

Programmieren in C++ (4): Abstrakte Klassen und Schnittstellen

Silke Seehusen, Fachhochschule Lübeck

Es wird eine Einführung ins Programmieren in C++ gegeben, die an Programmierkonzepten wie Datenabstraktion, objektorientierte Programmierung und Polymorphismus ausgerichtet ist. In diesem vierten Teil wird gezeigt, wie abstrakte Klassen und allgemeine Schnittstellen realisiert werden können.

Figuren

In den vorhergehenden Artikeln wurde eine Klasse als ein Typ für eine Menge von Objekten angesehen. Ein Objekt ist dabei eine Instanz einer Klasse, und von den bisher eingeführten Klassen konnten auch immer Objekte instanziiert werden.

Im folgenden werden abstrakte Klassen eingeführt, die die Schnittstelle einer Menge von Klassen festlegen. Von einer abstrakten Klasse selbst kann kein Objekt instanziiert werden.

Als ein wohl berühmtes Beispiel dienen grafische Figuren. Die Figuren sind zweidimensional, haben unterschiedliche Formen und können auf einer Leinwand gezeichnet werden. Konkrete Figuren sind unter anderem Kreise, Rechtecke, Linien, Dreiecke, Ellipsen und Polygone. Diese Figuren haben einiges gemeinsam: Sie können alle gezeichnet, verschoben und gedreht werden. Sie haben also "gleiche" Methoden, die sich sicherlich in der Implementierung unterscheiden werden.

Gemeinsame Daten haben die Figuren in der Regel nicht.

Von der objektorientierten Modellierung her beschreibt Figur das Gemeinsame, und die speziellen Figuren wie Kreis und Rechteck werden von einer Figur abgeleitet, siehe Abbildung 1.

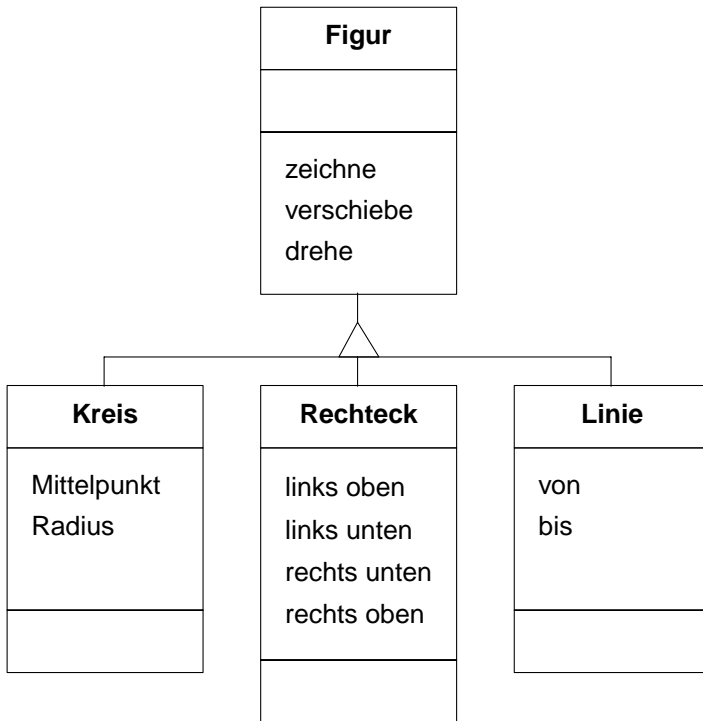


Abbildung 1: Figuren

Ein Kreis wird durch seinen Mittelpunkt und seinen Radius repräsentiert. Ein Rechteck wird durch seine vier Ecken festgelegt.

Bei der Implementierung der Klasse *Figur* merken wir, daß eine Figur zwar die Methode *zeichne* bereitstellen muß, daß wir aber allgemein keine Implementierung dafür angeben können. Dies wird in C++ dadurch notiert, daß die Methode mit 0 initialisiert wird:

```
virtual void zeichne() = 0;
```

Eine solche Funktion wird in C++ auch **rein virtuelle Funktion** genannt. In [1] wird eine solche Funktion als **abstrakte Operation** bezeichnet.

