Schnittstellen in Embedded Systemen

Karl Nieratschker, SKT Nieratschker

Durch den stetig wachsenden Applikationsumfang und die zunehmende Produktvielfalt werden Schnittstellen (Interfaces) immer wichtiger, denn sie sorgen für flexible und leicht anpassbare Softwarestrukturen. Allerdings ist ihr Einsatz in der Regel mit Speicherplatz- und/oder Laufzeitkosten verbunden, so dass bei der Anwendung in Embedded Systemen genau überlegt werden muss, wie oft und in welcher Form Schnittstellen zum Einsatz kommen sollen.

Was ist eine Schnittstelle und wo liegt ihr Vorteil?

Angenommen ein Embedded System besitzt einen Motor Namens SKT4711, der sich ein- und ausschalten lässt und dessen Drehrichtung eingestellt werden kann. Diese Funktionalität wurde in einer Klasse `MotorSKT4711` mit den Methoden `Start()`, `Stop()`, `Clockwise()` und `Anticlockwise()` gekapselt. Gesteuert wird der Motor durch eine Instanz der Klasse `MotorController`, die mithilfe eines `MotorSKT4711`-Pointers die Methoden des Motors aufruft und so den Motor steuert.

```
void Sleep(int milliseconds);

class MotorController
{
   enum { SLEEP_TIME = 3000 };

   MotorSKT4711* pMotor;

public:

   MotorController(MotorSKT4711* pMotor) :
       pMotor(pMotor) {}

   void Run()
   {
      do
      {
          pMotor->Stop();
          pMotor->Clockwise();
          pMotor->Start();

          Sleep(SLEEP_TIME);

          pMotor->Stop();
          pMotor->Anticlockwise();
          pMotor->Start();

          Sleep(SLEEP_TIME);

      } while(true);
   }
};
```

```
class MotorSKT4711
{
private:
   ...    // Datenelemente

public:

   MotorSKT4711(...);

   void Start();
   void Stop();
   void Clockwise();
   void Anticlockwise();
};
```

```
void main()
{
  MotorSKT4711 motor(...);
  MotorController mc(&motor);

  mc.Run();
}
```

Abb. 1: MotorController mit MotorSKT4711

Muss nun irgendwann der Motor ersetzt werden, weil dieses Modell z.B. nicht mehr gebaut wird, oder weil ein anderes Modell preisgünstiger ist, dann muss die Applikation aufgrund der engen Kopplung an allen Stellen, wo der Motor verwendet wurde, an den neuen Motor angepasst werden. Da das bisher existierende System in der Regel noch weiter gewartet werden muss, führt dies dazu, dass von nun an zwei Systeme gepflegt werden müssen.

Ein besserer Ansatz besteht darin, den konkreten Motor vom anwendenden Code zu entkoppeln, in dem man die vom Anwender benötigten Funktionen in einer Klasse `Motor` als abstrakte Methoden definiert, die dann von der konkreten Motorklasse implementiert werden.

