Factory Patterns

und deren Auswirkung auf die Softwarearchitektur in der Praxis

Klaus Kusche, Juni 2013

Inhalt

- Was ist das?
- Warum braucht man das?
- Was bringt das?
- Wann hilft es noch?
- Realistisches Beispiel

“Factory Patterns”

Ein **Factory Pattern** ist ein

Design Pattern

(= *Lösungs- bzw. Entwurfs-Muster*
für eine häufig vorkommende Art
von Problemstellungen)

von der ***“Gang of Four”:***

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
“Design Patterns. Elements of Reusable Object-Oriented Software”

Ziel aller dieser Patterns:

Code *universeller / flexibler / ...* zu machen!

Was heißt "*Factory*"?

Ganz grob:

Ein *"Ding"*
zur ***Erzeugung neuer Objekte***
(im OO-Sinn)

statt **new** + *Konstruktor*

Meistens: Zur Erzeugung von Objekten …

- *nicht* einer bestimmten, *fixen Klasse*,
- *sondern* einer *zur Laufzeit wählbaren Klasse*
  aus einer *ganzen Hierarchie von Klassen*
  (voneinander abgeleitet)

Was ist das Problem? (1)

Konstruktoren können ***nicht virtuell*** sein!

==> Generischer Code kann
nur auf schon übergebenen Objekten arbeiten,
aber *nicht "generisch" neue Objekte anlegen*!

==> *Eigener Code-Pfad pro Klasse* mit explizitem Aufruf
des jeweiligen *konkreten Konstruktors* nötig!

==> Code kann nur Objekte von Klassen anlegen,
die *zur Compile-Zeit schon definiert* sind,
aber *nicht von "zukünftigen"* (abgeleiteten) Klassen.

==> *Explizite Compile- und Link-Abhängigkeit*
von allen Klassen, mit denen der Code arbeiten soll!

Was ist das Problem? (2)

==> ***Frameworks***, *Libraries* und
allgemeine / generische Applikationslogik

müssten

- erst *nach den konkreten Datenklassen* entwickelt
  (im Besonderen: *Frameworks nach der Applikation!*)
- und *bei jeder neu dazukommenden Datenklasse*
  im Source geändert und frisch compiliert werden!

Erste Idee ...

Konstruktoren in (virtuelle!) Methode verpacken!

Einfachste Problemstellung:

Neues Objekt als *Kopie eines bestehenden Objektes* (aus einer Klassen-Hierarchie)

==> "**clone**-*Trick*": **`newObjP = origObjP->clone();`**

In der **Basisklasse**:

```
virtual Vehicle* clone() const;
```

In allen **abgeleiteten Klassen**:

```
virtual Truck* clone() const {
   return new Truck(*this);    // Aufruf Copy-Konstruktor
}
```

Ist `clone` schon eine Factory?

Nach den meisten Definitionen: **Nein!**

Eine “echte” Factory-Methode ist
nicht Methode der *Klasse der Objekte* selbst,
sondern einer ***eigenen Klasse*** *nur für diesen Zweck!*

==> *Wir haben:*

- Eine eigene **Factory-Klasse** (mit nur ***einer*** Methode!)
- Ein **Factory-Objekt** dieser Klasse (meist ein ***Singleton!***)
- Die **Factory-Methode** (mit Konstruktor-Aufruf):

Wird für ein Factory-Objekt aufgerufen,
liefert einen Pointer auf ein neu angelegtes Objekt.

Ist das schon das Factory Pattern?

Nein, *noch nicht ganz!*

Wir wollen wahlweise
Objekte mehrerer Klassen erzeugen können

... und zwar basierend auf ***Ableitung***:
"... but let subclasses decide
which class to instanciate ..." (GoF)

==> ***Pro zu erzeugender Klasse eine Factory-Klasse!***

==> ***Ableitungshierarchie der Factory-Klassen***
(ist oft *strukturell gleich* zur
Ableitungshierarchie der Objekt-Klassen)

... mit einer (meist *abstrakten*) ***Factory-Vaterklasse***

Beispiel Factory-Hierarchie

`Vehicle`

- **`Motorbike`**
- **`Pickup`**
- **`Cab`**
- **`Truck`**
  - **`Bus`**
  - **`RoadTrain`**

`VehicleFactory`

- **`MotorbikeFactory`**
- **`PickupFactory`**
- **`CabFactory`**
- **`TruckFactory`**
  - **`BusFactory`**
  - **`RoadTrainFactory`**

... jede mit
`createVehicle()`

Mehrfache Objekt-Hierarchien

Factory-Klassen enthalten oft

mehrere *Factory-Methoden*

für ***mehrere*** *parallele Objektklassen-Hierarchien*

Beispiel in **`VehicleFactory`**:

- **`createVehicle()`**
- **`createVehicleTax()`**
- **`createVehicleInsurance()`**

für Objektklassen-Hierarchien

- **`Vehicle`**, **`VehicleTax`**, **`VehicleInsurance`**

Nutzen (1)

Der **Framework- oder Applikationslogik-Code**

- ... *referenziert* nur mehr die beiden
  abstrakten Objekt- und Factory-Vaterklassen
- ... *erzeugt* alle Objekte
  unter *Verwendung eines Factory-Objekts*
- ... kann durch *"Übergabe"* *(siehe später)*
  geeigneter *konkreter Factory-Objekte* mit
  beliebigen abgeleiteten Objektklassen arbeiten

==> ***Keine konkreten Daten-Klassen*** mehr im Code
(und auch ***keine konkreten Factory-Klassen***!),
keine Abhängigkeit zur Compile- und Link-Zeit!

