self::$instance == null) {
            self::$instance = new DebuggerEcho();
        }
        return self::$instance;
    }

    public function debug($message) {
        echo "{$message}\n";
    }
}
```

Sie haben nun die statische Klasseneigenschaft `$instance` hinzugefügt und überprüfen beim Aufruf von `getInstance()`, ob diese Eigenschaft bereits einen Debugger enthält. Wenn nicht, erzeugen Sie einen neuen Debugger und weisen diesen der Klasseneigenschaft zu. Danach geben Sie den Debugger aus der Methode zurück.

Führen Sie mit diesem veränderten Code das Beispiel erneut aus, erhalten Sie die gewünschte Ausgabe:

```
Nutze die Macht, Luke!
Luke, ich bin dein Vater!
Die beiden Debugger sind dasselbe Objekt.
```

Egal wie oft Sie die `getInstance()`-Methode auch aufrufen, die Applikation verwendet immer dasselbe Objekt und spart somit Ressourcen ein.

Fallstricke

Es wäre zu einfach gewesen, wenn Sie damit schon Ihr erstes Design Pattern angewandt hätten. Das Singleton bringt leider auch noch einige Fallstricke mit; was passiert zum Beispiel, wenn einer Ihrer Teamkollegen nicht weiß, dass er die `getInstance()`-Methode verwenden muss, um einen Debugger zu erhalten? Er würde dann wie gewohnt einen neuen Debugger erzeugen:

```
use de\phpdesignpatterns\util\debug\DebuggerEcho;

// Ihr Debugger
$debuggerObi = DebuggerEcho::getInstance();
$debuggerObi->debug('Nutze die Macht, Luke!');

// der Debugger Ihres Kollegen
$debuggerVader = new DebuggerEcho();
$debuggerVader->debug('Luke, ich bin dein Vater!');

if ($debuggerObi === $debuggerVader) {
    echo "Die beiden Debugger sind dasselbe Objekt.\n";
} else {
    echo "Die beiden Debugger sind *NICHT* dasselbe Objekt.\n";
}
```

Nun sind die beiden Debugger natürlich wieder unterschiedliche Objekte, und Sie (oder, besser gesagt, Ihr Kollege) verschwenden erneut Speicher. Sie müssen Ihren Teamkollegen also verbieten, selbst neue Instanzen des Debuggers zu erzeugen, und sie somit zwingen, immer die `getInstance()`-Methode zu verwenden. Die Lösung dazu ist ganz simpel, Sie unterbinden einfach die Nutzung des Konstruktors von außerhalb der Klasse, indem Sie ihn als `protected` deklarieren:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerEcho implements Debugger {
    ... statische Eigenschaft und getInstance()-Methode ...
```

```
    protected function __construct() {}

    public function debug($message) {
        echo "{$message}\n";
    }
}
```

Versucht jetzt einer Ihrer Kollegen, eine neue Instanz des Debuggers zu erzeugen, reagiert PHP mit einem Fehler:

```
Fatal error: Call to protected de\phpdesignpatterns\util\debug\DebuggerEcho::__
construct() from context '' in ch3/singleton/debuggerechoconstructor.php on line 26
```

Ihr Kollege wird nun stattdessen wie gewünscht die `getInstance()`-Methode verwenden müssen, um einen Debugger zu erhalten.

Findige Kollegen, die trotzdem unbedingt eine neue Instanz erzeugen möchten, könnten vielleicht noch auf die Idee kommen, den Debugger, den sie von der `getInstance()`-Methode erhalten haben, zu klonen:

```
use de\phpdesignpatterns\util\debug\DebuggerEcho;

$debuggerObi = DebuggerEcho::getInstance();
$debuggerObi->debug('Nutze die Macht, Luke!');

$debuggerVader = clone $debuggerObi;
$debuggerVader->debug('Luke, ich bin dein Vater!');

if ($debuggerObi === $debuggerVader) {
    echo "Die beiden Debugger sind dasselbe Objekt.\n";
} else {
    echo "Die beiden Debugger sind *NICHT* dasselbe Objekt.\n";
}
```

Wenn Sie dieses Beispiel ausführen, werden Sie leider feststellen, dass Sie wieder zwei verschiedene Instanzen der Klasse verwenden. Auch wenn Ihnen niemand solche Kollegen wünscht, kann man mit der aktuellen Lösung nie sicher sein, dass wirklich nur eine Instanz der Klasse existiert; somit haben Sie die dritte Anforderung noch nicht zu 100% erfüllt.

Zum Glück erlaubt Ihnen PHP auch, das Klonverhalten für eine Klasse zu verändern, indem Sie eine Methode mit dem Namen `__clone()` implementieren. Um das Klonen zu verhindern, verbieten Sie einfach den Aufruf der `__clone()`-Methode von außerhalb der Klasse:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerEcho implements Debugger {
    ... statische Eigenschaft und getInstance()-Methode ...

    protected function __construct() {
    }
```

```
    private function __clone() {}

    public function debug($message) {
        echo "{$message}\n";
    }
}
```

Beim erneuten Versuch, ein Debugger-Objekt zu klonen, erhält der Entwickler nun auch eine Fehlermeldung, die ihn darauf hinweist, dass das Klonen des Objekts verboten ist:

```
Fatal error: Call to private de\phpdesignpatterns\util\debug\DebuggerEcho::__clone()
from context '' in ch3\singleton\debuggerechoclone.php on line 28
```

Sie haben nun also sichergestellt, dass von der Klasse `DebuggerEcho` immer nur ein Objekt existieren kann, und somit Ihr erstes Singleton implementiert.

Sie mögen sich vielleicht wundern, warum der Konstruktor als `protected` und die `__clone()`-Methode als `private` deklariert wurde. Der Grund dafür liegt in den Möglichkeiten, die Ihnen Vererbung bietet. Im Fall des Konstruktors möchten Sie es Klassen, die vom Debugger ableiten, erlauben, den Konstruktor zu überschreiben. Der Konstruktor darf zwar nicht von außerhalb einer Klasse verwendet werden, aber dennoch ist es möglich, den Konstruktur für seine üblichen Aufgaben zu nutzen.

Im Fall der `__clone()`-Methode ist dies nicht nötig, hier soll einfach nur die Nutzung unterbunden werden.

Definition des Patterns

Das Singleton-Pattern sichert ab, dass von einer Klasse nur eine Instanz existieren kann, und stellt einen globalen Zugriffspunkt auf diese Instanz bereit.

Um dieses Pattern in PHP zu implementieren, sind immer die folgenden vier Schritte nötig:

1. Deklarieren Sie eine statische Klasseneigenschaft, die das Exemplar der Klasse speichert
2. Implementieren Sie eine statische Methode, die das gespeicherte Exemplar zurückgibt und gegebenenfalls dieses Exemplar erstellt, falls es noch nicht vorhanden ist.
3. Sichern Sie ab, dass keine weiteren Instanzen durch Verwendung des `new`-Operators möglich sind, indem Sie den Konstruktor als `protected` deklarieren.
4. Sichern Sie ab, dass das Exemplar der Klasse nicht geklont werden kann, indem Sie die `__clone()`-Methode als `private` deklarieren.

Abbildung 4-1 zeigt ein UML-Diagramm des Patterns. Wenn Sie diese vier einfachen Regeln befolgen, steht weiteren Implementierungen des Singleton-Patterns nichts mehr im Weg.

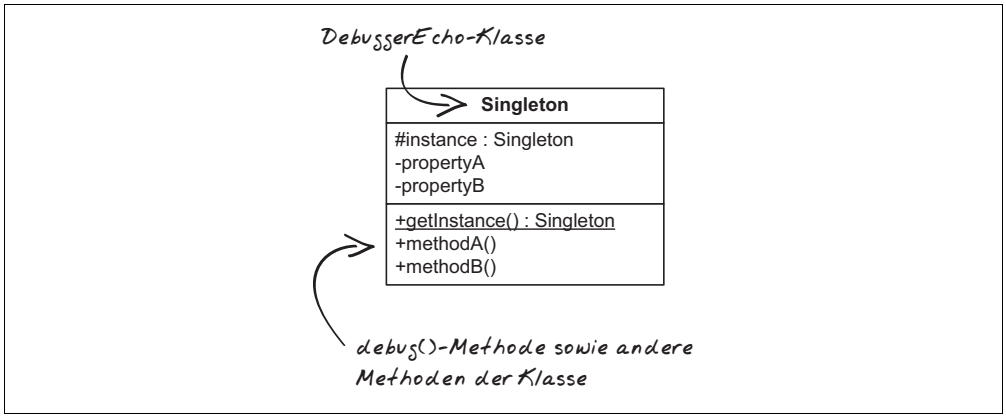

Abbildung 4-1: UML-Diagramm der Singleton-Implementierung

Konsequenzen

Wenn Sie das Singleton-Pattern anwenden, hat das für Ihre Applikation die folgenden Konsequenzen:

1. Sie können sehr genau kontrollieren, wie auf Ihre Klasse zugegriffen wird. Durch den zentralen Zugriffspunkt über die `getInstance()`-Methode können Sie beliebige Kontrollmechanismen einfügen.
2. Es gibt von Ihrer Klasse immer nur eine Instanz. Sollten Sie plötzlich eine zweite Instanz der Klasse benötigen, lässt das Singleton-Muster dies nicht zu.

   Allerdings gibt es Modifikationen des Singleton-Patterns, bei denen auch mehr als ein Objekt erlaubt wird.
3. Das Singleton-Pattern ermöglicht Ihnen, die Anzahl der globalen Variablen zu reduzieren oder diese komplett aus Ihrem Quellcode zu verbannen. Statt globale Instanzen der Klassen in Variablen im globalen Namensraum zu speichern, greifen Sie über statische Methoden auf diese Instanzen zu und halten somit den globalen Namensraum frei.

Beim Singleton handelt es sich um eines der einfachsten Design Patterns. Dies mag der Grund dafür sein, dass es sehr häufig auch in Situationen eingesetzt wird, in denen es eigentlich fehl am Platze ist. Denken Sie vor Einsatz dieses Design Patterns darüber nach, ob Sie wirklich nur eine Instanz der Klasse erlauben wollen oder ob auch mehr Instanzen der Klasse parallel nutzbar sind und Sie eigentlich nur einen globalen Zugriffspunkt auf ein Objekt benötigen. In letzterem Fall könnten Sie eventuell auch auf das *Registry-Pattern* zurückgreifen.

Weitere Anwendungen

Neben dem Debugger gibt es noch weitere Anwendungsmöglichkeiten für das Singleton-Pattern. Häufig wird das Singleton zum Beispiel eingesetzt, um einen zentralen Zugriffspunkt auf die Konfiguration einer Applikation bereitzustellen. Wenn nur ein Objekt existiert, das die Konfiguration speichert, haben Sie dadurch gleich zwei Vorteile:

1. Wenn ein Konfigurationswert einer Komponente verändert wird, gilt diese Änderung sofort für alle Komponenten.
2. Konfigurationsdateien müssen nur einmal eingelesen werden, was sich ressourcenschonend auf die Anwendung auswirkt.

Genauso werden Singletons häufig eingesetzt, wenn eine Applikation auf eine externe Ressource wie zum Beispiel eine Datenbank zugreift. Durch Verwendung eines zentralen Objekts, das sich um den Zugriff auf die Datenbank kümmert, muss nur eine Verbindung zur Datenbank aufgemacht werden, die sich die einzelnen Komponenten teilen können.

Variation des Singleton-Patterns

Vielleicht haben Sie sich schon gefragt, wie man das Singleton für Objekte umsetzen kann, die von außen parametrisierbar sein sollen. Angenommen, Sie möchten das Debugging nun wieder auf Logdateien umstellen, aber nicht alle Meldungen in eine Datei schreiben, sondern unterschiedliche Dateien für die unterschiedlichen Komponenten verwenden.

Ein Debbuger für Logfiles könnte zum Beispiel so aussehen:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerLog implements Debugger {
    protected $logfile = null;

    public function __construct($logfile) {
        $this->logfile = $logfile;
    }

    public function debug($message) {
        error_log("{$message}\n", 3, $this->logfile);
    }
}
```

