GoF-Design-Patterns in Embedded Systemen?

Karl Nieratschker, SKT Nieratschker

Die von Gamma und seinen Mitautoren beschriebenen Design Patterns gehören heute zu den Grundlagen eines modernen Softwareentwurfs. Ihre Anwendung kostet i.d.R. allerdings Speicherplatz und/oder CPU-Laufzeit, weshalb sie oft in Embedded Systemen nicht eingesetzt werden. Dieser Beitrag beschäftigt sich mit dem Nutzen und den Kosten von Entwurfsmustern und zeigt, unter welchen Voraussetzungen man sie auch in Embedded Systemen einsetzen kann.

Der Begriff „Design Pattern“ („Entwurfsmuster“) wurde Ende der 1970er Jahre von Christopher Alexander geprägt. Als Professor für Architektur beschäftigte er sich intensiv mit dem Entwurf von Gebäuden und schuf wichtige Grundlagen für die qualitative Beurteilung von Designs. Diese Arbeit hat auch die Softwareentwicklung beeinflusst und führte 1995 zu der Veröffentlichung des Buches **Design Patterns: Elements of Reusable Object-Oriented Software** der Autoren E. Gamma, R. Helm, R.Johnson und J. Vlissides, die aufgrund ihres Erfolges bald als „Gang of Four“ (GoF) bezeichnet wurden.

Definition

In dem Standardwerk werden 23 Entwurfsmuster beschrieben. Jedes Pattern wird durch einen charakteristischen Namen identifiziert und beschreibt, wie ein häufig vorkommendes Problem in einem spezifischen Umfeld (Kontext) gelöst werden kann. Die Lösung selbst ist generisch gehalten, und muss deshalb auf den konkreten Anwendungsfall angepasst werden.

Vorteile

Einer der größten Vorteile der Entwurfsmuster besteht darin, häufig wiederkehrende Aufgabenstellungen mit bewährten und in Bezug auf Flexibilität, Erweiterbarkeit und Wartbarkeit hochwertigen Lösungskonzepten realisieren zu können, ohne diese immer wieder neu entwickeln zu müssen. Dies schafft Freiräume für die Lösung der applikationsspezifischen Probleme, die mit jeder Anwendung ohnehin neu gelöst werden müssen. Darüber hinaus vereinfachen Design Patterns auch die Kommunikation Teammitglieder während eines Entwurfs.

Design Patterns nutzen intensiv die Möglichkeiten der objektorientierten Programmierung, um ihre Leistungsfähigkeit zur Geltung zu bringen. Das heißt aber nicht, dass sie nur zusammen mit objektorientierten Programmiersprachen eingesetzt werden können. Objektorientierte Programmierung ist eine

Programmiertechnik, die zwar von objektorientierten Sprachen direkt unterstützt wird, die prinzipiell aber in jeder Sprache angewendet werden kann. Ein wesentliches Prinzip der Patterns liegt darin, die Dinge, die leicht geändert werden können sollen, in eigene Klassen auszulagern und somit die Zuständigkeiten auf mehr Klassen zu verteilen. Diese Klassen können dann je nach Bedarf instanziiert werden und bieten so die gewünschte Flexibilität.

Beispiel

Ein Muster, das häufig eingesetzt wird, ist die Abstrakte Fabrik (**Bild 1**).