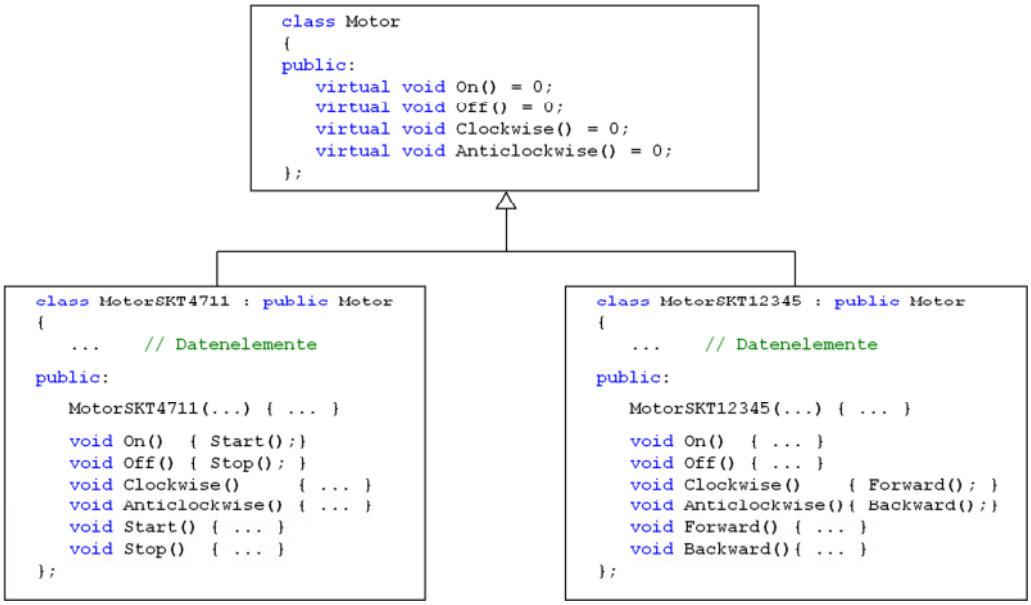

Abb. 2: Abstrakte Motorklasse

Statt einem `MotorSKT4711`-Pointer kann der anwendende Code nun mit einem `Motor`-Pointer arbeiten, der auf die Instanz des konkreten Motors zeigt. Durch diese Implementierung spielt es keine Rolle mehr, was der genaue Typ des Motors ist, so dass der Motor beliebig ausgetauscht werden kann. Die abstrakte Klasse `Motor` stellt die Schnittstelle (Interface) zwischen dem Anwendercode und der konkreten Motor-Instanz her.

```
void Sleep(int milliseconds);

class MotorController
{
    enum { SLEEP_TIME = 3000 };

    Motor* pMotor;

public:

    MotorController(Motor* pMotor) :
        pMotor(pMotor) {}

    void Run()
    {
       do
       {
           pMotor->Off();
           pMotor->Clockwise();
           pMotor->On();

           Sleep(SLEEP_TIME);

           pMotor->Off();
           pMotor->Anticlockwise();
           pMotor->On();

           Sleep(SLEEP_TIME);

       } while(true);
    }
};
```

```
void main()
{
  MotorSKT4711 motor(...);
  MotorController mc(&motor);

  mc.Run();
}
```

oder

```
void main()
{
  MotorSKT12345 motor(...);
  MotorController mc(&motor);

  mc.Run();
}
```

Abb. 3: MotorController mit Motor-Schnittstelle

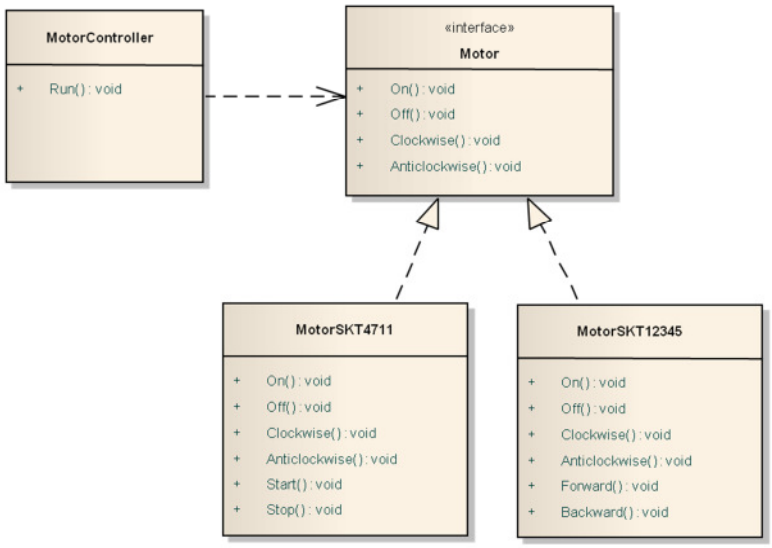

Abb. 4: UML-Darstellung der Motor-Schnittstelle

Was kostet mich der Einsatz einer Schnittstelle?