Nutzen (2)

*Das Factory Pattern **entkoppelt***
die Anwendungs- oder Framework-Logik
von den konkreten Klassen der Daten-Objekte!

==> Der Logik-Code kann unverändert
für ***nachträglich*** *dazugeschriebe Klassen*
verwendet werden!

Pro *neuer Objekt-Klasse*:

- *Neue Factory-Klasse* dafür ableiten...
- ... und an einer *einzigen Codestelle* (z.B. `main`, `Init`):
Ein *Objekt der neuen Factory* erzeugen.

Nutzen (3)

Im **Extremfall**:

- Neue Klassen sogar *dynamisch zur Laufzeit ladbar!*
- Objekte versch. Klassen *input-abhängig erzeugbar!*

Beispiel:

Dynamisch erstellte *Collection* (z.B. **`map<...>`**)
aller Klassen-Bezeichnungen samt *Factory-Objekten*

==> Eingabe einer Klassen-Bezeichnung

==> Suche des korrespondierenden Factory-Objektes

==> Abarbeitung der "Core Logic"
mit den davon erzeugten Daten-Objekten

Anwendung für Tests (1)

Problemstellung:

Unit-Tests der Klasse **A**
arbeiten mit Daten-Objekten der Klasse **B**

==> Klasse **B** sollte im Test von Klasse **A**
durch *Stubs / Mock-Objekte* ersetzt werden!

==> *Code-Modifikation* & Recompile von **A**
nur für den Test nötig (überall **MockB** statt **B**)!

Unerwünscht!

(Analoges Problem z.B. bei *"Test Driven Development"*
für noch nicht vorhandene Klassen...)

Anwendung für Tests (2)

Lösung:

- Alle Objekte von **B** im Code von **A** mittels *Factory* erzeugen.
- Im *Produktiveinsatz*: Factory für **B**
  Im *Test*: Factory für **MockB**
- Factory z.B.
  - als *Parameter* an A übergeben
  - oder *mittels globaler Funktion ermitteln* (zwei "**`getCurrentFactory()`**" in Appl. / TestMain)
  - oder in *virtueller Methode* von **A** erzeugen, diese in *abgeleiteter Klasse* **`TestA`** *überschreiben*

Andere Factory-Anwendungen (1)

Factory-Methoden sind *frei benennbar*,
Konstruktoren haben *fix den Klassennamen*:

- Factories sind ev. *besser lesbar*: "**`create`**..."
- Factories helfen gegen *zu viel Konstruktor-Overload*:

```
color1 = ColorFromRGB(r, g, b);
color2 = ColorFromHSV(h, s, v);
```

(statt zwei Mal **`Color(...)`** mit *identen* Parametern???)

Andere Factory-Anwendungen (2)

Bei Erzeugung von Objekten, die
- von einem anderen Objekt *abhängig* sind
- oder einem andern Objekt untergeordnet sind:

- Factory-Aufrufe *lesen sich besser*
  "von links nach rechts"
  als Konstruktoren mit Parameter.
- Die Initialisierungslogik gehört eher
  *zur "**übergeordneten**" Klasse!*

```
DBConnection *conn;
DBTransaction *trans;
trans = new DBTransaction(conn);   // klassisch
trans = conn.createTrans();        // Factory
```

Andere Factory-Anwendungen (3)

- Wenn die Objekt-Erzeugung
  viel Logik oder *viele **permanente** Daten* erfordert

  ==› Aus Objekt-Klasse *auslagern*!

  ==› ***Daten in das Factory-Objekt*** (Singleton!)
  statt statisch in der Objekt-Klasse speichern!

- Bei *ähnlichem Initialisierungscode*
  in vielen "verwandten" Klassen / Konstruktoren:

  Faktorisierung gleicher Codeteile
  gelingt aus Factory-Methoden
  leichter als aus Konstruktoren!

Beispiel (1)

***Portables* GUI-Framework**

(Klassenhierarchie für Button, Textfeld, Menü, ...)

Darunter zur *realen Anzeige* der GUI-Elemente:

Wie in Java: ***"Peer-Klassen"***

Parallel je *ein Satz von Klassen* für Linux, Mac, Win

mit gemeinsamen abstrakten Vaterklassen

(z.B. **`ButtonPeer`**: **`GtkButtonPeer`**, **`WinButtonPeer`**, ...)

Problem:

Wie *erzeugen* die portablen Klassen (z.B. **`Button`**)

die *dazugehörigen Peer-Objekte* (z.B. **`GtkButtonPeer`**)

ohne **`switch`** über alle Plattformen und

ohne Compile-Time-Abhängigkeit von allen Peers?

Beispiel (2)

Lösung:

- *Abstrakte Factory-Klasse* **PeerFactory**
  mit einer *Factory-Methode pro GUI-Peer-Klasse*:

  createButtonPeer(), **createMenuPeer()**, ...
- *Davon abgeleitete Factory-Klassen* pro Plattform:

  GtkPeerFactory, **AquaPeerFactory**, ...
- Feststellen der richtigen Peer-Factory *zur Laufzeit*
  `PeerFactory *pFact = getCurrentPlatform();`
- Erzeugen der "richtigen" Peers im portablen Code:
  `ButtonPeer *bPeer = pFact->createButtonPeer();`

“The end”

Fragen?