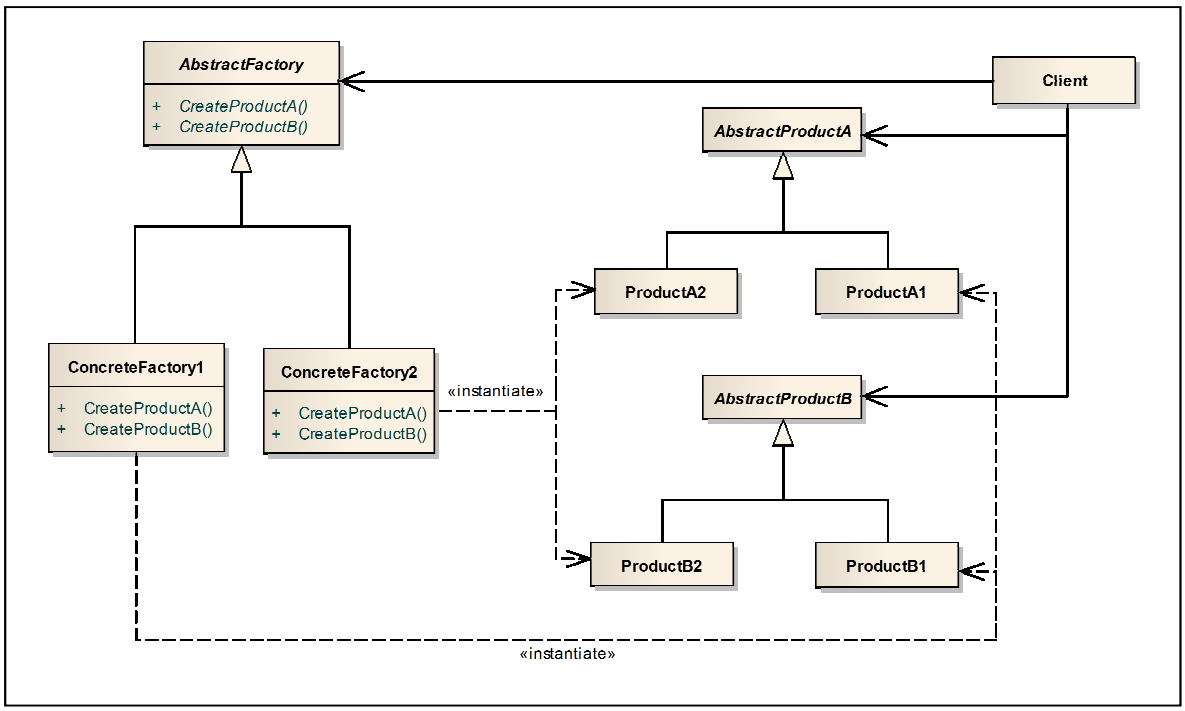


Bild 1: Entwurfsmuster Abstrakte Fabrik

Die Idee dieses Musters besteht darin, die Instanzen einer Familie von zusammenarbeitenden Klassen nicht direkt zu erzeugen, sondern durch ein nur für diesen Zweck angelegtes Fabrikobjekt erzeugen zu lassen. Der Vorteil dieser zunächst überflüssig erscheinenden Indirektionsstufe besteht darin, dass man das Verhalten des Systems ganz einfach verändern kann, indem man das Fabrikobjekt durch ein anderes ersetzt, das Instanzen von anderen Klassen erzeugt, die sich dann natürlich auch anders verhalten. Um eine Anpassung des Applikationscodes, der die erzeugten Objekte benutzen soll, zu vermeiden, müssen die austauschbaren Klassen eine gemeinsame Schnittstelle (Interface) besitzen, über die der anwendende Code das Objekt benutzen kann, ohne zu wissen, von welchem konkreten Typ das Objekt tatsächlich ist. Eine andere Anwendung des Fabrikmusters, die speziell für die Embedded-Programmierung interessant sein kann, besteht darin, dass der Erzeugungsprozess in Abhängigkeit der Verfügbarkeit von Hardwaregeräten, Speicher oder Laufzeitaspekten flexibel gesteuert werden kann.

Einarbeitung und Anwendung

Die Beschreibung der Design Patterns folgt einem festen Schema, das Dinge wie Name, Zweck, Struktur, Konsequenzen und Implementierungshinweise beinhaltet. Als Nachschlagewerk ist diese Form sehr nützlich, für die Einarbeitung allerdings weniger. Ein weiteres Problem ist, dass die Beispiele des Standardwerks nicht aus dem Umfeld der Embedded Systeme stammen und deshalb die Anwendbarkeit für diesen Bereich nicht unmittelbar ersichtlich ist. Generell ist die Umsetzung oft nicht so einfach, wie man das vielleicht erwarten könnte, denn es muss eine ganze Reihe von Dingen geklärt werden, wie z.B.

- **wo** ein Pattern eingesetzt werden kann,
- **welches** Pattern verwendet werden sollte,
- **wie** das gefundene Pattern angewendet werden soll und
- **ob** der Patterneinsatz gerechtfertigt ist in Bezug auf
    - Designkomplexität („Over-Engineering“) und
    - Ressourcenverbrauch (Speicher, Laufzeit)

C++ in Embedded Systemen

Gerade der letztgenannte Punkt ist für Embedded Systeme oft kritisch, denn hier sind die verfügbaren Ressourcen wie Speicherplatz oder CPU-Laufzeit oft sehr begrenzt. Embedded Systeme werden auch heute noch häufig in C programmiert, denn C++ steht im Ruf, nicht effizient genug zu sein. Obwohl man grundsätzlich in jeder Programmiersprache objektorientiert programmieren kann, ist die Verwendung einer objektorientierten Sprache unbedingt empfehlenswert, denn die Anwendung der objektorientierten Mechanismen ohne Sprachunterstützung ist in der Regel nicht trivial. Deshalb lohnt es sich, einen Blick auf die Speicherplatz- und Laufzeiteffizienz von C++ zu werfen.