Ein Auszug aus der Definition der Klasse *Figur* ist in Abbildung 2 wiedergegeben.

```
class Figur
{
public:
    virtual ~Figur() {};
    virtual void zeichne ( Leinwand * ) = 0;
    virtual void verschiebe ( const Punkt & ) = 0;
    virtual void drehe ( Winkel ) = 0;

    virtual Punkt mitte ( ) const = 0;
    virtual Punkt nord ( ) const = 0;
    virtual Punkt sued ( ) const = 0;
    // ...
    virtual Punkt suedost ( ) const = 0;
};
```

Abbildung 2: Die abstrakte Klasse *Figur*

Neben Zeichnen, Verschieben und Drehen können von einer Figur auch Punkte wie der nördlichste und südlichste Punkt bestimmt werden. Ein Punkt wird durch seine x- und y-Koordinate repräsentiert.

Eine Klasse, die mindestens eine rein virtuelle Funktion besitzt, ist eine **abstrakte Klasse**. Von einer abstrakten Klasse kann kein Objekt gebildet werden. Die Klasse *Figur* hat deshalb auch keinen Konstruktor.

Ein Kreis hat nun die Eigenschaften einer Figur und eventuell weitere Eigenschaften. Eine Klassendefinition von *Kreis* ist in Abbildung 3 angegeben.

```
class Kreis : public Figur
{
private:
    Punkt      _mittelpunkt;
    Koordinate _radius;
public:
    Kreis ( const Punkt & mittelpunkt,
           Koordinate radius = 1 );
    Kreis ( Koordinate radius = 1 );
    virtual void zeichne ( Leinwand * );
    virtual void verschiebe( const Punkt& );
    virtual void drehe    ( Winkel )  {};

    virtual Punkt mitte    () const;
    virtual Punkt nord     () const;
    virtual Punkt sued     () const;
    // ...
    virtual Punkt suedost () const;

    virtual Koordinate radius() const;
};
```

Abbildung 3: Kreis als spezielle Figur

Da der *Kreis* keine rein virtuelle Funktion mehr besitzt, kann ein Objekt von der Klasse gebildet werden, z.B. durch

```
Punkt mittelpunkt(80, 60);
Koordinate radius = 40;
Kreis * kreis = new
    Kreis(mittelpunkt, radius);
```

Ein Kreis muß alle Funktionen implementieren, die in der Klasse *Figur* rein virtuell sind. Er kann zusätzliche Funktionen anbieten. In Abbildung 3 wird die Zugriffsfunktion *radius()* neben den vorgeschriebenen Funktionen bereitgestellt.

Die Funktion *zeichne()* ist so definiert, daß sie eine Figur verändern kann. Dies läßt z.B. eine Veränderung des Zustands einer Figur, hier eines Kreises, zu, wenn die Figur gezeichnet wird. Denkbar ist ein Zwischenspeicher für eine gezeichnete Figur.

Alle Funktionen eines Kreises, bis auf den Konstruktor, sind in Abbildung 3 virtuell deklariert. Diese Art ist nur deshalb gewählt worden, um weitere Spezialisierungen eines Kreises wie eine normale Figur behandeln zu können.

Analog zum Kreis können jetzt die Klassen *Rechteck*, *Linie* und die Figur *Text* definiert werden. Eine Text-Figur enthält eine Zeichenkette und eine Position.

Bild

Der wirkliche Nutzen der abstrakten Klasse *Figur* kommt dann zum Tragen, wenn verschiedenartige Figuren in einem Bild auf einer Leinwand dargestellt werden sollen. Ein Bild beherberge eine Menge solcher Figuren. Die Klasse *Bild* wird in Abbildung 4 definiert.

```
class Bild
{
private:
    typedef less<Figur*>      lessFigur;
    typedef set<Figur*,lessFigur> setFigur;
    setFigur  _figuren; // Figuren auf dem Bild
    Leinwand * _leinwand; // Leinwand zur Darstellung
public:
    Bild( const String & id = ".c" );
```

```

Bild( Leinwand *, const String & id = ".c" );
virtual ~Bild();
virtual void zeichne(); // stellt Bild dar
virtual void neueFigur ( Figur * );
virtual void loeschFigur ( Figur * );
};

