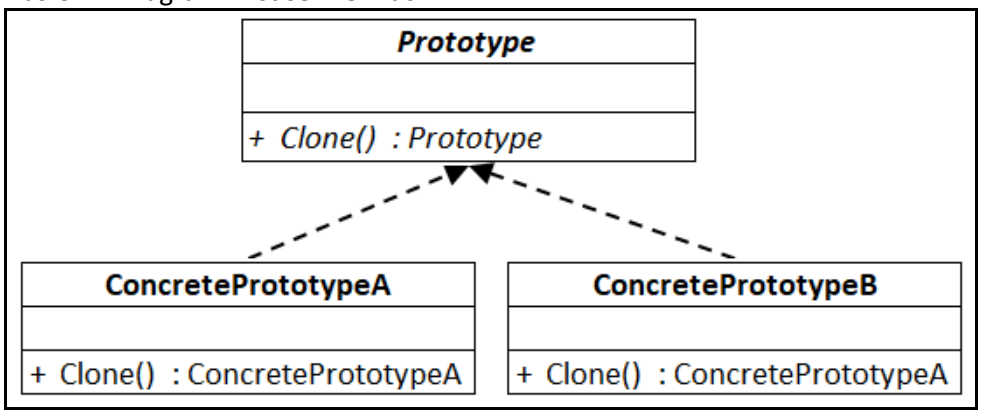
# *Prototype*

# 1 Prototype

Dieses Designpattern zählt ebenfalls zu den Erzeugungsmustern. Dabei wird ein Objekt als Vorlage für spätere Serienfertigung verwendet.
Es ist immer dann einsetzbar, wenn Objekte durch Klonen schneller bzw. effizienter erzeugt werden können als durch wiederholtes Erzeugen durch Konstruktion.

## 1.1 UML

Das UML-Diagramm ist sehr einfach.




Hinweise:

## 1.2 Erkennungsmerkmale

Bei folgenden Aufgabenstellungen macht der Prototyp Sinn:

- Man braucht eine Vielzahl von Objekten, deren **Erzeugung zeitintensiv** ist und ein Klonen schneller geht.
- Einzelne Instanzen weisen nur geringe Unterschiede auf. Durch Klonen eines Objekts und Anpassen der Unterschiede kommt man evtl. schneller ans Ziel.
- Man bräuchte viele Unterklassen, die sich nur wenig unterscheiden. Z.B. bei Problemen, bei denen jeder Kunde eine minimal andere Konfiguration benötigt.

## 1.3 Beispiele

In folgenden Beispielen würde sich ein Prototyp anbieten:

- Ein Objekt ist mit einer **Default-Konfiguration** als Datei vorhanden. Diese Datei wird einmalig als Prototyp geladen und daraus dann die einzelnen Objekte erzeugt und **angepasst**.
- Dienstplan: Es gibt für verschiedene Dienststellen Standarddienstpläne, die für jeden Mitarbeiter dann minimal angepasst werden.
- Objekte werden erzeugt, indem sie von der Datenbank gelesen oder über eine (möglicherweise langsame) Internetverbindung geladen werden. In diesem Fall kann das Klonen deutliche Performancesteigerungen ergeben.

## 1.4 Vor-/Nachteile

### 1.4.1 Vorteile

- Man kann viele schwergewichtige Objekte sehr schnell erzeugen
- Man erspart sich oftmals die Erstellung vieler ähnlicher Unterklassen
- Erhält man den Prototyp aus einer Datei (JSON, XML), kann sich der Prototyp jederzeit ändern, ohne dass der weitere Programmcode davon etwas wissen muss.
- Hat man mehrere Prototypen in einem Dictionary, kann man ein geklontes und angepasstes Objekt ganz leicht als zusätzlichen Prototyp registrieren.

### 1.4.2 Nachteile

- Durch Klonen werden möglicherweise auch Unterobjekte neu erzeugt, obwohl das nicht erwünscht ist.

## 1.5 Klonen

Dieses Pattern läuft eigentlich auf genau eine Methode hinaus, nämlich das Klonen eines Objekts.
In C# kann man das so lösen, dass man das Objekt serialisiert und wieder einliest. Damit man nicht sinnloserweise eine Datei anlegen muss, schreibt man die Serialisierung üblicherweise in einen **`MemoryStream`**.
Die Objekte müssen dabei mit dem Attribut **`[Serializable]`** markiert werden.

```
[Serializable]
4 references
class Prototype
{
  1 reference
  internal Prototype Clone()
  {
    var stream = new MemoryStream();
    var formatter = new BinaryFormatter();
    formatter.Serialize(stream, this);
    stream.Seek(0, SeekOrigin.Begin);
    var copy = (Prototype)formatter.Deserialize(stream);
    stream.Close();
    return copy;
  }
}
```

### 1.5.1 Copy vs. Deep Copy

Obiges Klonen ist eine tiefe Kopie (**deep copy**).
Im Gegensatz dazu wird bei einer flachen Kopie (**memberwise clone**) nur jede Instanzvariable kopiert. D.h. ist eine Instanzvariable ein Zeiger auf z.B. eine Liste, wird die Liste selbst nicht kopiert, sondern nur der Zeiger darauf.
Je nach Anwendung kann das gewünscht sein, oder aber auch nicht.