Eine C++-Klasse mit ihren Methoden ist vergleichbar mit einer C-Struktur und C-Funktionen, die auf die Elemente der Struktur zugreifen. Grundsätzlich besteht zwischen diesen beiden Lösungen kein Unterschied in Bezug auf Speicherplatz- und Laufzeitbedarf, wobei C++ bei der Verwendung von inline-Funktionen mitunter sogar Vorteile hat (**Bild 2**).

C

```
typedef struct _Counter
{
  char* name;
  int   count;
} Counter;


void incCounter(Counter* p)
        { ++p->count; }

void decCounter(Counter* p)
        { --p->count; }

void printCounter(Counter* p)
        { printf("Counter %s: %d\n",
              p->name, p->count); }

int main()
{
  Counter c1 = {"c1", 0};
  incCounter(&c1);
  printCounter(&c1);
  printf("%d\n",sizeof(c1));
}
```

Speicherplatzbedarf:	**108 Bytes**
Ausführungszeit (ohne print):	**16 Zyklen**

C++

```
class Counter
{
  char* name;
  int   count;
public:
  Counter(char* n, int iv)
    { name = n; count = iv; }
  void inc()
    { ++count; }
  void dec()
    { --count; }
  void print()
    { printf("Counter %s: %d\n",
        name, count); }
};


int main()
{
  Counter c1("c1", 0);
  c1.inc();
  c1.print();
  printf("%d\n",sizeof(c1));
}
```

Speicherplatzbedarf:	**56 Bytes**
Ausführungszeit (ohne print):	**2 Zyklen**

Bild 2: Beispiel: Vergleich C-Struktur / C++-Klasse mit inline-Methoden

Auch die Vererbung ist unkritisch, wenn die Konstruktoren der Vererbungshierarchie inline sind. Einer der wichtigsten Mechanismen, die von Design Patterns benötigt werden, sind virtuelle Funktionen. Für jede Klasse, die solche Funktionen besitzt, wird eine Tabelle mit den Adressen ihrer virtuellen Funktionen (vtable) angelegt. Außerdem bekommt jede Instanz einer solchen Klasse einen zusätzlichen Pointer, der auf diese vtable zeigt (**Bild 3**).

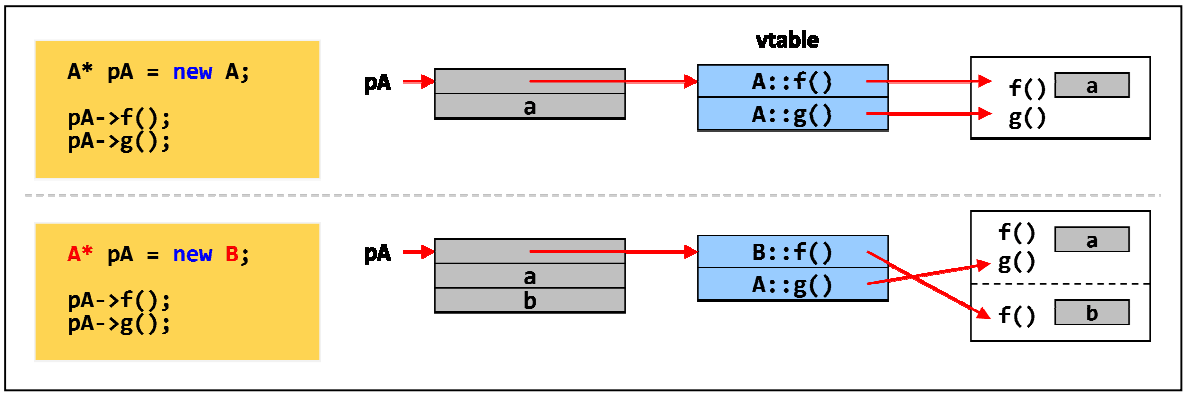


Bild 3: Implementierung von virtuellen Funktionen

Dieser Aufwand ist nötig, um Funktionen eines abgeleiteten Objektes aufzurufen können, von dem man nur einen Interface- oder Basisklassen-Pointer besitzt. **Bild 4** (oberer Teil) zeigt ein Beispiel, bei dem der virtuelle Aufruf doppelt so viel Speicher und fast drei Mal so viel Laufzeit benötigt, wie ein nicht-virtueller Aufruf. Hier ist zu