```

Abbildung 4: Ein Bild

Die Menge der Figuren wird durch das von der C++-Standard-Bibliothek [2] bereitgestellte Container-Templete *set<class Key, class Compare>* implementiert. Das Templete funktioniert so ähnlich wie das Templete *Map* aus dem vorhergehenden Artikel dieser Reihe [3], außer daß *set* keine Abbildung sondern nur Elemente abspeichert. Das Einfügen einer Figur ist damit ein Einzeiler:

```

void Bild::
neueFigur( Figur * f )
{
    _figuren.insert( f );
}

```

Die Funktion *zeichne()* der Klasse *Bild* wird unabhängig von den konkreten Figuren des Bilds implementiert:

```

void Bild::zeichne()
{
    _leinwand->clear();
    for( setFigur::iterator
        i = _figuren.begin();
        i != _figuren.end();
        i++ )
    {
        (*i)->zeichne(_leinwand);
    }
    _leinwand->zeige();
}

```

Dabei wird ausgenutzt, daß jede Figur die Schnittstelle, die durch die Klasse *Figur* festgelegt wurde, einhalten muß. Zwar existiert die rein virtuelle Funktion *Figur::zeichne()* nicht, aber zur Laufzeit ist bekannt, welche konkrete Funktion, z.B. *Kreis::zeichne()*, ausgeführt werden soll.

Werden Linien ins Bild eingetragen, werden auch Linien gezeichnet:

```

Bild bild;
Punkt ecke[9]; // Hausecken
// ... werden initialisiert
// und ergeben 8 Linien:
for ( int i=0; i < 8; i++ )
{
    bild.neueFigur( new Linie
                    (ecke[i], ecke[i+1]));
    bild.zeichne();
    sleep(1); // zum Zuschauen
}

```

Das obige Beispiel erzeugt das Haus, das in Abbildung 5 dargestellt ist. Die Initialisierung der Ecken ist ein bekanntes Kinderspiel.

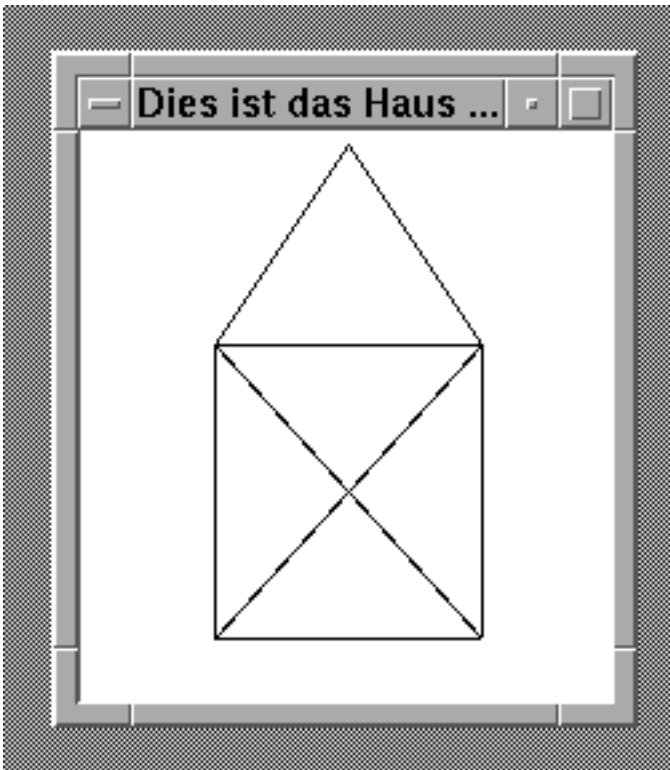


Abbildung 5: Dies ist das Haus von Ni-ko-laus

Die Zeichenfunktionen der Figuren sind für die Beispiele dieses Artikels auf Tcl/Tk-Funktionen [4] abgebildet. Die Beispielprogramme erzeugen als Ausgabe Kommandos für die Window-Shell *wish*.

