Eine erneute Anwendung des Funktors erzeugt eine neue Struktur:

```
module MalEnum = struct
    type enum = int
    let null = 1
    let incr x = 2*x
    let int_of_enum x = x
  end

# module MalCounter = GenCounter (MalEnum);;
module MalCounter : Counter

# MalCounter.incCounter();
  MalCounter.incCounter();
  MalCounter.int_of_counter (MalCounter.getCounter());;
- : int = 4
```

Mit einem Funktor kann man sich auch mehrere Instanzen des gleichen Moduls erzeugen:

```
# module CountApples = GenCounter (PlusEnum);;
module CountApples : Counter
# module CountPears = GenCounter (PlusEnum);;
module CountPears : Counter
...
```

```
...
# CountApples.incCounter();;
- : unit = ()
# CountApples.incCounter();;
- : unit = ()
# CountPears.incCounter ();;
- : unit = ()
# CountApples.int_of_counter (CountApples.getCounter());;
- : int = 2
# CountPears.int_of_counter (CountPears.getCounter());;
- : int = 1
```

7.5 Getrennte Übersetzung

- Eigentlich möchte man Ocaml-Programme nicht immer in der interaktiven Umgebung starten :-)
- Dazu gibt es u.a. den Compiler `ocamlc` ...

```
> ocamlc Test.ml
```

  interpretiert den Inhalt der Datei `Test.ml` als Folge von Definitionen einer Struktur `Test`.
- Als Ergebnis der Übersetzung liefert `ocamlc` die Dateien:

`Test.cmo`	Bytecode für die Struktur
`Test.cmi`	Bytecode für das Interface
`a.out`	lauffähiges Programm

- Gibt es eine Datei `Test.mli` wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:

```
> ocamlc Test.mli Test.ml
```

- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:

```
> ocamlc B.mli B.ml A.mli A.ml
```

- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:

```
> ocamlc B.cmo A.mli A.ml
```

- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet Linux das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)