Da eine abstrakte Methode keinen ausführbaren Code besitzt, muss der Compiler beim Aufruf durch den Anwendercode dafür sorgen, dass stattdessen die Methode der konkreten Klasse aufgerufen wird, die die abstrakte Methode implementiert. Dies wird dadurch erreicht, dass die abstrakte Methode mit dem Schlüsselwort `virtual` deklariert wird. Allerdings hat dies zur Folge, dass der Compiler für jede Klasse, die virtuelle Methoden implementiert, eine Tabelle (vtable) anlegt, die die Adressen aller virtuellen Methoden der Klasse enthält. Außerdem erhält jede Instanz einer solchen Klasse (z.B. `MotorSKT4711`) einen zusätzlichen Pointer, der auf die vtable der Klasse zeigt. Beim Aufruf einer Schnittstellenmethode erzeugt der Compiler dann Code, der mithilfe des vtable-Pointers der (`MotorSKT4711`-) Instanz und einem entsprechenden Offset die Adresse der aufzurufenden Methode ermittelt. Das Beispiel in Abb. 5 zeigt, dass für die Anweisung `pMotor->Off()` fünf Assemblerbefehle bzw. 12 Bytes benötigt werden. Für den direkten Aufruf `motor.Stop()` wird zwar nur ein Befehl erzeugt, aber der Vergleichbarkeit halber muss der weiter oben stehende Befehl, der den this-Pointer in das Register R0 lädt, auch berücksichtigt werden, so dass insgesamt 2 Befehle bzw. 6 Bytes benötigt werden. D.h., in diesem Beispiel ist der Aufruf über die Schnittstelle um den Faktor 2 bis 2,5 teurer.

```
void main()
{
  MotorSKT4711 motor;
  Motor* pMotor = &motor;
  motor.Stop();
  pMotor->Off();
}
```

```
          void main()
          {
                  main:
00000000   00B5              PUSH     {LR}
00000002   83B0              SUB      SP,SP,#+12

           MotorSKT4711 motor;
00000004   6846              MOV      R0,SP    ; this-Pointer nach R0
00000006   ........          BL       _ZN12MotorSKT4711C1Ev
           Motor* pMotor = &motor;
           motor.Stop();
0000000A   ........          BL       _ZN12MotorSKT47114StopEv
           pMotor->Off();
0000000E   6846              MOV      R0,SP          ; this-Pointer nach R0
00000010   0099              LDR      R1,[SP, #+0]   ; vtble-Adresse nach R1
00000012   4968              LDR      R1,[R1, #+4]   ; Adr. d. Methode Off nach R1
00000014   ........          BL       __iar_via_R1   ; Sprung über R1 nach Off
          }
00000018   03B0              ADD      SP,SP,#+12
0000001A   08BC              POP      {R3}
0000001C   1847              BX       R3               ;; return
```

```
__iar_via_R1:     4708        BX       R1
```

Abb. 5: Assembler Code für die Methode Off

Kann ich mir den Einsatz einer Schnittstelle im Embedded System leisten?

Ob der Zusatzaufwand durch die Verwendung eines Interfaces für die Applikation kritisch ist, hängt stark vom jeweiligen Anwendungsfall ab. Im Verhältnis zu der Zeit, die ein Motor braucht, um anzuhalten, könnte es z.B. vernachlässigbar sein, ob man dafür 2 oder 5 Assemblerbefehle benötigt, insbesondere dann, wenn der Motor nur im Abstand von

einigen Sekunden geschaltet wird. Wird eine virtuelle Methode dagegen in einer häufig ausgeführten Schleife oder in einer Interruptserviceroutine aufgerufen, dann kann der Unterschied durchaus relevant sein. Andererseits muss man aber auch bedenken, dass der Umfang des zur Verfügung stehenden Speichers und die Leistungsfähigkeit moderner Mikrocontroller ständig zunimmt, während der schnittstellenbedingte Mehraufwand an Speicherplatz und auszuführender Befehle gleich bleibt und somit immer weniger eine Rolle spielt.

Was kann ich tun, um den Schnittstellenaufwand zu reduzieren?