Beim Instanziieren des Debuggers wird also dem Konstruktor einfach der Name der Datei übergeben, in die die Meldungen geschrieben werden sollen. Somit ist es möglich, die Meldungen je nach Komponente in eine andere Datei zu schreiben:

```
use de\phpdesignpatterns\util\debug\DebuggerLog;

$debuggerObi = new DebuggerLog('./obi.log');
$debuggerObi->debug('Nutze die Macht, Luke!');
```

```
$debuggerVader = new DebuggerLog('./vader.log');
$debuggerVader->debug('Luke, ich bin dein Vater!');
```

Leider können Sie hier das Singleton-Pattern nicht mehr anwenden, Sie wollen mehr als eine Instanz derselben Klasse erlauben. Allerdings wollen Sie dies nur erlauben, solange die verschiedenen Instanzen in verschiedene Dateien schreiben. Wird in dasselbe Log geschrieben, soll auch dieselbe Instanz verwendet werden. Durch eine leichte Modifikation des Singleton ist auch dies möglich. Statt einer globalen Instanz müssen Sie eine Instanz der Klasse pro verwendeter Logdatei in der statischen Eigenschaft speichern. Diese wird dazu durch ein Array ersetzt:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerLog implements Debugger {

    protected $logfile = null;
    private static $instances = array();

    public static function getInstance($logfile) {
        if (!isset(self::$instances[$logfile])) {
            self::$instances[$logfile] = new DebuggerLog($logfile);
        }
        return self::$instances[$logfile];
    }

    protected function __construct($logfile) {
        $this->logfile = $logfile;
    }

    private function __clone() {}

    public function debug($message) {
        error_log("{$message}\n", 3, $this->logfile);
    }
}
```

Jetzt müssen Sie nur noch den Namen der Logdatei an die `getInstance()`-Methode übergeben. Der folgende Code erzeugt also lediglich zwei Instanzen der `DebuggerLog`-Klasse, obwohl drei angefordert werden. Der erste und der letzte Methodenaufruf liefern das gleiche Objekt zurück:

```
use de\phpdesignpatterns\util\debug\DebuggerLog;

$debuggerObi = DebuggerLog::getInstance('./jedi.log');
$debuggerVader = DebuggerLog::getInstance('./sith.log');
$debuggerLuke = DebuggerLog::getInstance('./jedi.log');
```

Sie haben nun zwar kein Singleton-Pattern im klassischen Sinn mehr implementiert, aber in der Realität werden Sie sehr häufig diese kleine Variation des Musters antreffen.

Das Factory-Method-Pattern

In den bisherigen Beispielen haben Sie ein neues Auto oder Flugzeug immer durch Verwendung des new-Operators erstellt. Mal ganz davon abgesehen, dass dies in der Realität leider nicht so einfach funktioniert, binden Sie damit Ihren Code an eine konkrete Implementierung, was Sie eigentlich vermeiden sollten. Denn damit erschweren Sie die Einbindung neuer Klassen, wenn Sie z.B. neue Fahrzeugtypen hinzufügen möchten. Wenn Sie Objekte mit dem new-Operator direkt dort erzeugen, wo Sie sie benötigen, haben Sie Instanziierungen über den gesamten Quellcode der Applikation verteilt. Bei Änderungen an der Instanziierung durch Hinzufügen von Argumenten zum Konstruktor oder Einfügen neuer Klassen muss jede der Dateien angepasst werden, die Objekte erzeugt. Mit der Anzahl der Stellen, die geändert werden müssen, steigt natürlich auch die Anzahl der Fehler, die dabei gemacht werden können.

Besser wäre es, wenn Sie eine Klasse oder ein Objekt hätten, das die Fahrzeuge für Sie erzeugt, sozusagen eine *Auto-Fabrik*. Genau für dieses Problem ist das *Factory-Method-Pattern*, auch *Fabrikmethode* genannt, zuständig, das Sie im Folgenden anwenden werden.

Motivation

Um die Integration neuer Fahrzeugtypen zu erleichtern, möchten Sie also die Applikation um Autohersteller erweitern. Diese könnten alles über die Herstellung eines neuen Autos wissen und dabei gleichzeitg Aufgaben durchführen, die Sie immer durchführen möchten, bevor ein neues Auto vom Band rollt. Diese Aufgaben umfassen zum Beispiel:

- Erzeugen des neuen Objekts und Setzen des Herstellers und der Farbe.
- Starten des Motors und Bewegen des Autos um einen Kilometer, um zu überprüfen, ob alles funktioniert.

Allerdings wird Ihnen eine Autofabrik nicht genügen, schließlich vermieten Sie nicht nur Limousinen, sondern auch Cabrios und eben Flugzeuge. Einzelne Hersteller spezialisieren sich hierbei wie im richtigen Leben auf einen Fahrzeugtyp, daher benütigen Sie für jeden dieser Fahrzeugtypen eine eigene Fabrik. Allerdings unterscheiden sich diese nur bei der eigentlichen Herstellung des Fahrzeugs, alle sollen nach dem eigentlichen Herstellungsprozess den Wagen einmal probeweise anlassen.

Zweck des Patterns

Diese Autohersteller werden Sie im Folgenden mithilfe einer *Fabrikmethode* implementieren. Der Zweck dieses Entwurfsmusters deckt sich perfekt mit der Aufgabenstellung:

Das Fabrikmethoden-Muster definiert eine Schnittstelle zur Erzeugung von Objekten. Es verlagert aber die eigentliche Instanziierung in Unterklassen; es lässt die Unterklassen entscheiden, welche konkreten Implementierungen verwendet werden.

Für den konkreten Anwendungsfall bedeutet dies nun:

1. Sie müssen eine Herstellerklasse implementieren, um eine Schnittstelle zu definieren.
2. In die Basisklasse fügen Sie Code ein, der für alle Hersteller gleich ist.
3. Danach implementieren Sie beliebige Unterklassen des Herstellers, um verschiedenen Fahrzeugtypen herstellen zu können.

Implementierung

Sie beginnen die Umsetzung der Hersteller, indem Sie eine abstrakte Klasse implementieren, die als Basis für die konkreten Autohersteller fungieren wird.

Diese Klasse `AbstractManufacturer` soll es Ihnen später ermöglichen, beliebige Fahrzeuge an die Autovermietung zu verkaufen. Wie im richtigen Leben werden verschiedene Hersteller auch verschiedene Autos produzieren, und so werden Sie am Ende der Implementierung eine BMW-Fabrik, aber auch eine Peugeot- oder Mercedes-Fabrik umsetzen können. Jeder der Hersteller braucht also einen Namen, daher benötigt die Basiklasse eine Eigenschaft, um den Namen des Herstellers aufzunehmen. Um den entsprechenden Wert dieser Eigenschaft setzen zu können, nutzen Sie den Konstruktor der Klasse:

```
namespace de\phpdesignpatterns\manufacturers;

class AbstractManufacturer {
    protected $name;

    public function __construct($name) {
        $this->name = $name;
    }
}
```

Was der Klasse jetzt noch fehlt, ist eine Methode, um ein Auto zu verkaufen. Hierzu implementieren Sie die Methode `sellVehicle()`. Dieser Methode möchten Sie später die Farbe des zu verkaufenden Autos übergeben. Die Methode selbst soll dann ein neues Auto produzieren, den Motor anlassen und eine kleine Testfahrt durchführen. Allerdings soll die Methode nicht wissen, wie das Auto hergestellt wird, schließlich wird eine Limousine anders produziert als ein Cabrio. Stattdessen verwendet die `sellVehicle()`-Methode dazu eine weitere Methode mit dem Namen `manufactureVehicle()`. Diese Methode soll ein neues Objekt erzeugen, das das Interface `Vehicle` implementiert, und dieses danach zurückgeben. Durch Implementieren dieses Interface können Sie sicher sein, dass Sie auf dem zurückgegebenen Objekt die Methoden `startEngine()`, `moveForward()` und `stopEngine()` aufrufen können.

```
namespace de\phpdesignpatterns\manufacturers;

abstract class AbstractManufacturer {

    protected $name;
```

```
    public function __construct($name) {
        $this->name = $name;
    }

    public function sellVehicle($color) {
        $vehicle = $this->manufactureVehicle($color);
        $vehicle->startEngine();
        $vehicle->moveForward(1);
        $vehicle->stopEngine();

        return $vehicle;
    }

    abstract protected function manufactureVehicle($color);
}
```

Die `manufactureVehicle()`-Methode haben Sie als abstrakte Methode deklariert, da die Hersteller-Basisklasse nicht wissen kann, wie ein Auto produziert werden muss. Die Basisklasse haben wir deshalb auch als abstrakte Klasse deklariert, sie kann nicht instanziiert werden, stattdessen müssen Unterklassen gebildet werden, um ein Objekt zu erzeugen.

Diese Unterklassen müssen dann lediglich die abstrakte Methode `manufactureVehicle()` implementieren und in ihr ein neues Fahrzeug instanziieren. Für eine Limousine sieht die neue Herstellerklasse folgendermaßen aus:

```
namespace de\phpdesignpatterns\manufacturers;

use de\phpdesignpatterns\vehicles\Car;

class CarManufacturer extends AbstractManufacturer {

    protected function manufactureVehicle($color) {
        $vehicle = new Car($this->name, $color);
        return $vehicle;
    }
}
```

Sie erzeugen also lediglich eine Instanz der Klasse `Car` und übergeben dabei den Herstellernamen und die gewünschte Farbe. Sie können diesen Hersteller nun in der Applikation einsetzen, um Autos zu produzieren und diese der Autovermietung zu verkaufen:

```
use de\phpdesignpatterns\manufacturers\CarManufacturer;

$bmwManufacturer = new CarManufacturer('BMW');
$bmw = $bmwManufacturer->sellVehicle('blau');

print "Neues Fahrzeug gekauft:\n";
print "Fahrzeugtyp: " . get_class($bmw) . "\n";
print "Hersteller : " . $bmw->getManufacturer() . "\n";
print "Farbe      : " . $bmw->getColor() . "\n";
```

Nach Erzeugen einer Instanz des Herstellers verwenden Sie die Methode `sellVehicle()`, um ein neues Auto zu erstellen. Dabei müssen Sie nicht mehr angeben, von welcher Klasse das neue Auto instanziiert wird, dieses Wissen ist in der Klasse `CarManufacturer` gekapselt. Wenn Sie dieses Skript ausführen, erhalten Sie die folgende Ausgabe:

```
Neues Fahrzeug gekauft:
Fahrzeugtyp: de\phpdesignpatterns\factoryMethod\Car
Hersteller : BMW
Farbe      : blau
```

Wie erwartet, ist das gekaufte Auto also eine Instanz der Klasse `Car`. Analog dazu können Sie einen weiteren Hersteller implementieren, der Cabrios produziert, Sie müssen nur noch eine Klasse von `AbstractManufacturer` ableiten:

```
namespace de\phpdesignpatterns\manufacturers;

use de\phpdesignpatterns\vehicles\Convertible;

class ConvertibleManufacturer extends AbstractManufacturer {

    protected function manufactureVehicle($color) {
        $vehicle = new Convertible($this->name, $color);
        return $vehicle;
    }
}
```

In dieser Klasse erzeugen Sie eine neue Instanz der Klasse `Convertible`, wenn Sie angewiesen werden, ein neues Auto zu produzieren. In beiden Klassen kümmert sich die Implementierung der Methode nur um das Erzeugen neuer Objekte, sie weiß nicht, wie diese im weiteren Verlauf verwendet werden. Den Cabrio-Hersteller verwenden Sie genau so wie im vorherigen Beispiel:

```
use de\phpdesignpatterns\manufacturers\ConvertibleManufacturer;

$peugeotManufacturer = new ConvertibleManufacturer('Peugeot');
$peugeot = $peugeotManufacturer->sellVehicle('schwarz');

print "Neues Fahrzeug gekauft:\n";
print "Fahrzeugtyp: " . get_class($peugeot) . "\n";
print "Hersteller : " . $peugeot->getManufacturer() . "\n";
print "Farbe      : " . $peugeot->getColor() . "\n";
```

Weitere Hersteller (die auch komplexeren Code verwenden könnten), um ein Auto zu produzieren, können also leicht implementiert und der Anwendung hinzugefügt werden. Sie haben es somit geschafft, das Erzeugen der Auto-Objekte aus dem Code zu entfernen und stattdessen hinter einer Schnittstelle zu kapseln. Dabei verwenden Sie stets nur die Schnittstelle der Klasse `AbstractManufacturer`, Sie müssen nicht wissen, welche Klassen dabei instanziiert werden und welcher Code dazu nötig ist.

Definition des Patterns

Das Fabrikmethoden-Muster definiert eine Schnittstelle zur Erzeugung von Objekten. Es verlagert aber die eigentliche Instanziierung in Unterklassen; es lässt die Unterklassen entscheiden, welche konkreten Implementierungen verwendet werden.

Um dieses Ziel zu erreichen, müssen Sie die folgenden Schritte durchführen:

1. Implementieren Sie eine abstrakte Klasse, in der Sie eine oder mehrere abstrakte Methoden deklarieren, die die Schnittstelle zum Erzeugen von Objekten vorgeben.
2. Fügen Sie dieser Klasse weitere Methoden hinzu, die Logik enthalten, die bei allen konkreten Implementierungen identisch sind. Sie können in diesen Methoden bereits auf die abstrakte Fabrikmethode zugreifen.
3. Bilden Sie beliebig viele Unterklassen, in denen Sie verschiedene Implementierungen der abstrakten Methode einfügen.
4. Verwenden Sie nun diese konkreten Unterklassen, um die tatsächlichen Objekte zu instanziieren und Ihren Applikationscode von den konkreten Implementierungen zu lösen.

Wann immer Sie eine Fabrikmethode verwenden möchten, achten Sie einfach darauf, die hier gezeigten Schritte durchzuführen, und dem Erfolg Ihres Vorhabens steht nichts mehr im Weg. Abbildung 4-2 zeigt Ihnen die Beziehungen zwischen den Beteiligten des Factory-Method-Patterns und illustriert noch einmal, wie das Pattern auf die Erzeugung der `Vehicle`-Implementierungen angewandt wurde.

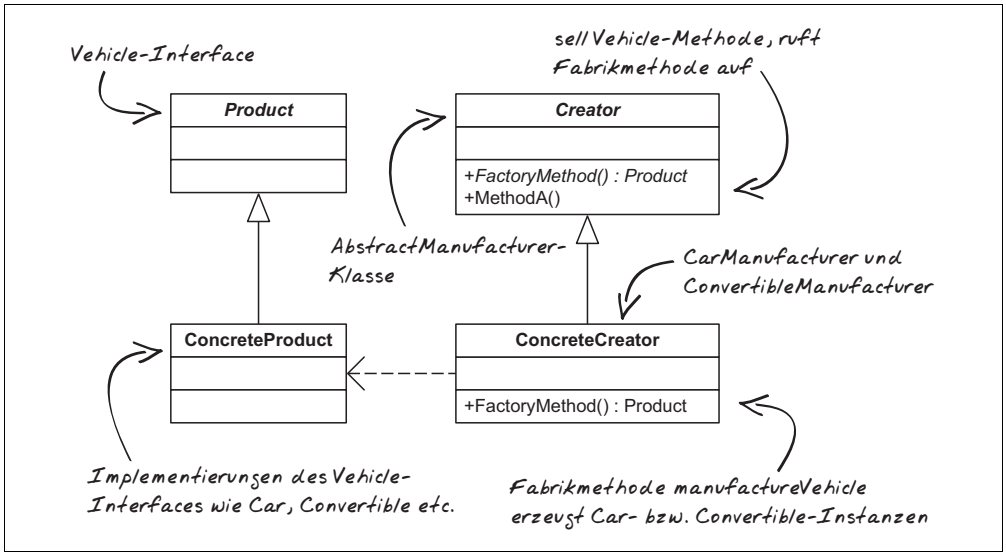

Abbildung 4-2: UML-Diagramm des Factory-Method-Patterns

Konsequenzen

Durch den Einsatz von Fabrikmethoden wird es Ihnen ermöglicht, dass Sie in Framework-Code stets nur gegen Schnittstellen entwickeln und konkrete Implementierungen dieser Schnittstelleen aus dem Framework-Code heraushalten können. Aus diesem Grund ist dieses eines der Muster, das Sie am häufigsten in den verschiedensten Open Source-Frameworks antreffen.

Die Fabrikmethode verlangt, dass Sie zur Erzeugung einer konkreten Implementierung eine andere Klasse ableiten. Oft müssen Sie die Erzeugerklasse nur zu diesem Zweck ableiten, was zu erhöhter Komplexität der Anwendung führt. Trotz dieses Nachteils überwiegen die Vorteile des Design Patterns, da Sie problemlos spezialisierte Klassen (wie im obigen Beispiel die `Convertible`-Klasse) in Ihre Applikation einfügen können. Im nächsten Abschnitt sehen Sie, wie durch eine Modifikation des Patterns sogar dieser kleine Nachteil umgangen werden kann.

Weitere Anwendungen

Die Vorgehensweise, die Instanziierung von Objekten in einer Fabrikmethode zu verstecken, ist ein mächtiges Hilfsmittel bei der Entwicklung objektorientierter Architekturen. Allerdings führt die Anwendung der Fabrikmethode wie in diesem Beispiel oft zu sehr vielen Klassen, die eigentlich nur eine Methode bereitstellen, die wiederum aus nur zwei Zeilen Quellcode besteht. Diese Fabriken müssen erneut instanziiert werden, womit Sie den Code an irgendeiner Stelle durch das Erzeugen der Fabrik-Instanz doch wieder an eine konkrete Implementierung binden.

Um diese Objektinflation zu vermeiden, wird in vielen Applikationen eine Abwandlung der Fabrikmethode angewandt, die *statische Fabrikmethode*. Wie der Name schon sagt, wird die Methode, die das Objekt erzeugt, statisch aufgerufen, anstatt zuerst ein Objekt zu erzeugen. Durch die Verwendung einer statischen Methode fällt natürlich die Möglichkeit der Bildung von Unterklassen weg, schließlich muss beim Aufruf der Methode fest der Name einer Klasse angegeben werden.

Stattdessen wird die statische Fabrikmethode parametrisiert, und anhand der übergebenen Parameter wird entschieden, welche konkrete Klasse instanziiert und zurückgegeben werden soll. Eine gute Anwendung der statischen Fabrikmethode ist erneut das Erzeugen der `Debugger`-Instanz. Dazu implementieren Sie eine neue Klasse mit dem Namen `DebuggerFactory`, die die Debugger erzeugen soll. Hierfür stellt die Klasse eine statische Methode `createDebugger()` zur Verfügung, der Sie den Typ des zu erzeugenden Debuggers übergeben:

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerFactory {
    static public function createDebugger($type) {
        switch (strtolower($type)) {
            case 'echo':
```

```
                require_once 'DebuggerEcho.php';
                $debugger = new DebuggerEcho();
                break;
            case 'log':
                require_once 'DebuggerLog.php';
                $debugger = new DebuggerLog();
                break;
            default:
                throw new UnknownDebuggerException();
        }
        return $debugger;
    }
}
```

In der Methode entscheiden Sie mithilfe einer `switch/case`-Anweisung und anhand des `$type`-Parameters, welche Klasse instanziiert werden soll. Dazu laden Sie zunächst die entsprechende Klasse über das Einbinden der entsprechenden PHP-Datei und erzeugen danach eine neue Instanz. Wenn ein unbekannter Typ übergeben wird, reagiert die Fabrikmethode mit dem Werfen einer Exception.

Diese Fabrikmethode können Sie jetzt verwenden, um den Debugger zu erzeugen:

```
use de\phpdesignpatterns\util\debug\DebuggerFactory;

define('DEBUG_MODE', 'echo');

$debugger = DebuggerFactory::createDebugger(DEBUG_MODE);
$debugger->debug('Danger, Will Robsinson!');
```

Der Code muss nun nichts mehr über die konkreten Implementierungen des Debuggers kennen, Sie rufen lediglich die Fabrikmethode auf, die Ihnen einen Debugger zurückliefert.

Statische Fabrikmethode und Singleton

Bei der Beschreibung des Singleton haben Sie gelernt, dass es sinnvoll ist, von zustandslosen Objekten wie dem Debugger durch Anwendung des Singleton-Patterns dafür zu sorgen, dass Sie nicht unnötig viele Instanzen der Klassen erzeugen. Durch Einführen der statischen Fabrikmethode haben Sie diesen Vorteil allerdings gegen einen anderen Vorteil eingetauscht, da Sie nicht einmal mehr die Klassennamen der Debugger-Implementierungen kennen müssen. Besser wäre es natürlich, wenn Sie beides in einem nutzen könnten.

Auch das ist mithilfe der statischen Fabrikmethode möglich, sie muss lediglich Objekte zurückliefern. Es ist nicht definiert, dass diese mithilfe des `new`-Operators erzeugt werden müssen. Also kombinieren Sie doch einfach die Fabrikmethode mit Ihrer Singleton-Implementierung ...

```
namespace de\phpdesignpatterns\util\debug;

class DebuggerFactory {
    static public function createDebugger($type) {
```

```
            switch (strtolower($type)) {
                case 'echo':
                    $debugger = DebuggerEcho::getInstance();
                    break;
                case 'log':
                    $debugger = DebuggerLog::getInstance();
                    break;
                default:
                    throw new UnknownDebuggerException();
            }
            return $debugger;
        }
    }
```

Statt des new-Operators verwenden Sie jetzt die getInstance()-Methode, die immer die gleiche Instanz zurückliefert. Sie haben also die statische Fabrikmethode und das Singleton sowie deren Vorteile in einem und mussten dabei nicht einmal den existierenden Quellcode verändern.