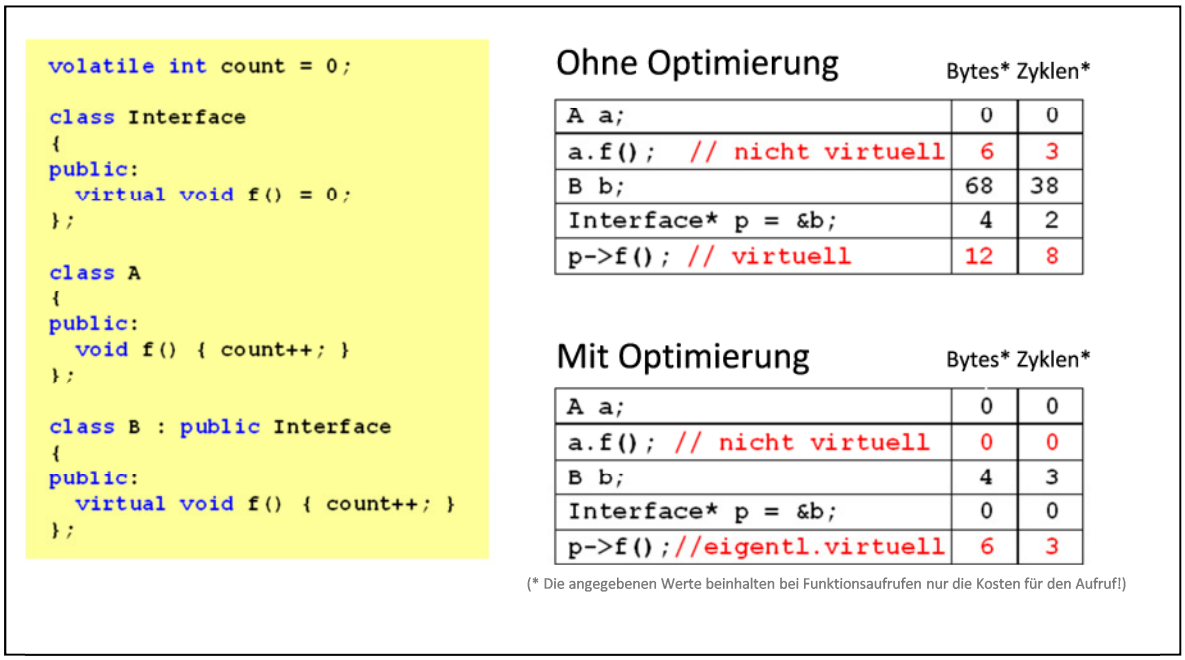
beachten, dass es sich dabei nicht um ein ineffizientes Verhalten von C++, sondern um die Implementierung eines grundlegenden objektorientierten Konzeptes handelt, das in C auf ähnliche Art nachgebildet werden müsste!

Design Patterns in Embedded Systemen?

Ob man sich den Einsatz von Patterns leisten kann, hängt in erster Linie von den Eigenschaften des Systems ab. Generell kann man aber sagen, dass der Speicher auch in Embedded Systemen im Laufe der Jahre stetig zunimmt, so dass immer mehr Speicher auch für solche Zwecke zur Verfügung steht. Analysiert man laufzeitkritische Embedded Systeme, dann wird man feststellen, dass der wirklich zeitkritische Codeanteil in der Regel eher klein (häufig in der Größenordnung von ca. 5 %) ist. D.h., dass ein geringer Mehraufwand durch die Verwendung von Patterns bei dem weitaus größten Teil selbst solcher Anwendungen unkritisch ist.

Man sollte die Verwendung eines Patterns sehr genau prüfen, wenn das Pattern
- Teil zeitkritischen Codes ist
- Teil einer Schleife ist, die sehr häufig ausgeführt wird
- den überwiegenden Teil einer Gesamtoperation ausmacht.