Der durch die virtuellen Methoden entstehende Mehraufwand lässt sich vermeiden, wenn man den Typ des Motors im anwendenden Code als Template-Typ deklariert (Abb. 6). Dies setzt allerdings voraus, dass der Compiler Templates unterstützt, was nicht selbstverständlich ist, da dies nicht Bestandteil des Embedded-C++-Standards ist. Außerdem kann eine `MotorController`-Instanz dann nur mit einer Motorart arbeiten, die zum Übersetzungszeit bekannt sein muss. Es ist zwar möglich, dass verschiedene `MotorController`-Instanzen Motoren unterschiedlichen Typs steuern, aber dies führt zu mehr Speicheraufwand, weil der Code der `MotorController`-Klasse einmal pro verwendetem Motortyp angelegt wird. Verwendet die Applikation nur einen einzigen Motortyp, der zum Übersetzungszeitpunkt festgelegt wird, dann kann die Template-Definition auch durch eine `typedef`-Deklaration ersetzt werden. Diese Lösung hat den Vorteil, dass sie auch mit Compilern funktioniert, die nur den minimalen Embedded-C++-Standard unterstützen.

Abb. 6: Template-Lösung

Was gehört zu einem guten Schnittstellendesign?

Um zu verdeutlichen, dass es sich um eine Schnittstelle (und nicht um eine normale Klasse) handelt, wird Interfaces in der Regel ein Name gegeben, der mit einem großen „I“ beginnt, also `IMotor` statt nur `Motor`. Allerdings gibt es keinen Zwang, diese Konvention einzuhalten.
Muss eine Applikation mit verschiedenen Arten von Motorfunktionalitäten umgehen können, wie z.B. Motoren, die man nur ein- und ausschalten kann und solche, bei denen zusätzlich auch die Drehrichtung geändert werden kann, dann hat das bisher gezeigte `IMotor`-Interface den Nachteil, dass sich die Methoden `Clockwise()` und `Anticlockwise()` bei den einfachen Motorarten nicht sinnvoll implementieren lassen. Da der Compiler die Implementierung aller Elemente einer Schnittstelle fordert, werden diese Methoden in diesem Fall oft leer oder so implementiert, dass sie Fehler liefern. Diesen Lösungsansatz sollte man unbedingt vermeiden, denn der Code der unsinnigen Methoden belegt wertvollen Speicherplatz und ermöglicht dem Anwender, den Motor falsch zu verwenden. Stattdessen sollte man besser mit zwei Interfaces arbeiten, die hierarchisch aufeinander aufbauen, wie dies in Abb. 7 gezeigt wird. Auf diese Weise kann dafür gesorgt werden, dass jeder Anwender nur die Funktionalität angeboten bekommt, die er benötigt. Generell sollte eine Schnittstelle so klein wie möglich sein. Häufig besitzt eine Schnittstelle nur eine einzige Methode!