Das Abstract-Factory-Pattern

Mithilfe der Fabrikmethode und des Singleton haben Sie einzelne Objekte erzeugt und dabei die eigentliche Instanziierung der Objekte in Methodenaufrufen versteckt. Oft werden Sie jedoch vor Problemen stehen, bei denen Sie mehr als ein Objekt herstellen müssen und diese Objekte zu einer bestimmten Objektfamilie gehören. Hier hilft Ihnen das *Abstract-Factory-Pattern*, auch *abstrakte Fabrik* genannt.

Motivation

Nachdem Sie nun neue Autos kaufen und vermieten können, möchten Sie sich als Nächstes um die Darstellung des Fuhrparks kümmern. Den Anfang soll eine einfache Liste aller Autos machen. Dabei haben Sie die Daten schon in einer recht einfachen Form zugänglich gemacht und z.B. ein solches Array erhalten:

```
$vehicles = array(
           array('BMW', 'blau'),
           array('Peugeot', 'rot'),
           array('VW', 'schwarz'),
         );
```

Sie haben also ein Array, das zu jedem Wagen den Hersteller und die Farbe enthält. Diese Informationen möchten Sie nun mindestens in zwei Formaten darstellen können. Zum einen sollen die Daten auf einer Website als HTML-Tabelle angezeigt werden, wie Abbildung 4-3 zeigt, und zum anderen möchten Sie dieselben Daten auch auf der Kommandozeile ausgeben können. Abbildung 4-4 zeigt, wie diese Darstellung aussehen sollte.

Möglicherweise möchten Sie die Daten später noch in anderen Ausgabemedien als Tabelle darstellen können. Weiterhin wäre es wünschenswert, wenn Sie diese Tabellen

später auch verwenden könnten, um zum Beispiel eine Liste aller Kunden oder auch die Ausleihvorgänge eines Kunden zu generieren. Sie möchten also die Darstellung von den Inhalten getrennt halten.

Abbildung 4-3: Darstellung der Liste als HTML-Tabelle

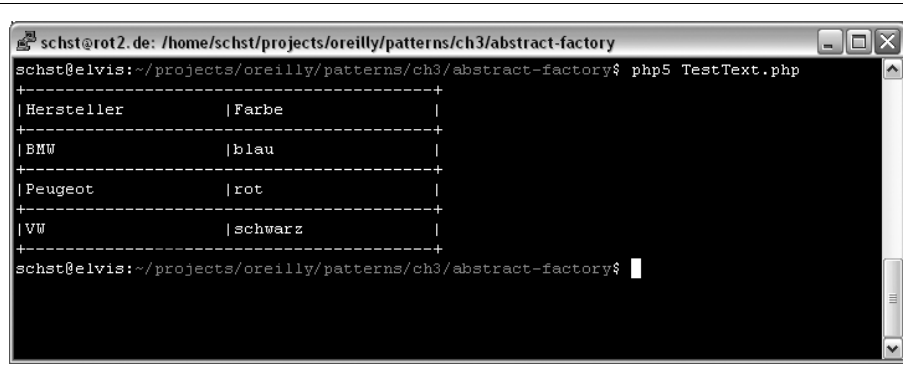

Abbildung 4-4: Darstellung der Liste auf der Kommandozeile

Um dies zu ermöglichen, splitten Sie die Ausgabe in ihre einzelnen Komponenten auf, um diese dann nach den bisher gelernten Prinzipien leicht austauschen zu können. Im Fall einer Tabelle benötigen Sie die folgenden Komponenten:

- Die Tabelle selbst, die alle weiteren Elemente beinhaltet.
- Die Kopfzeile der Tabelle, die die Überschriften beinhaltet.
- Die einzelnen Zeilen der Tabelle.
- Die einzelnen Tabellenzellen, aus denen sich eine Zeile zusammensetzen lässt.

Sie möchten nun eine Lösung finden, diese einzelnen Elemente zu verwenden und daraus beliebige Tabellen zu konstruieren und mit Daten zu befüllen, ohne dass Sie dabei schon wissen müssen, wie die Tabelle später dargestellt werden soll. Dies bedeutet also, dass

Sie einzelne Instanzen dieser Elemente erzeugen müssen, ohne zu wissen, ob diese Instanzen für HTML- oder Textausgaben zuständig sind.

Zweck des Patterns

Um die einzelnen Komponenten zu erzeugen, ohne deren konkrete Implementierungen angeben zu müssen, werden Sie nun eine *abstrakte Fabrik* verwenden. Die Definition dieses Musters passt genau zum aktuellen Problem.

Die abstrakte Fabrik bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne deren konkrete Klassen zu benennen.

Um die Listen mithilfe einer abstrakten Fabrik umzusetzen, sind die folgenden Schritte nötig:

1. Definieren einer Schnittstelle zum Erzeugen der einzelnen Elemente der Liste, wie Tabellen, Kopfzeilen, Zeilen und Zellen.
2. Umsetzen von mindestens zwei konkreten Implementierungen dieser Schnittstelle für HTML- und Textausgabe.
3. Verwenden der Fabrik, um mit den von ihr erzeugten Objekten eine Liste der verfügbaren Fahrzeuge auf Basis der vorhandenen Daten zu erstellen.

Implementierung

Um die Liste mithilfe der abstrakten Fabrik umzusetzen, benötigen Sie zunächst die einzelnen Elemente, aus denen Sie die Tabelle zusammenbauen können.

Dazu müssen Sie die folgenden Klassen implementieren:

- `Table` als Repräsentation der gesamten Tabelle
- `Row` als Repräsentation einer Tabellenzeile
- `Header` als Repräsentation des Tabellenkopfs
- `Cell` als Repräsentation einer Tabellenzelle

Jede dieser Klassen soll eine Methode mit dem Namen `display()` bieten, die aufgerufen wird, wenn das Element ausgegeben werden soll. Da die Repräsentation erst einmal nicht weiß, wie die Darstellung aussehen soll, wird diese Methode als abstrakt deklariert, wodurch auch sämtliche dieser Klassen abstrakte Klassen sein müssen.

Diese bieten jedoch die Grundfunktionalität, die benötigt wird, um aus den einzelnen Elementen eine komplette Tabelle zu erzeugen. Im Folgenden werden Sie nun diese Basisfunktionalität in den einzelnen Klasse implementieren. Dabei beginnen Sie mit der Klasse `Table`.

```
namespace de\phpdesignpatterns\tables;

abstract class Table {
```

```
    protected $header = null;
    protected $rows = array();

    public function setHeader(Header $header) {
        $this->header = $header;
    }

    public function addRow(Row $row) {
        $this->rows[] = $row;
    }

    abstract public function display();
}
```

Zur Tabelle selbst fügen Sie zwei Eigenschaften hinzu, `$header` speichert den Tabellenkopf und `$rows` die einzelnen Zeilen in einem Array. Um den Tabellenkopf zu definieren, haben Sie die Methode `setHeader()` hinzugefügt, der Sie eine Instanz der Klasse `Header` übergeben können. Über die Methode `addRow()` können beliebig viele Zeilen der Tabelle hinzugefügt werden. Weiterhin deklarieren Sie die bereits besprochene `display()`-Methode.

Als Nächstes kümmern Sie sich um die Implementierung einer Tabellenzeile:

```
namespace de\phpdesignpatterns\tables;

abstract class Row {
    protected $cells = array();

    public function addCell(Cell $cell) {
        $this->cells[] = $cell;
    }

    abstract public function display();
}
```

Jede Zeile besteht aus mehreren Tabellenzellen, Sie fügen deshalb eine Eigenschaft `$cells` der Klasse hinzu, die diese Zellen in einem Array aufnehmen kann. Natürlich fügen Sie die Methode `addCell()` hinzu, um diese Eigenschaft zu füllen. Zur Darstellung der Zeile wird auch hier die `display()`-Methode deklariert.

Die Implementierung der Kopfzeile ist die einfachste Klasse der Tabellenelemente. Diese wird lediglich von der Klasse `Row` abgeleitet, da sie die gleiche Funktionalität wie eine einzelne Zeile bieten muss.

```
namespace de\phpdesignpatterns\tables;

abstract class Header extends Row {
}
```

Als Letztes bleibt Ihnen nur noch, die Klasse für eine einzelne Tabellenzelle zu implementieren. Zu einer Tabellenzelle speichern Sie lediglich den Inhalt als String; dieser kann im Konstruktor der Klasse übergeben werden.

```
namespace de\phpdesignpatterns\tables;

abstract class Cell {
    protected $content = null;

    public function __construct($content) {
        $this->content = $content;
    }
    abstract public function display();
}
```

Und auch diese Klasse verfügt natürlich über die abstrakte `display()`-Methode.

Nachdem Sie die Klassen für die einzelnen Tabellenelemente implementiert haben, kümmern Sie sich als Nächstes um die Schnittstelle der Fabriken, die Sie verwenden werden, um die Tabellenelemente zu erzeugen. Diese müssen pro Element eine Methode bieten:

```
namespace de\phpdesignpatterns\tables;

interface TableFactory {
    public function createTable();
    public function createRow();
    public function createHeader();
    public function createCell($content);
}
```

Da Sie in der Schnittstelle keine konkreten Methoden bereitstellen, verwenden Sie hier, im Gegensatz zur Fabrikmethode, einfach ein Interface. Bei der Methode, die eine Tabellenzelle erstellen soll, wird der Inhalt der Zelle an die Fabrikmethode übergeben, da wir diese beim Instanziieren der Tabellenzelle übergeben müssen. Alle anderen Methoden werden ohne Parameter aufgerufen.

Damit haben Sie die Schnittstellen aller Beteiligten definiert.

Die Tabelle in HTML

Um nun eine HTML-Tabelle zu erstellen, benötigen Sie zunächst konkrete Unterklassen der Tabellenelemente, die die `display()`-Methode so implementieren, dass bei deren Aufruf HTML erzeugt wird. Sie gehen dieses Mal genau umgekehrt vor und implementieren zuerst die neue Klasse für eine Zelle:

```
namespace de\phpdesignpatterns\tables\html;

use de\phpdesignpatterns\tables\Cell;

class HtmlCell extends Cell {
    public function display() {
        print "    <td>{$this->content}</td>\n";
    }
}
```

Um eine Tabellenzelle in HTML darzustellen, kann das `<td></td>`-Tag verwendet werden. Der Inhalt der Tabellenzelle steht Ihnen in der Objekteigenschaft `$content` zur Ver-

fügung. Danach kümmern Sie sich um die Darstellung einer Zeile, indem Sie die neue Klasse `HtmlRow` implementieren:

```
namespace de\phpdesignpatterns\tables\html;

use de\phpdesignpatterns\tables\Row;

class HtmlRow extends Row {
    public function display() {
        print "  <tr>\n";
        foreach ($this->cells as $cell) {
            $cell->display();
        }
        print "  </tr>\n";
    }
}
```

Eine Zeile in einer HTML-Tabelle wird dabei durch die Tags `<tr>` und `</tr>` begrenzt. Zwischen diesen Tags werden alle Zellen der Zeile ausgegeben, indem Sie über alle Zellen iterieren, die in der Eigenschaft `$cells` gespeichert werden, und diese werden durch Aufruf der `display()`-Methode ausgeben. Analog dazu kann der Tabellenkopf ausgegeben werden, Sie formatieren lediglich den Text dieser Zeile in Fettschrift:

```
namespace de\phpdesignpatterns\tables\html;

use de\phpdesignpatterns\tables\Header;

class HtmlHeader extends Header {
    public function display() {
        print "  <tr style=\"font-weight: bold;\">\n";
        foreach ($this->cells as $cell) {
            $cell->display();
        }
        print "  </tr>\n";
    }
}
```

