Aspekt-Orientierte Programmierung mit C++

Georg Blaschke (georg.blaschke@stud.informatik.uni-erlangen.de)

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Betriebssysteme und Verteilte Systeme

17. Juni 2003

Um aspekt-orientierte Programmierung zu betreiben, ist es nicht unbedingt nötig spezielle Compiler einzusetzen oder den fachlichen Code mit dem Aspektcode vor der Übersetung mit Hilfe eines zusätzlichen Programms zusammenzuführen. Mit einem herkömmlichen C++ Compiler (GNU,VCC) ist es möglich, AOP in einem Projekt zu verwenden.

1 Einleitung und Motivation

Dieses Dokument beruht zum größten Teil auf einen dreiteiligen Bericht in der Zeitschrift iX[1], der eine Untersuchung der Programmiersprache C++ hinsichtlich ihrer Eignung für aspekt-orientiertes Programmieren beschreibt. Als Rahmenbedingungen gelten, dass aspekt-orientierter Code in Standard-C++ geschrieben ist, ohne zusätzliche Software (wie z.B. ein Präprozessor) auskommt, auf existierenden Quellcode anwendbar ist und sich mit GNU C++ 2.95.2 bzw. V++ 6.0 übersetzen läßt. Ausserdem soll der vorhandene Quellcode möglichst keinen Änderungen unterworfen sein.
Anhand von Beispielen werden nun einige Möglichkeiten aufgezeigt, Aspektcode mit fachlichen Code zu verküpfen. Dieser *gewobene* Code findet dann seine Anwendung im Clientcode, also in einer konkreten Anwendung. Abbildung 1 verdeutlicht die *Benutzt*-Beziehung zwischen den verschiedenen Arten von Code.

Abbildung 1: Beziehung zwischen verschiedenen Code-Arten

2 Basistechniken

Anhand der einfachen Implementierung eines Stacks (Listing 1) werden verschiedene Methoden erläutert, den Tracing-Aspekt auf eine C++-Klasse anzuwenden.

Listing 1: IntStack

```
class IntStack{
  public:
    IntStack(): size_(0){
    }
    void push(const int& i){
      elements_[size_++] = i;
```

```
    }
    int pop() {
      return elements_[--size_];
    }
  private:
    enum { max_size_ = 3 };
    int elements_[max_size_];
    int  size_ ;
};
```

2.1 Manipulation der Quellcodes

Die wohl simpelste Lösung, Aspekte auf Fachcode anzuwenden, ist, den Aspektcode direkt in den Fachcode hineinzuschreiben. Zu diesem Zweck wird eine Helferklasse implementiert, deren Konstruktor und Destruktor den Text *"Before"* bzw. *"After"* und den Übergedenen Funktionsnamen ausgeben. Der Grund, weswegen für eine Zeichenaugabe eine ganze Klasse implementiert wird, liegt darin, dass es in C/C++ nicht möglich ist, Code anzugeben, der nach der Beendigung einer Funktion ausgeführt werden soll. Durch die Verwendung der Helferklasse ReportTracing wird jedoch genau dies erreicht, da die Instanz der Klasse erst nach Ausführung des Funktionsrumpfes zerstört wird, somit also auch ihr Destruktor mit Tracing-Textausgabe ausgeführt wird.

Listing 2: Report Tracing

```
class ReportTracing // Hilfsklasse zur Implementierung des Tracing-Aspekts
{
  public:
    ReportTracing(const char* function): function_(function){
      cout << "Before " << function_ << endl;
    }
    ~ReportTracing(){
      cout << "After " << function_ << endl;
    }
  private:
    const char* function_;
};

class IntStack
{
  public:
    IntStack(): size_(0){
      ReportTracing t("constructor");
    }
    void push(const int& i){
      ReportTracing t("push");
      elements_[size_++] = i;
      return;
    }
    int pop(){
      ReportTracing t("pop");
      return elements_[--size_];
    }
  private:
    enum { max_size_ = 3 };
    int elements_[max_size_];
    int  size_ ;
};
```

Der grobe Eingriff in den Quellcode verstößt natürlich gegen eine der Voraussetzung, und hat eigentlich nichts mit AOP zu tun, da der Aspekt nicht zentral gekapselt ist, sondern in an mehreren Stellen den fachspezifischen Code durchzieht. Dies ist in Listing 1 durch einen Unterstrich gekennzeichnet.