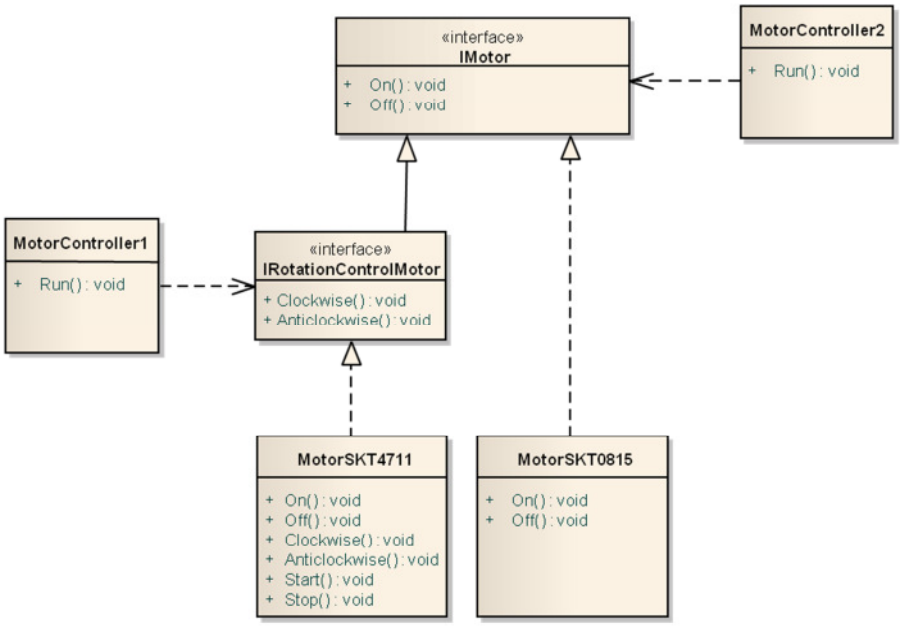

Abb. 7: Schnittstellenerweiterung

Grundsätzlich kann man zwei Arten von Schnittstellen unterscheiden. Das `IMotor`- bzw. `IRotationControlMotor`-Interface ist ein Beispiel für eine „vertikale“ Schnittstelle, die die grundlegende Funktionalität einer Hierarchie von Motorklassen beschreibt. Eine „horizontale“ Schnittstelle beschreibt eine Funktionalität, die von völlig unabhängigen Klassen, die keine gemeinsame Vererbungshierarchie besitzen, implementiert werden kann. Ein Interface `ITimerEventUser` mit einer Methode `HandleTimerEvent()` z.B.,

könnte von allen Klassen implementiert werden, die auf einen Timer-Interrupt reagieren müssen. Bei der Implementierung einer derartigen Schnittstelle kann es allerdings passieren, dass die Allgemeingültigkeit der ursprünglichen Klasse verletzt wird, was ihre Wiederverwendbarkeit einschränkt. In diesem Fall sollte eine von der ursprünglichen Klasse abgeleitete Klasse das Interface implementieren, was allerdings zu einer Mehrfachvererbung führt. Bei der Wahl des Compilers sollte deshalb darauf geachtet werden, dass Mehrfachvererbung unterstützt wird, da dieses Problem sonst mit viel Aufwand auf Anwenderebene gelöst werden muss.
Prinzipiell stellt sich beim Schnittstellendesign immer die Frage, ob man eher wenige, möglichst allgemeingültige, oder eher viele, möglichst spezialisierte Schnittstellen verwenden sollte. Leider gibt es hierfür kein Patentrezept, denn jede Vorgehensweise hat ihre Vor- und Nachteile. Aus diesem Grund kann eine Entscheidung in der Praxis immer nur anhand der gegebenen Situation getroffen werden.

Zusammenfassung

Schnittstellen bilden eine fundamentale Voraussetzung für die Entwicklung moderner und flexibler Softwaresysteme. Sie ermöglichen die Realisierung von Abstraktionsebenen und tragen zur Vereinfachung der Softwareentwicklung bei. Hinter Schnittstellen verborgene Systemkomponenten können beliebig ausgetauscht und so das Systemverhalten auf einfache Weise an neue Anforderungen angepasst werden. Der Einsatz von Schnittstellen kostet im Allgemeinen zwar Systemressourcen und muss deshalb gerade in Embedded Systemen genau geprüft werden. Doch aufgrund der rapiden Leistungszunahme moderner Embedded Systeme ist diese Einschränkung immer weniger Ausschlag gebend. Bereits heute gibt es viele Möglichkeiten, auch im Embedded Bereich von den Vorteilen des Schnittstellendesigns profitieren zu können.

Autor

Dipl. Inf. Karl Nieratschker verfügt über eine mehr als 25-jährige Erfahrung in den Bereichen Softwareentwicklung, Support, Beratung und Training für Embedded-/Realtime- und Windows-Programmierung. Seit 1999 ist er als freiberuflicher Trainer, Softwareberater und Coach tätig. Seine besonderen Schwerpunkte sind „Objektorientierte Programmierung in ressource-limitierten Systemen“ und „Multithreading/Multicore-Programmierung“.

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