Schließlich müssen Sie sich nur noch um die Implementierung der Tabelle selbst kümmern. Hier geben Sie zuerst ein öffnendes `<table>`-Tag aus, gefolgt vom Tabellenkopf. Danach iterieren Sie über die einzelnen Zeilen der Tabelle und geben diese aus, bevor Sie die Tabelle mit einem schließenden `</table>`-Tag beenden:

```
namespace de\phpdesignpatterns\tables\html;

use de\phpdesignpatterns\tables\Table;

class HtmlTable extends Table {
    public function display() {
        print "<table border=\"1\">\n";
        $this->header->display();
        foreach ($this->rows as $row) {
            $row->display();
        }
```

```
            print "</table>";
        }
    }
```

Diese Objekte können jetzt verwendet werden, um eine HTML-Tabelle zu erstellen. Dazu erzeugen Sie einfach die einzelnen Elemente und verwenden die `addCell()`-, `addRow()`- und `setHeader()`-Methoden, um die gesamte Tabelle aus den einzelnen Elementen zusammenzusetzen:

```
use de\phpdesignpatterns\tables\html\HtmlTable;
use de\phpdesignpatterns\tables\html\HtmlHeader;
use de\phpdesignpatterns\tables\html\HtmlRow;
use de\phpdesignpatterns\tables\html\HtmlCell;

$table = new HtmlTable();
$header = new HtmlHeader();
$header->addCell(new HtmlCell('Spalte 1'));
$header->addCell(new HtmlCell('Spalte 2'));
$table->setHeader($header);

$row = new HtmlRow();
$row->addCell(new HtmlCell('Zeile 1 / Spalte 1'));
$row->addCell(new HtmlCell('Zeile 1 / Spalte 2'));
$table->addRow($row);

$table->display();
```

Wenn Sie dieses Skript ausführen, sehen Sie eine einfache HTML-Seite in Ihrem Browser, da der folgende HTML-Code erzeugt wurde:

```
<table border="1">
  <tr style="font-weight: bold;">
    <td>Spalte 1</td>
    <td>Spalte 2</td>
  </tr>
  <tr>
    <td>Zeile 1 / Spalte 1</td>
    <td>Zeile 1 / Spalte 2</td>
  </tr>
</table>
```

Sie haben jetzt also einen Teil der Aufgabe erfüllt: Sie können eine HTML-Tabelle erzeugen und dabei den Inhalt der Tabelle von außen übergeben. Allerdings haben Sie die Objekte über den `new`-Operator erzeugt und damit leider eine feste Bindung des Codes an die konkreten Implementierungen in Kauf nehmen müssen. Wollen Sie jetzt statt einer HTML-Tabelle eine rein textbasierte Tabelle erzeugen, müssten Sie fast jede Zeile des Quellcodes anpassen. Um dies zu vermeiden, hatten Sie eigentlich die Schnittstelle `TableFactory` definiert. Sie brauchen nun also eine Klasse, die diese Schnittstelle implementiert und dabei die konkreten Implementierungen zurückliefert, die für die HTML-Tabelle zuständig sind. Dazu erstellen Sie eine neue Klasse `HtmlTableFactory`, die das `TableFactory`-Interface implementiert:

```
namespace de\phpdesignpatterns\tables\html;

use de\phpdesignpatterns\tables\TableFactory;

class HtmlTableFactory implements TableFactory {
    public function createTable() {
        $table = new HtmlTable();
        return $table;
    }
    public function createRow() {
        $row = new HtmlRow();
        return $row;
    }
    public function createHeader() {
        $header = new HtmlHeader();
        return $header;
    }
    public function createCell($content) {
        $cell = new HtmlCell($content);
        return $cell;
    }
}
```

In jeder der Methoden erzeugen Sie einfach ein neues Objekt vom entsprechenden Typ und geben dies zurück. Mithilfe dieser Fabrik können Sie jetzt das Beispiel so umschreiben, dass Sie keine konkreten Implementierungen der Tabelle, Zeilen oder Zellen erzeugen müssen. Stattdessen verwenden Sie dazu die Methoden der konkreten Fabrik:

```
use de\phpdesignpatterns\tables\html\HtmlTableFactory;

$factory = new HtmlTableFactory();

$table = $factory->createTable();
$header = $factory->createHeader();
$header->addCell($factory->createCell('Spalte 1'));
$header->addCell($factory->createCell('Spalte 2'));
$table->setHeader($header);

$row = $factory->createRow();
$row->addCell($factory->createCell('Zeile 1 / Spalte 1'));
$row->addCell($factory->createCell('Zeile 1 / Spalte 2'));
$table->addRow($row);

$table->display();
```

Wenn Sie dieses Skript ausführen, erhalten Sie die gleiche Ausgabe wie im obigen Beispiel.

Als Letztes bleibt jetzt noch, eine Klasse zu implementieren, die die Liste des Fuhrparks auf Basis eines Arrays ausgibt. Die Klasse soll zum Erzeugen der Tabellenelemente eine beliebige Fabrik verwenden, die das Interface `TableFactory` implementiert. Damit haben Sie die Klasse, die die Liste erzeugt, von der eigentlichen Darstellung in HTML entkop-

pelt. Die konkrete Fabrik wollen Sie der neuen Klasse dann beim Instanziieren übergeben, Sie verwenden hierzu also erneut *Dependency Injection*.

```
namespace de\phpdesignpatterns\tables\util;

use de\phpdesignpatterns\tables\TableFactory;

class VehicleList {
    protected $tableFactory = null;

    public function __construct(TableFactory $tableFactory) {
        $this->tableFactory = $tableFactory;
    }
}
```

Die übergebene Fabrik wird in einer Eigenschaft des Objekts gespeichert, um zu einem späteren Zeitpunkt erneut darauf zugreifen zu können. Der folgende Code erzeugt eine neue Instanz der Listenklasse:

```
use de\phpdesignpatterns\tables\util\VehicleList;
use de\phpdesignpatterns\tables\html\HtmlTableFactory;

$factory = new HtmlTableFactory();
$list = new VehicleList($factory);
```

Der Klasse `VehicleList` fehlt nun noch eine Methode, die die tatsächliche Tabelle auf Basis des Arrays erstellt. Um Ihr Gedächtnis aufzufrischen, schauen Sie sich noch einmal das zur Verfügung stehende Array an:

```
$vehicles = array(
           array('BMW', 'blau'),
           array('Peugeot', 'rot'),
           array('VW', 'schwarz'),
        );
```

Es enthält also mehrere Arrays, die die einzelnen Zeilen repräsentieren. Jede Zeile besteht aus zwei Elementen, dem Hersteller und der Farbe. Die Methode muss daher eine zweispaltige Tabelle erstellen. Die erste Spalte wird den Hersteller enthalten, die zweite die Farbe. Mithilfe der Klassen `Table`, `Header`, `Row` und `Cell` ist dies in wenigen Zeilen Code möglich:

```
namespace de\phpdesignpatterns\tables\util;

use de\phpdesignpatterns\tables\TableFactory;

class VehicleList {
    ... Konstruktor ...
    public function showTable($data) {

        $table = $this->tableFactory->createTable();

        // Kopfzeile erstellen.
        $header = $this->tableFactory->createHeader();
```

```
            $header->addCell($this->tableFactory->createCell('Hersteller'));
            $header->addCell($this->tableFactory->createCell('Farbe'));

            $table->setHeader($header);

            // Einzelne Zeilen ausgeben.
            foreach ($data as $line) {
                $row = $this->tableFactory->createRow();
                $table->addRow($row);
                foreach ($line as $field) {
                    $cell = $this->tableFactory->createCell($field);
                    $row->addCell($cell);
                }
            }
            $table->display();
        }
    }
```

Der Code ähnelt stark dem bisherigen Beispiel zum Erstellen einer HTML-Tabelle mithilfe der Klasse `HtmlTableFactory`. Und wenn Sie sich den Code genauer betrachten, stellen Sie fest, dass Sie an keiner Stelle gegen eine konkrete Implementierung, sondern immer nur gegen Schnittstellen programmiert haben. Sie haben also wieder einmal eine der wichtigsten Regeln befolgt.

Um den Code zu testen, genügt das folgende Skript:

```
use de\phpdesignpatterns\tables\util\VehicleList;
use de\phpdesignpatterns\tables\html\HtmlTableFactory;

$list = new VehicleList(new HtmlTableFactory());
$list->showTable($data);
```

Öffnen Sie dieses Skript im Browser, erhalten Sie als Ausgabe genau die Tabelle aus Abbildung 4-3, eben genau das Ergebnis, das Sie erreichen wollten. Sie erzeugen also eine HTML-Tabelle, ohne bei der Generierung der Tabelle in der Klasse `VehicleList` zu wissen, dass es sich um eine HTML-Tabelle handelt.

Ausgabe auf der Kommandozeile

Was noch fehlt, ist jetzt die zweite Tabelle, die auf der Kommandozeile ausgegeben werden soll. Dazu benötigen Sie nur eine zweite Implementierung der `TableFactory`-Schnittstelle, die Elemente zurückliefert, die eine reine Textversion der Tabelle erstellen können.

Der Weg dazu ist der gleiche, den Sie auch beim Erstellen der HTML-Tabelle beschritten haben. Als Erstes schreiben Sie die konkreten Implementierungen der `Table`-, `Header`-, `Row`- und `Cell`-Klassen:

```
namespace de\phpdesignpatterns\tables\ascii;

use de\phpdesignpatterns\tables\Table;
use de\phpdesignpatterns\tables\Row;
use de\phpdesignpatterns\tables\Header;
```

```
use de\phpdesignpatterns\tables\Cell;

class TextTable extends Table {
    public function display() {
        $this->header->display();
        foreach ($this->rows as $row) {
            $row->display();
        }
    }
}

class TextRow extends Row {
    public function display() {
        foreach ($this->cells as $cell) {
            $cell->display();
        }
        print "|\n";
        print "+" . str_repeat("-", (count($this->cells) * 21)-1) . "+\n";
    }
}

class TextHeader extends Header {
    public function display() {
        print "+" . str_repeat("-", (count($this->cells) * 21)-1) . "+\n";
        foreach ($this->cells as $cell) {
            $cell->display();
        }
        print "|\n";
        print "+" . str_repeat("-", (count($this->cells) * 21)-1) . "+\n";
    }
}

class TextCell extends Cell {
    public function display() {
        print '|' . str_pad($this->content, 20);
    }
}
```

Um die Zeilen und Spalten zu begrenzen, verwenden Sie die Zeichen »-«, »|« und »+«. Ansonsten ähnelt die Implementierung sehr stark der HTML-Implementierung:

- Bei Ausgabe der Tabelle wird zuerst der Tabellenkopf ausgegeben und danach über die einzelnen Zellen iteriert.
- Bei der Ausgabe einer Zeile wird über die einzelnen Zellen iteriert und jede Zelle ausgegeben.

Durch die Funktion `str_pad()` legen Sie die Breite einer jeden Zelle auf 20 Zeichen fest. Wenn der Inhalt einer Zelle kürzer als 20 Zeichen ist, wird dieser durch Leerzeichen aufgefüllt. Mithilfe der PHP-Funktion `str_repeat()` erzeugen Sie die Zeilentrenner, indem Sie den Bindestrich mehrfach wiederholen. Um die Anzahl der Wiederholungen

zu erhalten, zählen Sie einfach die Anzahl der Spalten und multiplizieren diese mit der Breite der Spalten.