Auch unterschiedliche Figuren in einem Bild sind kein Problem. Mit dem Kreis *kreis* von oben erzeugt

```
Rechteck * recht = new
    Rechteck(kreis->mitte(),
             2*kreis->mitte());

bild.neueFigur( kreis );
bild.neueFigur( recht );
bild.neueFigur( new Quadrat
    (kreis->mitte() -
     Punkt(radius,radius),
     2*radius) );
bild.neueFigur( new Linie
    (kreis->west(),
     recht->suedwest()) );
bild.zeichne();
```

die Abbildung 6.

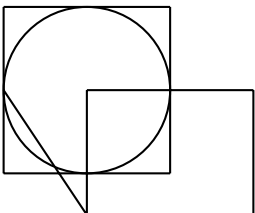


Abbildung 6: Figuren auf Leinwand

Werden bei der Konstruktion eines Rechtecks wie oben nur zwei Punkte angegeben, so werden die Kanten parallel zu den Achsen ausgerichtet und die beiden Punkte als zwei diagonal gegenüberliegende Ecken interpretiert.

Die Multiplikation eines Punkts mit einem Skalar, wie z.B. in $2 * \text{kreis} \rightarrow \text{mitte}()$, wird als Multiplikation jeder Koordinate des Punktes mit dem Skalar definiert.

Quadratur?

Soll das System aus Abbildung 1 um eine weitere Figurenart, z.B. ein Quadrat, das oben schon benutzt wird, erweitert werden, muß nur die Klasse *Quadrat* geschrieben werden, die von der abstrakten Klasse *Figur* erben und die Schnittstelle einer Figur bereitstellen muß.

Dann kann das Quadrat schon wie z.B. ein Rechteck benutzt werden, d.h. es kann in ein Anwendungsprogramm integriert werden.

Die Klassen *Bild* und *Figur* bleiben bei der ganzen Erweiterung unberührt, es sei denn, die Abstraktion der Figur war nicht hinreichend. Aber das hieße, daß nicht eine weitere Figurenart ergänzt werden soll, sondern daß plötzlich ganz andere Arten von Objekten dargestellt werden sollen. Eine ganz andere Art sind z.B. Bewegtbilder.

So einfach wie um ein Quadrat kann die Figurenvielfalt auch um Ellipse, Bogen, Polygon und Spline bereichert werden. Dabei müssen die neuen Figuren nicht direkt von der Klasse *Figur* erben, sondern können von abgeleiteten Klassen erben. Das Quadrat kann z.B. als spezielles Rechteck definiert werden.

Es sollte immer das Prinzip gelten, daß eine Erweiterung, die am zugrundeliegenden Modell eines Systems nichts ändert, einen Aufwand proportional zur Größe der Erweiterung verursacht. Desweiteren sollten nur die Klassen geändert werden müssen, die das Neue benutzen.

Im obigen Beispiel muß die Klasse *Bild* nicht verändert werden, weil ein *Bild* sich auf die durch die Klasse *Figur* festgelegte Schnittstelle bezieht. Diese Schnittstelle kann auch als eine Art Protokoll bezeichnet werden, wie mit Objekten, die Figuren sind, umgegangen werden kann.

Der Übersetzer überprüft, ob ein *Bild* als Benutzer diese Schnittstelle einhält, und er überprüft, ob ein Objekt, das das *Bild* verwenden will, von einem Typ ist, das diese Schnittstelle zur Verfügung stellt.

Der Übersetzer kann natürlich durch explizite Typkonvertierungen in seiner Arbeit behindert werden, aber das gilt bei Typkonvertierungen fast immer.

Eine andere Leinwand

Die Grafiken in Abbildung 5 und 6 sind auf einer Leinwand von Tcl/Tk erstellt worden. Da es sehr viele verschiedene Ausgabemedien und auch sehr viele verschiedene Bibliotheken zur grafischen Darstellung gibt, sollte ein *Bild* und auch eine *Figur* relativ unabhängig davon realisiert werden.