Darüber hinaus kann auch die Wahl eines gut optimierenden Compilers dazu beitragen, den Zusatzaufwand von Patterns zu reduzieren. **Bild 4** (unterer Teil) zeigt ein Beispiel eines optimierten Aufrufs über eine Schnittstelle, der nicht mehr kostet, als ein nichtoptimierter normaler Aufruf, weil in diesem Fall die Funktion direkt aufgerufen werden kann.



```
volatile int count = 0;

class Interface
{
public:
  virtual void f() = 0;
};

class A
{
public:
  void f() { count++; }
};

class B : public Interface
{
public:
  virtual void f() { count++; }
};
```

Ohne Optimierung

	Bytes*	Zyklen*
A a;	0	0
a.f(); // nicht virtuell	6	3
B b;	68	38
Interface* p = &b;	4	2
p->f(); // virtuell	12	8

Mit Optimierung

	Bytes*	Zyklen*
A a;	0	0
a.f(); // nicht virtuell	0	0
B b;	4	3
Interface* p = &b;	0	0
p->f();//eigentl.virtuell	6	3

(* Die angegebenen Werte beinhalten bei Funktionsaufrufen nur die Kosten für den Aufruf!)

Bild 4: Beispiel: Kosten für einen virtuellen und nicht-virtuellen Funktionsaufruf

Manchmal können virtuelle Funktionen auch mithilfe von Templates vermieden werden. Allerdings muss man sich dann zum Übersetzungszeitpunkt festlegen, mit welchen Klassen gearbeitet werden soll, wodurch ein großer Teil der Flexibilität der Patterns verloren geht.

Schließlich sollte man auch erwähnen, dass die geschickte Verwendung von Patterns in manchen Fällen sogar zur Einsparung von Ressourcen führen kann, so dass der Pattern-Mehraufwand mehr als nur kompensiert wird (**Bild 5**).

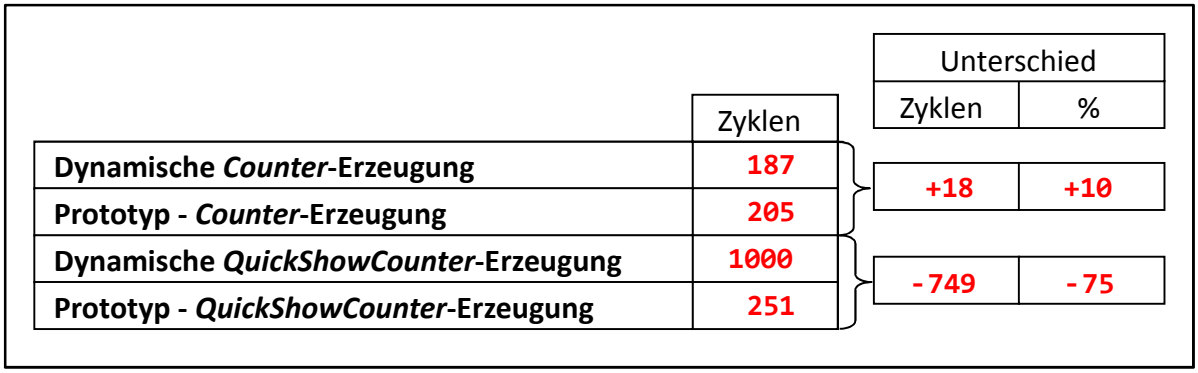


Bild 5: Beispiel: Einsparpotenzial bei Anwendung des Prototyp-Musters

Zusammenfassung

- Design Patterns bieten viele Vorteile in Bezug auf Flexibilität, Wiederverwendbarkeit, Wartbarkeit und Zukunftssicherheit

- Im Allgemeinen führt ihr Einsatz zu etwas mehr Speicherplatz- und/oder Laufzeitbedarf

- In der Regel ist aber selbst der Einsatz in ressourcelimitierten Embedded Systemen möglich, denn
  - Speicherplatz ist nicht mehr so knapp wie früher
  - normalerweise ist nur sehr wenig Code wirklich zeitkritisch
  - hochoptimierende Compiler und Templates können den Zusatzaufwand geringer halten
  - Nachteile können mitunter kompensiert oder sogar überkompensiert werden, so dass es letztendlich sogar zu einem Ressourcengewinn kommen kann

- **Eine unverzichtbare Voraussetzung dafür ist allerdings eine sehr genaue Kenntnis der Eigenschaften und Kosten der Patterns!**

Literatur:

Design Patterns: Elements of Reusable Object-Oriented Software,
E. Gamma, R. Helm, R.Johnson und J. Vlissides

Autor

Dipl. Inf. Karl Nieratschker verfügt über eine langjährige Erfahrung in den Bereichen Softwareentwicklung, Support, Beratung und Training, überwiegend für Embedded-/Realtime- und Windows-Systeme. Seit 1999 ist er als freiberuflicher Trainer, Softwareberater und Coach tätig. Einer seiner Schwerpunkte ist die objektorientierte Programmierung in ressourcelimitierten Systemen.

Internet: www.skt-nieratschker.de
Email: office@skt-nieratschker.de