Als Nächstes brauchen Sie nur noch eine Fabrik, die diese Tabellenelemente erzeugen kann und die Schnittstelle `TableFactory` implementiert. Auch diese wird analog zur `HtmlTableFactory`-Klasse implementiert:

```
namespace de\phpdesignpatterns\tables\ascii;

use de\phpdesignpatterns\tables\TableFactory;

class TextTableFactory implements TableFactory {
    public function createTable() {
        $table = new TextTable();
        return $table;
    }
    public function createRow() {
        $row = new TextRow();
        return $row;
    }
    public function createHeader() {
        $header = new TextHeader();
        return $header;
    }
    public function createCell($content) {
        $cell = new TextCell($content);
        return $cell;
    }
}
```

Der einzige Unterschied zur Fabrik zum Erzeugen von HTML-Tabellen sind die Namen der Klassen, die verwendet werden. Da Sie alle Schnittstellen nun vollständig implementiert haben, können Sie die neue Tabellenfabrik testen, indem Sie statt der `HtmlTableFactory` eine `TextTableFactory` an das Listen-Objekt übergeben:

```
use de\phpdesignpatterns\tables\util\VehicleList;
use de\phpdesignpatterns\tables\ascii\TextTableFactory;

$list = new VehicleList(new TextTableFactory());
$list->showTable($data);
```

Natürlich müssen Sie dieses Skript jetzt auf der Kommandozeile statt im Browser ausführen, da Sie keinen HTML-Code mehr erzeugen. Nach dem Starten des Skripts sehen Sie eine Tabelle, die genau wie das gewünschte Ergebnis in Abbildung 4-4 aussieht.

Die abstrakte Fabrik hat es Ihnen also ermöglicht, eine Familie verwandter Objekte zu erzeugen, ohne dass Sie dabei die eigentlichen Klassennamen angeben mussten. Daher können Sie durch Austauschen der Fabrik eine ganze Familie von Objekten austauschen. In diesem Fall waren die Objekte für die Darstellung der Daten verantwortlich, natürlich ist die abstrakte Fabrik nicht darauf beschränkt.

Definition des Patterns

Die abstrakte Fabrik bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne deren konkrete Klassen zu benennen.

Um dies zu erreichen, sind die folgenden Schritte nötig:

1. Definieren Sie die Schnittstelle der Fabrik, indem Sie pro zu erzeugendem Objekt der Objektfamilie eine Methode deklarieren. Für die Schnittstelle kann sowohl ein Interface als auch eine abstrakte Klasse verwendet werden.
2. Implementieren Sie abstrakte Klassen, die die einzelnen Objekte der Objektfamilie repräsentieren.
3. Implementieren Sie beliebige konkrete Unterklassen dieser abstrakten Klassen.
4. Implementieren Sie eine oder mehrere konkrete Fabriken, die diese Objekte erzeugen.
5. Übergeben Sie die konkrete Fabrik, die verwendet werden soll, an Ihre Applikation. Hierzu können Sie zum Beispiel Dependency Injection, aber auch eine Fabrikmethode verwenden. Dadurch ist es zu einem späteren Zeitpunkt möglich, die Fabrik und somit die von ihr erstellten Objekte auszutauschen.

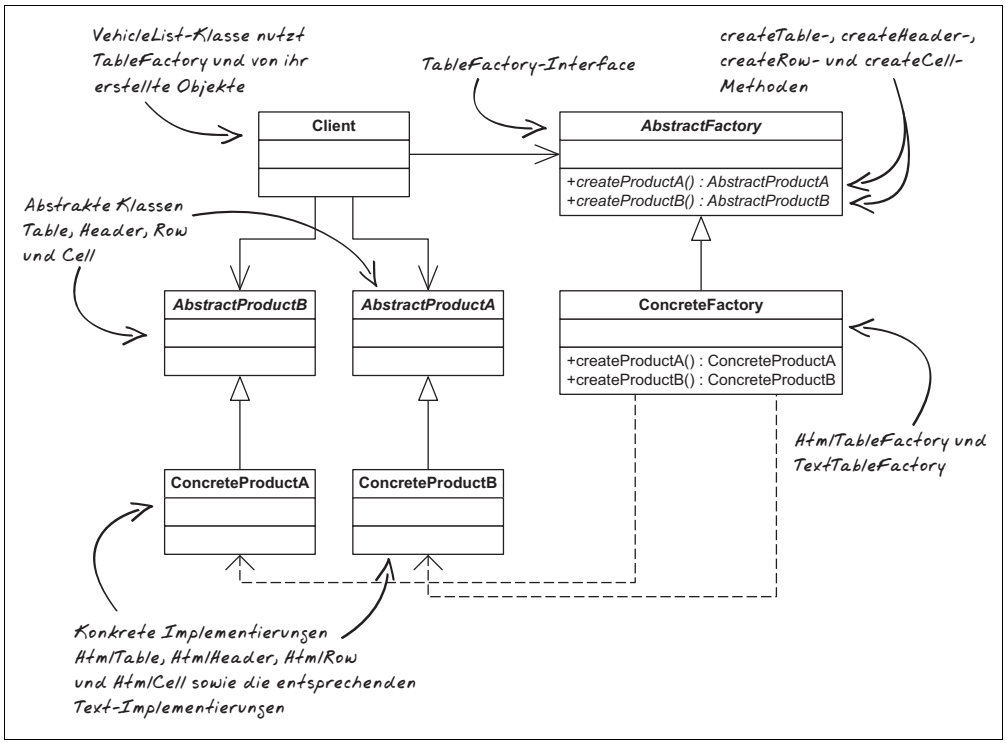

Abbildung 4-5: UML-Diagramm der abstrakten Fabrik

Befolgen Sie einfach diese Schritte, wenn Sie eine abstrakte Fabrik an weiteren Stellen Ihrer Anwendung verwenden möchten. Abbildung 4-5 zeigt Ihnen die Beziehungen der Klassen zueinander und wie das Pattern auf die Erstellung der Tabellen angewandt wurde.

Konsequenzen

Wie die meisten Erzeugungsmuster erlaubt Ihnen die abstrakte Fabrik, die konkreten Klassen zu isolieren und sie von den Klassen und Objekten zu trennen, die diese Klassen verwenden. Sie können dadurch sehr einfach komplette Produktfamilien austauschen. Unter Produktfamilien versteht man dabei Objekte, die in einer Beziehung zueinander stehen. Die abstrakte Fabrik stellt sicher, dass immer die komplette Produktfamilie zusammen eingesetzt wird und es nicht möglich ist, nur einzelne Klassen auszutauschen.

Die Unterstützung neuer Produkte ist sehr komplex, da ein neues Produkt für jede Produktfamilie implementiert werden muss.

Weitere Anwendungen

Die abstrakte Fabrik wird sehr häufig in Frameworks eingesetzt. Dabei liefert das Framework die Grundfunktionalität für die einzelnen Komponenten, ermöglicht jedoch die Erweiterung der Komponenten um zusätzliche Funktionen. Dazu muss einfach nur eine andere Fabrik verwendet werden, die dann anstelle der mitgelieferten Klassen die angepassten Klassen verwendet. Diese müssen lediglich die gleichen Interfaces implementieren. Damit können Sie zum Beispiel die Benutzerverwaltung eines Frameworks austauschen, und somit kann die Anwendung an ein bestehendes System angepasst werden. Es wird Ihnen ermöglicht, Benutzernamen und Passwörter zum Beispiel gegen einen bestehenden LDAP-Server zu authentifizieren, obwohl die Applikation ursprünglich Benutzerdaten in einer Datenbank speichert.

Außerdem wird die abstrakte Fabrik oft als Singleton implementiert, da es pro Objektfamilie ausreicht, eine Fabrikinstanz zu verwenden. Dieses Muster kann dabei genau so, wie bereits beschrieben, implementiert werden.

Das Prototype-Pattern

Wenn Sie die bislang vorgestellten Muster zur Erzeugung von Objekten verwenden, kann es zu Situationen kommen, in denen sehr viele Klassen implementiert werden müssen, die nur dazu dienen, verschiedene konkrete Implementierungen Ihrer Klassen zu erzeugen. Das *Prototype-Pattern* kann helfen, die Anzahl der benötigten Klassen klein zu halten.

Motivation

Autos werden heutzutage hauptsächlich über gutes Marketing verkauft. Um eine Bindung des Kunden an ein bestimmtes Auto herzustellen, werden häufig Sondereditionen von bestimmten Modellen verkauft, um gezielt eine Kundengruppe anzusprechen. Eine beliebte Methode ist es, Sondermodelle mit den Namen von bekannten Künstlern herzustellen, wie z.B. den Golf »Edition Rolling Stones« oder »Edition Elvis Presley«. Natürlich kommen Sie in Ihrer Anwendung auch nicht umhin, Ihren Kunden diese Sondermodelle als Mietwagen anzubieten. Vor dem Vermieten steht jedoch der Kauf, deswegen sollten Sie sich zuerst wieder mit der Fabrikation der Sondermodelle beschäftigen.

Das folgende Beispiel zeigt, wie eine Instanz eines solchen Sondermodells erzeugt wird. Ein Sondermodell ist in erster Linie eine Instanz vom Typ `Vehicle`, normalerweise immer in einer bestimmten Farbe und mit zusätzlichen Eigenschaften:

```
$GolfElvis = new Car('VW', 'silber');
$GolfElvis->setAirConditioned(true);
$GolfElvis->setGraphics('Gitarre');
```

In diesem Beispiel wurden der Klasse `Car` zwei Methoden hinzugefügt, um zu definieren, ob das Auto über eine Klimaanlage verfügt und ob es mit Grafiken auf der Karosserie ausgestattet wurde. Im Fall der »Sonderedition Elvis Presley« liegt es nahe, das Auto mit dem Bild einer Gitarre zu verschönern.

Um dieses Sondermodell jetzt in Serie herzustellen, könnten Sie das Abstract-Factory-Pattern verwenden und eine `GolfElvisPresleyManufacturer`-Klasse implementieren, die genau weiß, wie dieses Sondermodell hergestellt werden muss. Das Problem bei dieser Art der Implementierung ist jedoch, dass Sie danach noch eine Klasse `GolfRollingStonesManufacturer` implementieren müssten, die in der Lage ist, die »Sonderedition Rolling Stones« zu erstellen. Für jede Sonderedition benötigen Sie also eine Fabrik, was zu einem inflationären Wachstum der Klassenanzahl führt.

Zweck des Patterns

Das Entwurfsmuster, das Ihnen hilft, diese inflationäre Klassenvermehrung zu verhindern, ist das *Prototype-Pattern*:

Das Prototyp-Muster bestimmt die Arten der zu erzeugenden Objekte durch die Verwendung eines prototypischen Exemplars, das zur Erzeugung neuer Instanzen kopiert wird.

Um dieses Muster auf Ihr Problem mit der Erzeugung von Sondereditionen zu übertragen, müssen Sie die folgenden Schritte gehen:

1. Erstellen Sie eine Fabrik-Klasse, der Sie die zu kopierenden Objekte übergeben können, und implementieren Sie eine Methode, die die Prototyp-Instanzen kopiert.
2. Erstellen Sie Prototyp-Instanzen Ihrer Sondereditionen und übergeben Sie diese an die Fabrik.

Implementierung

Bevor Sie mit der Implementierung der Fabrik beginnen können, müssen Sie zunächst die beiden Methoden implementieren, die den Autos Informationen darüber, ob eine Klimaanlage vorhanden ist und welche Grafiken verwendet wurden, hinzufügen.