Deshalb bietet es sich an, auch eine Leinwand als abstrakte Klasse zu definieren. Eine Leinwand unter Tcl/Tk ist dann eine spezielle Leinwand. Ebenso gibt es eine (schlechte) Leinwand mit reiner Zeichenausgabe. Eine Definition der Klasse *Leinwand* ist in Abbildung 7 wiedergegeben.

```
class Leinwand
{
private:
    String    _id;
protected:
    Leinwand( const String & id = ".c" ) : _id( id ) {}
public:
    virtual ~Leinwand()    {}
    virtual void zeige () = 0;
    virtual void clear () = 0;
```

```

virtual FID erzeugeLinie
    ( const Punkt& von, const Punkt& bis )    =0;
virtual FID erzeugeKreis
    (const Punkt& mitte, const Koordinate& r)=0;
virtual FID erzeugeText
    ( const Punkt& mitte, const String& )    =0;
// ...
};

```

Abbildung 7: Leinwand

Zwei abgeleitete, konkrete Klassen sind *LeinwandTk* für die Ausgabe unter Tcl/Tk und *LeinwandTty* für die reine Zeichenausgabe.

Die Klasse *Leinwand* ist eine abstrakte Klasse, obwohl sie einen Konstruktor und einen Destruktor besitzt. Da sie eine abstrakte Klasse ist, kann der Konstruktor jedoch nur durch die Konstruktoren der abgeleiteten Klassen benutzt werden. Er ist deshalb als *protected* spezifiziert, wodurch explizit angegeben wird, daß er nur für abgeleitete Klassen sichtbar ist.

Der Rückgabewert vom Typ *FID* von Funktionen wie *erzeugeLinie()* soll eine Identifikation des jeweilig gezeichneten Objekts sein.

Um nun ein Bild mit Zeichenausgabe darzustellen, muß dem Konstruktor eines Bilds eine Leinwand vom Typ *LeinwandTty* mitgegeben werden. Das ist die einzige Änderung. Ein Rechteck sieht jetzt mit:

```

LeinwandTty * tty = new
    LeinwandTty();
Bild bild(tty);

Rechteck * recht = new
    Rechteck(Punkt( 25, 40),
    Punkt(125,115));
bild.neueFigur( recht );
bild.zeichne();

```

etwas anders aus als unter Tcl/Tk:

```

*****
*           *
*           *
*           *
*           *
*           *
*           *
*****

```

Und nach einer Drehung um $\pi/4$

```

recht->drehe(pi/4);

```

sieht es mit einem zugegebenermaßen schlechten Linienalgorithmus nicht eben besser aus:

```

    ***
  *** ***
 ***   ***
***   ***
**    ***
***   ***
**   ***
***   ***
***   ***
***   ***
***   ***
***   ***
***   ***
***   ***
***   ***

```

Das Rechteck ist erkennbar und die Darstellung dem Ausgabemedium angemessen.

Die Erweiterung um einen neuen Leinwandtyp ist genauso einfach wie die Erweiterung um eine neue Figurenart. Damit das so einfach ist, müssen die konkreten Figuren wie *Kreis* oder *Rechteck* das Protokoll benutzen, das durch die abstrakte Klasse *Leinwand* festgelegt wird. Den Rest sollte der Übersetzer besorgen. Ein Rechteck wird z.B. wie folgt gezeichnet:

```
void Rechteck::zeichne
    ( Leinwand * v )
{ v->erzeugeLinie
  (_nordwest, _nordost );
  v->erzeugeLinie
  (_nordost, _suedost );
  v->erzeugeLinie
  (_suedost, _suedwest);
  v->erzeugeLinie
  (_suedwest, _nordwest);
}
```

Die Klasse *Bild* ist sogar so konstruiert, daß jederzeit die Leinwand ausgetauscht werden kann.

Verwendung von abstrakten Klassen

Abstrakte Klassen können aus unterschiedlichen Gründen beim Entwurf und der Implementierung eines Systems entstehen.

Wird bei der Modellierung der Anwendung eine Klasse eingeführt, von der selbst kein Objekt instanziiert werden kann, die jedoch die gemeinsamen Eigenschaften von einigen abgeleiteten Klassen beschreibt, so ist die Klasse schon von der Modellbildung her eine abstrakte Klasse. Ein typisches Beispiel ist die Klasse *Figur*.

