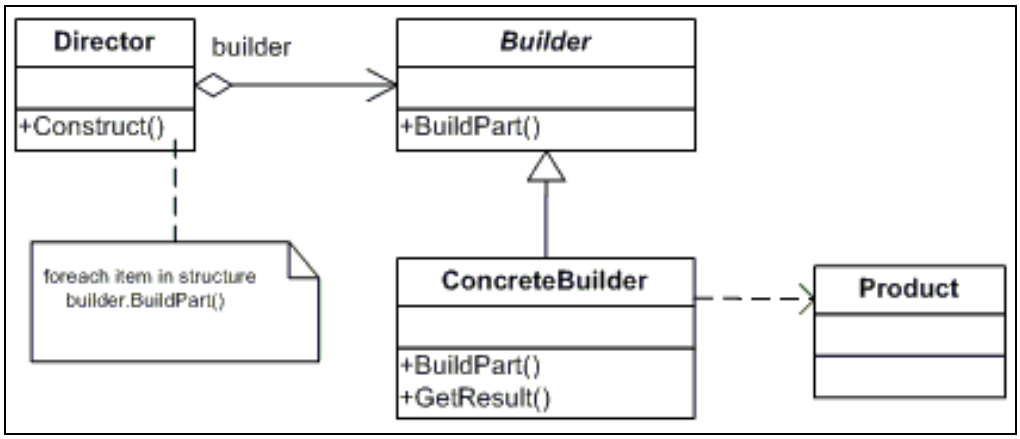
Builder

1 Builder

Das Designpattern Builder zählt zu den Erzeugungsmustern.
Man kann es dann sinnvoll einsetzen, wenn man Objekte erzeugen möchte, die viele Attribute haben, wobei einige verpflichtend, andere aber optional sind. D.h. man bräuchte viele verschiedene Konstruktoren.
Das Pattern gibt es in einer Standardversion und in einer vereinfachten Version, wobei mir vor allem letztere am häufigsten einsetzbar erscheint.

1.1 UML

1.1.1 Standard



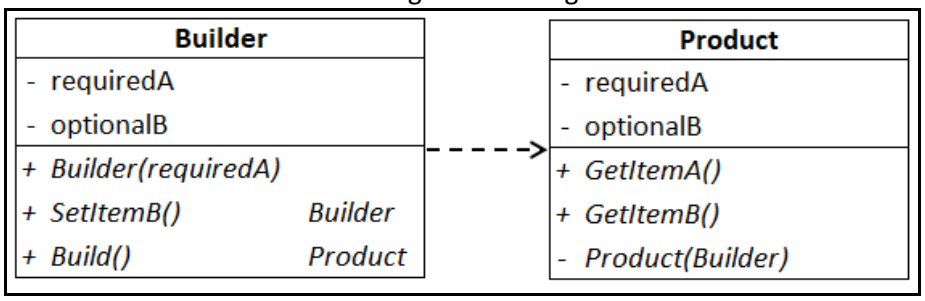
Hinweise:

- **`Builder`** definiert die Schnittstelle, um alle Teile eines Produkts erstellen zu können.
- **`ConcreteBuilder`** erstellt ein bestimmtes Produkt
- **`Product`** ist das zu erstellende Produkt.
- **`Director`** erstellt das Produkt mittels einer konkreten Builder-Instanz.

Diese Variante möchte ich nicht weiter besprechen.

1.1.2 Fluent

Diese Variante braucht man meines Erachtens relativ oft. Man kapselt die Erstellung des Produkts durch einen Builder. Entsprechend ist der Konstruktor auch private. Das Produkt selber soll unveränderlich sein.
Der Builder kann dann auch etwaige Validierungen durchführen.



Hinweise:

- **`Product`**
  - Das Produkt hat einige verpflichtende und einige optionale Felder
  - Das Produkt ist unveränderlich und hat daher nur Getter, aber keine Setter
  - Der Konstruktor des Produkts ist private – es kann also ausschließlich über den Builder erzeugt werden.
- **`Builder`**
  - Der Builder hat dieselben Felder wie das Produkt.
  - Er ist eine innere Klasse von Product (und hat damit Zugriff auf den privaten Konstruktor)

  - Über Setter können diese gesetzt werden. Dabei wird der Builder zurückgegeben – Method Chaining.
  - Die verpflichtenden Felder werden im Konstruktor gesetzt
  - Es gibt eine public Methode, mit der das Produkt letztendlich erzeugt wird.
- Das Product mit Autoproperties auszustatten ist oft keine Lösung, weil man damit
  - nicht zw. verpflichtenden und optionalen Feldern unterscheiden kann
  - das Product auch nachträglich noch verändern kann

1.2 Validierungen

Etwaige Validierungen können in der Methode Build() vorgenommen werden. Aus Gründen der Threadsicherheit muss das nach Erstellen des Produkts erfolgen.
Falsch:

```
public Product Build()
{
  if (optionalD == null) throw new InvalidOperationException("Hoppala");
  var product = new Product(this);
  return product;
}
```

Richtig:

```
public Product Build()
{
  var product = new Product(this);
  if (product.OptionalD == null) throw new InvalidOperationException("Hoppala");
  return product;
}
```

1.3 Erkennungsmerkmale

An folgenden Merkmalen kann man erkennen, dass der Einsatz eines Builders sinnvoll ist:

- Das zu erstellende Objekt hat viele Felder, wobei einige davon optional sind. Man hätte daher viele verschiedene Konstruktoren. Als Faustregel kann man eine Feldanzahl von 4 als Indikator für den Builder annehmen.
- Das Objekt soll nach der Erstellung unveränderlich sein.
- Man muss viele gleiche oder ähnliche Objekte erzeugen.

1.4 Beispiele

In folgenden Beispielen würde sich ein Builder anbieten:

- Objekte aus CSV-Datei erzeugen
- StringBuilder in Java und C#
- Android: AlertDialog.Builder, Notification.Builder, GsonBuilder, LatLngBuilder,…

```
new AlertDialog.Builder(this)
        .setMessage("Das ist meine MessageBox")
        .setNeutralButton("Ok", null)
        .show();
```

```
final LatLngBounds bounds = new LatLngBounds.Builder()
        .include(new LatLng(48, 14))
        .include(new LatLng(48.1, 14.05))
        .include(new LatLng(48.2, 14.03))
        .include(new LatLng(48.14, 15.1))
        .build();
```

1.5 Vor-/Nachteile

1.5.1 Vorteile

- Man erspart sich bei einer Klasse mit vielen Feldern die Programmierung vieler Konstruktoren mit unterschiedlicher Parameteranzahl und -reihenfolge.
- Reihenfolge der Parameter ist beliebig – man hat ja pro Parameter einen Setter
- Man kann den gleichen Builder für die Erzeugung vieler (ähnlicher) Objekte wiederverwenden.
- Validierungen können im Build() noch vorgenommen werden.
- Man kann den Builder anderen Methoden bzw. Objekten als Parameter übergeben, womit diese dann neue Produkte erzeugen können, ohne Details kennen zu müssen.
- Die Lesbarkeit des Codes für die Produkterzeugung wird erhöht – man hat ja sprechende Setter-Methoden.

1.5.2 Nachteile

- Builder und Product duplizieren praktisch die Felder.