Neben den Methoden zum Speichern der Informationen fügen Sie natürlich auch Getter-Methoden ein, um diese Informationen abzufragen. Als Basis dient die Klasse Car, die bereits aus den vorherigen Kapiteln kennen.

```
namespace de\phpdesignpatterns\vehicles;

class Car implements Vehicle {
    ... bisherige Properties ...
    protected $airConditioned = false;
    protected $graphics = null;

    ... bisherige Methoden ...
    public function setGraphics($graphics) {
        $this->graphics = $graphics;
    }

    public function setAirConditioned($airConditioned) {
        $this->airConditioned = $airConditioned;
    }

    public function getGraphics() {
        return $this->graphics;
    }

    public function hasAirCondition() {
        return $this->airConditioned;
    }
}
```

Nachdem Sie diese Klasse mit den Methoden ausgestattet haben, können Sie nun ganz einfach einen Prototyp für die »Sonderedition Elvis Presley« erstellen:

```
use de:phpdesignpatterns\vehicles\Car;

$GolfElvis = new Car('VW', 'silber');
$GolfElvis->setAirConditioned(true);
$GolfElvis->setGraphics('Gitarre');
```

Nach der Erzeugung der Instanz legen Sie fest, dass diese Sonderedition immer mit Klimaanlage ausgestattet und die Karosserie durch Anbringen des Bildes einer Gitarre verschönert werden soll.

Möchten Sie nun einen neuen Golf aus dieser Edition erstellen, verwenden Sie den `clone`-Operator von PHP, um das Objekt zu kopieren:

```
$myGolf = clone $GolfElvis;
```

Sie sind nun also in der Lage, eine »Vorlage« für eine Sonderedition zu erstellen und diese beliebig oft zu reproduzieren. Als Nächstes kapseln Sie diese Funktionalität in einer Klasse, die Ihre Sondereditionen verwaltet. Dazu erstellen Sie eine neue Klasse `SpecialEditionManufacturer`, die die verschiedenen Sondereditionen in der Eigenschaft $properties speichert. Um neue Sondermodelle registrieren zu können, fügen Sie die Methode `addSpecialEdition()` ein:

```
namespace de\phpdesignpatterns\manufacturers;
use de\phpdesignpatterns\vehicles\Vehicle;

class SpecialEditionManufacturer {
    protected $prototypes = array();

    public function addSpecialEdition($edition, Vehicle $prototype) {
        $this->prototypes[$edition] = $prototype;
    }
}
```

Diese Methode erwartet zwei Parameter:

1. Einen eindeutigen Namen des Sondermodells.
2. Eine Instanz des Typs `Vehicle`, die als Prototyp zum Erzeugen von Instanzen dieses Sondermodells verwendet werden kann.

Die Prototypen werden in einem assoziativen Array gespeichert, in dem der Name der Sonderedition als Key verwendet wird.

Als Nächstes fügen Sie Ihrer Fabrik nun noch eine Methode hinzu, die ein Sondermodell durch Kopieren des entsprechenden Prototyps erstellen kann. Dazu prüfen Sie, ob der übergebene Name in der Liste der Prototypen vorhanden ist, und werfen bei negativem Ergebnis eine Exception, um den Nutzer darüber zu informieren, dass das Sondermodell nicht existiert. Ansonsten erstellen Sie durch den `clone`-Operator eine Kopie des Prototyps und liefern diesen zurück.

```
namespace de\phpdesignpatterns\manufacturers;
use de\phpdesignpatterns\vehicles\Vehicle;

class SpecialEditionManufacturer {
    ... bisherige Properties und Methoden ...
    public function manufactureVehicle($edition) {
        if (!isset($this->prototypes[$edition])) {
            throw new UnknownSpecialEditionException('No prototype for special
                      edition ' . $edition . ' registered.');
        }
        return clone $this->prototypes[$edition];
    }
}
```

Diese Fabrik können Sie nun verwenden, um neue Autos des Sondermodells zu erstellen. Dazu erzeugen Sie eine Instanz der Fabrik sowie einen Protoyp für den Golf »Sonderedition Elvis Presley« und übergeben diesen an die Fabrik:

```
use de\phpdesignpatterns\manufacturers\SpecialEditionManufacturer;
use de\phpdesignpatterns\vehicles\Car;
use de\phpdesignpatterns\vehicles\Convertible;

$manufacturer = new SpecialEditionManufacturer();

$GolfElvis = new Car('VW', 'silber');
$GolfElvis->setAirConditioned(true);
$GolfElvis->setGraphics('Gitarre');
$manufacturer->addSpecialEdition('Golf Elvis Presley Edition', $GolfElvis);
```

Nun können Sie die Methode `manufactureVehicle()` nutzen, um weitere Instanzen desselben Sondermodells zu erstellen:

```
$golf1 = $manufacturer->manufactureVehicle('Golf Elvis Presley Edition');
echo "Typ        : ", get_class($golf1), "\n";
echo "Hersteller : {$golf1->getManufacturer()}\n";
echo "Farbe      : {$golf1->getColor()}\n";
echo "Grafiken   : {$golf1->getGraphics()}\n";
echo "Klima      : ", $golf1->hasAirCondition() ? "ja" : "nein", "\n";
```

Die Ausgabe dieses Skripts fällt wie erwartet aus:

```
Typ        : de\phpdesignpatterns\vehicles\Car
Hersteller : VW
Farbe      : silber
Grafiken   : Gitarre
Klima      : ja
```

Beim Erzeugen eines neuen Modells vom Typ »Sonderedition Elvis Presley« erhalten Sie einen silbernen Golf mit Klimaanlage und einer Gitarre als Verzierung zurück.

Die Standardimplementierung des `clone`-Operators erstellt nur eine Kopie des Objekts und übernimmt alle Eigenschaften. Wenn es sich bei den Eigenschaften des Objekts um weitere Objekte handelt, wird von diesem Objekt keine Kopie erstellt. Man spricht hier von einer *flachen Kopie*. PHP ermöglicht Ihnen, durch Implementieren des `__clone()`-Interzeptors das Klonen selbst durchzuführen. Dadurch ist es Ihnen möglich, auch *tiefe Kopien*, also Kopien aller referenzierten Objekte, zu erstellen. Mehr Informationen zum `__clone()`-Interzeptor entnehmen Sie Kapitel 1.

Um weitere Kundensegmente zu erschließen, wollen Sie zusätzlich noch einen Golf »Sonderedition Rolling Stones« anbieten. Dazu erstellen Sie zunächst eine prototypische Instanz des Sondermodells:

```
$GolfStones = new Convertible('VW', 'rot');
$GolfStones->setAirConditioned(false);
$GolfStones->setGraphics('Zunge');
```

Es handelt sich hierbei um einen roten Golf Cabrio ohne Klimaanlage (wozu auch, der Fahrer kann ja das Dach abnehmen) und natürlich mit der typischen Rolling-Stones-

Zunge als Verzierung. Dieses Sondermodell soll nun von derselben Fabrik erstellt werden können, die schon für die Herstellung der Elvis-Presley-Edition verantwortlich ist. Dazu registrieren Sie den Prototyp einfach unter dem gewünschten Namen:

```
$manufacturer->addSpecialEdition('Golf Rolling Stones Edition', $GolfStones);
```

Nun können Sie wie gewohnt neue Autos dieser Edition herstellen:

```
$golf2 = $manufacturer->manufactureVehicle('Golf Rolling Stones Edition');
echo "Typ        : ", get_class($golf2), "\n";
echo "Hersteller : {$golf2->getManufacturer()}\n";
echo "Farbe      : {$golf2->getColor()}\n";
echo "Grafiken   : {$golf2->getGraphics()}\n";
echo "Klima      : ", $golf2->hasAirCondition() ? "ja" : "nein", "\n";
```

Die folgende Ausgabe zeigt, dass die Fabrik auch dieses Auto wie gewünscht produziert hat.

```
Typ        : de\phpdesignpatterns\vehicles\Convertible
Hersteller : VW
Farbe      : rot
Grafiken   : Zunge
Klima      : nein
```

Die Fabrik kann also beliebige Objekte erstellen, solange diese das `Vehicle`-Interface implementieren und Sie ihr zuvor einen Prototyp übergeben, der als Basis für die Erzeugung neuer Instanzen dient.

Um sicherzugehen, dass Ihre Fabriken auch tatsächlich bei jedem Aufruf ein neues Auto erstellt, lassen Sie noch einen zweiten Golf »Sonderedition Elvis Presley« produzieren und prüfen, ob Sie dabei dasselbe Objekt zurückbekommen:

```
$golf3 = $manufacturer->manufactureVehicle('Golf Elvis Presley Edition');
if ($golf1 !== $golf3) {
    echo "\$golf1 ist nicht identisch mit \$golf3\n";
}
```

Wie erwartet, zeigt Ihnen die Ausgabe dieses Skripts, dass es sich bei den beiden Autos um unterschiedliche Instanzen handelt, sie können also gleichzeitig an unterschiedliche Kunden ausgeliehen werden:

```
$golf1 ist nicht identisch mit $golf3
```

Als Letztes prüfen Sie, wie die Fabrik reagiert, wenn Sie sie anweisen, ein Sondermodell zu produzieren, für das Sie zuvor keinen Prototyp registriert haben:

```
$golf4 = $manufacturer->manufactureVehicle('Golf Ray Charles Edition');
```

Da Sie in der `manufactureVehicle()`-Methode einen Check eingefügt haben, der im Prototypen-Array prüft, ob der Prototyp existiert, wird hier eine Exception geworfen, die den Klienten darüber informiert, dass die Fabrik nicht in der Lage ist, das Sondermodell zu produzieren:

```
PHP Fatal error:  Uncaught exception 'de\phpdesignpatterns\manufacturers\
UnknownSpecialEditionException' with message 'No prototype for special edition Golf Ray
Charles Edition registered.' in SpecialEditionManufacturer.php:19
Stack trace:
#0 test-special-edition.php(39): de\phpdesignpatterns\manufacturers\
SpecialEditionManufacturer->manufactureVehicle('Golf Ray Charle...')
#1 {main} thrown in SpecialEditionManufacturer.php on line 19
```

Das Prototype-Pattern hat Ihnen also geholfen, verschiedene Typen von Objekten zu erstellen, ohne dass Sie für jeden Typ eine eigene, konkrete Fabrik benötigt haben.

Definition des Patterns

Das Prototyp-Muster bestimmt die Arten der zu erzeugenden Objekte durch die Verwendung eines prototypischen Exemplars, das zur Erzeugung neuer Instanzen kopiert wird.

Um dieses Pattern in PHP anzuwenden, sind immer die folgenden Schritte nötig:

1. Implementieren Sie einen *Prototypenverwalter*, der alle Prototypen, die im System verfügbar sind, speichert und von außen zugreifbar macht. Dieser Verwalter muss die Möglichkeit bieten, neue Prototypen unter einem Namen abzulegen und mit demselben Namen wieder zugreifbar zu machen.
2. Implementieren Sie bei Bedarf eine `__clone()`-Methode in den Klassen, aus denen Ihre Prototypen erstellt werden. Dies ist nur nötig, wenn Ihre Prototypen weitere Objekte referenzieren, die beim Kopieren auch geklont werden müssen. Ist dies nicht der Fall, müssen Sie die `__clone()`-Methode nicht implementieren.
3. Erstellen und initialisieren Sie alle Prototypen, die im System verwendet werden können, und übergeben Sie diese an den Prototypenverwalter.