Soll eine Funktionalität, wie z.B. eine grafische Darstellung, leicht austauschbar sein, eventuell auch zur Laufzeit, so bietet es sich an, für die Funktionalität eine abstrakte Klasse zu definieren. Alle Objekte, die die Funktionalität brauchen, benutzen dann die durch die abstrakte Klasse festgelegte Schnittstelle. Eine solche Klasse nennen wir auch **Schnittstellenklasse**. Sie kann für einige Funktionen schon Implementierungen vorsehen, die jedoch bei Bedarf von den abgeleiteten Klassen überschrieben werden können. Eine Schnittstellenklasse muß sogar nicht unbedingt abstrakt sein. Sie ist es vom Prinzip her.

Insbesondere lassen sich durch abstrakte Klasse größere Subsysteme besser separieren. In einer oder vielleicht auch ein paar wenigen Klassen wird die Schnittstelle zu einem Subsystem festgelegt. Dann kann die weitere Entwicklung der Subsysteme besser parallel erfolgen, und das Subsystem kann leicht ausgetauscht werden.

Das Subsystem kann z.B. auch ein Gerät oder einen anderen Prozessor inkapseln. Dies ist auch in der Testphase sehr hilfreich. Heute gibt es für viele unterschiedliche CPUs, u.a. auch für Signalprozessoren, C++-Übersetzer, insbesondere Cross-Übersetzer. Ein Subsystem, das beim Endprodukt auf einem Spezialprozessor läuft, kann einschließlich der Benutzung des Subsystems auf dem Entwicklungssystem getestet werden. Sowohl die Kommunikation mit dem Subsystem als auch das Subsystem selbst können getrennt getestet werden.

Falls ein System mit unterschiedlichen Bibliotheken, die eigentlich dasselbe tun, zurecht kommen muß, bietet sich ein ähnlicher Weg wie bei der Schnittstelle zu einem Subsystem an. Ein Beispiel ist hier die Benutzung einer eingebetteten Datenbankabfragesprache unterschiedlicher Datenbanksysteme.

Abstrakte Klassen spielen bei der Definition von Frameworks und

Entwurfsmustern (*design patterns*) eine Schlüsselrolle. Darauf soll in einem späteren Artikel eingegangen werden.

Der Vorteil abstrakter Klassen besteht darin, daß der Übersetzer die Einhaltung der Schnittstelle sowohl aus der Sicht des Benutzers als auch aus der Sicht des Anbieters (abgeleitete Klasse) überprüft. Dabei kann der Übersetzer jedoch nur das Vorhandensein der jeweiligen Funktion überprüfen. Ob die Funktion auch semantisch richtig ist, bleibt das Geheimnis des Programmierers oder der Programmiererin.

Der Nutzen von abstrakten Klassen in C++ ist ohne das Konzept der virtuellen Funktionen und ohne Referenzen oder Zeiger nicht denkbar. Natürlich können auch Templates von abstrakten Klassen gebildet werden.

Zusammenfassung

Abstrakte Klassen dienen der Festlegung von Abstraktionsebenen und Schnittstellen in Systemen:

- Abstrakte Klassen in C++ dienen der Realisierung von Klassen, die bei der objektorientierten Modellierung als abstrakt identifiziert werden (siehe [1]).
- Ein System läßt sich in der Regel leicht durch eine weitere von der abstrakten Klasse abgeleiteten Klasse erweitern.
- Schnittstellen zu Subsystemen lassen sich gut durch abstrakte Klassen festlegen.
- Dadurch werden Subsysteme leichter austauschbar. Das unterstützt die Portierung, Anpassung und das Testen.
- Die Einhaltung der Schnittstellen wird durch den Übersetzer überprüft.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um*.

Literatur

- [1] J. Rumbaugh, M. Blaha, W. Premeriani, F. Eddy, W. Lorensen *Objektorientiertes Modellieren und Entwerfen* Carl Hanser Verlag, München, 1994.
- [2] A. Koenig (Ed.) *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++* Dokumentnr. X3J16/95-0087, WG21/N0687, 1995.
- [3] S. Seehusen *Programmieren in C++; Teil III, Templates: Wir fertigen Schablonen und gehen in Serie* **unix/mail** 4/95, 1995.
- [4] J. K. Ousterhout *Tcl und Tk. Entwicklung grafischer Benutzerschnittstellen für das X Windows System* Addison-Wesley, 1995.