Abbildung 4-6 zeigt Ihnen noch einmal das Zusammenspiel der Klassen und Objekte in einem UML-Diagramm.

Konsequenzen

Der Einsatz des Prototype-Patterns versteckt die konkreten Implementierungen vor dem Klienten und reduziert dadurch dessen Abhängigkeit von den konkreten Implementierungen. Im Gegensatz zur abstrakten Fabrik ermöglicht das Prototyp-Muster weiterhin das Hinzufügen und Entfernen von Produkten zur Laufzeit dadurch, dass die verschiedenen Prototypen über einen Methodenaufruf beim Prototypenverwalter registriert werden können. Dadurch ist das Prototyp-Muster flexibler als die anderen Muster, die der Erzeugung von Objekten dienen.

Kombiniert mit anderen Mustern und der Komposition von Objekten zu komplexeren Strukturen, ermöglicht Ihnen das Prototyp-Muster, neue »Klassen« ohne Programmierung zu erstellen, da lediglich die einzelnen Eigenschaften eines Prototyps verändert werden, um das Verhalten der Objekts zu verändern.

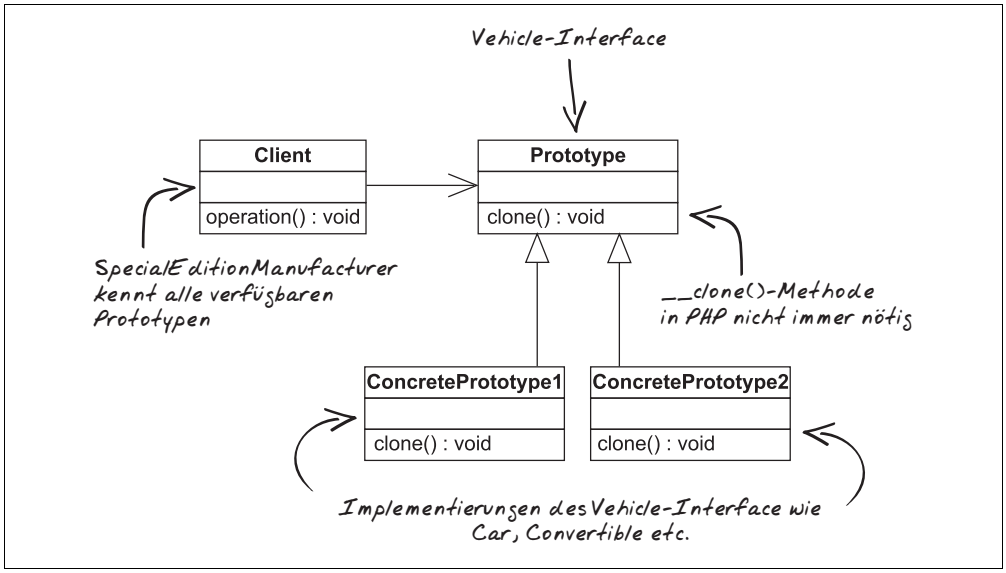

Abbildung 4-6: UML-Diagramm des Prototype-Patterns

Der Einsatz des Prototype-Patterns reduziert die Anzahl der Klassen, die Ihre Applikation benötigt, da Sie weniger Unterklassen bilden müssen, um die einzelnen Produkte zu erzeugen. Stattdessen werden alle Produkte auf Basis der Prototypen von einer Klasse erzeugt.

Fallstricke

Das Prototype-Pattern verlangt lediglich, dass alle Ihre Klassen, aus denen Prototypen gebildet werden sollen, geklont werden können. Dies erscheint auf den ersten Blick jedoch einfacher, als dies in der Realität oft der Fall ist. Solange Ihre Prototypen lediglich skalare Werte in ihren Eigenschaften speichern, reicht die Verwendung des `clone`-Operators aus, aggregieren Ihre Prototypen jedoch weitere Objekte, müssen Sie eine eigene `__clone()`-Methode implementieren, die sich um das Erstellen von Kopien für diese Objekte kümmert.

Im folgenden Beispiel speichert eine Instanz eines Autos nicht nur die Information, ob eine Klimaanlage vorhanden ist, sondern stattdessen direkt die Instanz dieser Klimaanlage. Für das Beispiel reicht eine konkrete Implementierung für alle Klimaanlagen aus, in einer echten Anwendung würden Sie hier sicher ein Interface sowie verschiedene konkrete Implementierungen einsetzen. Die Klimaanlage speichert im Beispiel nur die Gradzahl, auf die sie eingestellt ist:

```
namespace de\phpdesignpatterns\vehicles\addons;

class AirCondition {
    protected $degrees = 20;
```

```
    public function setDegrees($degrees) {
        $this->degrees = $degrees;
    }

    public function getDegrees() {
        return $this->degrees;
    }
}
```

Um die Klimaanlage zum Auto zu speichern, müssen Sie minimale Korrekturen an der Klasse `Car` vornehmen. So fügen Sie eine Eigenschaft `$airCondition` hinzu, die diese aufnehmen kann, und ergänzen eine Getter- und Setter-Methode:

```
namespace de\phpdesignpatterns\vehicles;

use de\phpdesignpatterns\vehicles\addons\AirCondition;

class Car implements Vehicle {
    ... andere Eigenschaften wie bisher ...
    protected $airCondition;

    ... andere Methoden wie bisher ...
    public function setAirCondition(AirCondition $airCondition) {
        $this->airCondition = $airCondition;
    }
    public function getAirCondition() {
        return $this->airCondition;
    }
}
```

Erstellen Sie nun einen Prototyp, der mit Klimaanlage ausgestattet ist, und erzeugen Sie darauf zwei neue Fahrzeuge:

```
use de\phpdesignpatterns\manufacturers\SpecialEditionManufacturer;
use de\phpdesignpatterns\vehicles\Car;
use de\phpdesignpatterns\vehicles\addons\AirCondition;

$manufacturer = new SpecialEditionManufacturer();

$GolfElvis = new Car('VW', 'silber');
$GolfElvis->setAirCondition(new AirCondition());
$GolfElvis->setGraphics('Gitarre');

$manufacturer->addSpecialEdition('Golf Elvis Presley Edition', $GolfElvis);

$golf1 = $manufacturer->manufactureVehicle('Golf Elvis Presley Edition');
$golf2 = $manufacturer->manufactureVehicle('Golf Elvis Presley Edition');
```

Wenn Sie nun den Wert ausgeben, auf den die Klimaanlage eingestellt ist, sollte dieser in beiden Autos 20 Grad betragen:

```
echo "Einstellung in \$golf1: ", $golf1->getAirCondition()->getDegrees(), "\n";
echo "Einstellung in \$golf2: ", $golf2->getAirCondition()->getDegrees(), "\n";
```

Die Ausgabe des Skripts bestätigt dies auch:

```
Einstellung in $golf1: 20
Einstellung in $golf2: 20
```

Ändern Sie nun die Einstellung für eines der beiden Objekte und geben Sie danach die Werte erneut aus:

```
$golf1->getAirCondition()->setDegrees(16);

echo "Einstellung in \$golf1: ", $golf1->getAirCondition()->getDegrees(), "\n";
echo "Einstellung in \$golf2: ", $golf2->getAirCondition()->getDegrees(), "\n";
```

Wie Sie sehen können, haben Sie nicht nur die Temperatur des ersten Golfs geändert, sondern auch gleichzeitig die des zweiten Fahrzeugs:

```
Einstellung in $golf1: 16
Einstellung in $golf2: 16
```

Das Problem ist, dass es sich bei den beiden Objekten in `$golf1` und `$golf2` zwar um zwei unterschiedliche Objekte handelt, aber beide auf dieselbe Instanz der `AirCondition`-Klasse referenzieren. Bestätigt wird diese Vermutung, wenn Sie die Objekt-Hashes der Klimaanlagen ausgeben lassen:

```
echo spl_object_hash($golf1->getAirCondition()) . "\n";
echo spl_object_hash($golf2->getAirCondition()) . "\n";
```

Die beiden Hashes sind identisch und bezeichnen somit auch dasselbe Objekt. Abgesehen davon, dass es der Klimaanlage physikalisch nicht möglich sein wird, in zwei Autos gleichzeitig eingebaut zu sein, ist dies auch nicht das gewünschte Verhalten für Ihre Anwendung. Stattdessen sollte jede Kopie des Prototyps auch eine eigene Klimaanlage referenzieren. Dazu müssen Sie die `__clone()`-Methode in der Klasse `Car` implementieren:

```
namespace de\phpdesignpatterns\vehicles;
class Car implements Vehicle {
    ... bisherige Eigenschaften und Methoden ...
    public function __clone() {
        $this->setAirCondition(clone $this->getAirCondition());
    }
}
```

Die `__clone()`-Methode wird von PHP auf der Kopie des Objekts aufgerufen, nachdem diese erzeugt wurde. Alles was Sie dort noch machen müssen, ist also, eine Kopie der Klimaanlage zu erstellen und diese im neuen Objekt zu referenzieren. Dazu verwenden Sie die Setter- und Getter-Methode, um die Kapselung nicht zu durchbrechen. Durch Verwendung des `clone`-Operators würde PHP auch die `__clone()`-Methode der Klasse `AirCondition` aufrufen, sofern Sie diese implementiert hätten.

Somit erstellen Sie nun statt einer *flachen Kopie (shallow copy)* eine *tiefe Kopie (deep copy)* und stellen sicher, dass die beiden Objektbäume strikt voneinander getrennt sind und sich keine Objekte teilen.

In der Realität müssen Sie bei jeder Eigenschaft Ihrer Klassen genau abwägen, ob von einer Eigenschaft eine Kopie erstellt werden soll oder nicht. Die korrekte Implementierung der `__clone()`-Methode ist also der schwierigste Schritt bei der Implementierung des Prototype-Patterns.

Übersicht über die Erzeugungsmuster

Nachdem Sie verschiedene Patterns, die sich mit der Erzeugung von Objekten befassen, in diesem Kapitel kennengelernt haben, finden Sie in Tabelle 4-1 noch einmal eine kurze Übersicht über die Patterns:

Tabelle 4-1: Überblick über die erzeugenden Patterns

Pattern	Zweck	Konsequenzen
Singleton	Stellt sicher, dass von einer Klasse nur eine Instanz existiert, und stellt einen globalen Zugriffspunkt für diese Instanz zur Verfügung.	Ermöglicht Zugriffskontrolle auf die Instanz. Reduziert die Verwendung globaler Variablen. Verhindert, dass mehr als eine Instanz erzeugt werden kann.
Factory-Method (Fabrikmethode)	Delegiert die Erzeugung von Objekten an Unterklassen.	Ermöglicht das Einfügen spezialisierter Klassen. Fördert die Programmierung gegen Schnittstellen. Die Anzahl der Klassen in einer Applikation wird erhöht.
Abstract-Factory (abstrakte Fabrik)	Erzeugt Familien verwandter Objekte.	Stellt sicher, dass Objekte nur zusammen mit den Objekten verwendet werden, mit denen diese kompatibel sind. Das Hinzufügen neuer Produkte ist aufwendig.
Prototype (Prototyp)	Erzeugt Objekte durch Kopieren eines prototypischen Exemplars.	Ermöglicht das Hinzufügen von Produkten zur Laufzeit. Reduziert die Anzahl der benötigten Unterklassen.