2.2 Komposition mit Weiterleitung

Um einen Aspekt seperat vom Fachcode zu implementieren, kann man dessen Code in eine eigene Klasse schreiben, die alle Member-Funktionen der Fachklasse implementiert und den Aufruf an die jeweilige Methode der Fachklasse weiterleitet.

Listing 3: Komposition mit Weiterleitung

```
// Klasse IntStack wie in Listing 1

class IntStackWithTracing{
  public:
    IntStackWithTracing(){
      ReportTracing t("constructor");
      myStack = new IntStack();
    }
    void push(const int& i){
      ReportTracing t("push");
      myStack->push(i);
    }
    int pop(){
      ReportTracing t("pop");
      return myStack->pop();
    }
  private:
    IntStack *myStack;
};
```

Die Komposition von Fach- und Aspektcode unter Verwendung von Funktionsaufruf-Weiterleitung hat den Nachteil, dass alle Methoden der Fachklasse implementiert werden müssen. Ausserdem muss der Programmierer jedesmal wenn eine neue Funktion dem Fachcode hinzugefügt wird, auch den Aspektcode entsprechend erweitern.

2.3 Komposition mit Vererbung

Um den Nachteilen zu begegnen, die durch Verwendung von Weiterleitung der Funktionsaufrufe entstehen, wird nun Vererbung eingesetzt. Somit kann die Fachklasse nach Belieben erweitert werden und die Aspektklasse muss nicht alle Methoden von ihr implementieren.

Listing 4: Komposition mit Vererbung

```
// Klasse IntStack wie in Listing 1

class IntStackWithTracing: public IntStack{
  public:
    IntStackWithTracing(){
      ReportTracing t("constructor");
    }
    void push(const int& i) {
      ReportTracing t("push");
      IntStack::push(i);
    }
    int pop(){
      ReportTracing t("pop");
```

```
        return IntStack::pop();
    }
};
```

Der Nachteil dieser Art der Komposition ist, dass es nicht ohne weiteren Aufwand möglich ist Code zu spezifizieren der **vor** einem Konstruktor ausgeführt wird. Bei der Instanziierung der Klasse IntStackWithTracing aus Listing 4 wird die Ausgabe *"Before constructor"* erst erscheinen, wenn der Konstruktor der Eltern-Klasse (hier: IntStack) abgearbeitet ist. Eine Lösung zu diesem Problem wird später noch vorgestellt.

2.4 Verwendung von Namesräumen

Um Aspekte anzuwenden musste bisher der Clientcode so angepasst werden, dass er die Klasse instanziiert, die genau diesen Aspekt implementiert; z.B IntStackWithTracing. Diese Verfahrensweise widerspricht jedoch der Voraussetzung, dass kein Quellcode verändert werden darf, also auch nicht der Code der Anwendung. Um den Clientcode nicht ändern zu müssen, muss die Aspektklasse den selben Namen besitzten wie die Fachklasse, wodurch jedoch ein Namenskonflikt auftritt. Gelöst wird dieser Konflikt indem geringfügige Änderungen am Fachcode zugelassen werden, so dass dieser in einem eigenen Namensraum (**namespace** original) definiert ist. Aspektklassen und dazugehörige Helferklassen befinden analog im **namespace** aspects. Der Clientcode importiert mittels **using namespace** composed den Namensraum, in dem die Klasse IntStack definiert ist; entweder mit Aspekten oder ohne. Um einen Aspekt mit hinzuzunehmen, oder wegzulassen, muss man nur im Namensraum composed den Typ IntStack entsprechend definieren.

Listing 5: Verwendung von Namensräumen

```
namespace original{
  // Klasse IntStack wie in Listing 1
}
namespace aspects{
  //Klasse Report Tracing wie in Listing 2

  typedef original::IntStack IntStack;

  class IntStackWithTracing: public IntStack
  {
    public:
      IntStackWithTracing(){
        ReportTracing t("constructor");
      }
      void push(const int& i) {
        ReportTracing t("push");
        IntStack::push(i);
      }
      int pop(){
        ReportTracing t("pop");
        return IntStack::pop();
      }
  };
}

namespace composed{
  // typedef aspects::IntStack IntStack; // "pure" IntStack
  // typedef aspects::IntStackWithChecking Stack; // IntStack with Checking (not shown here)
  typedef aspects::IntStackWithTracing IntStack; // Stack with Tracing
}
```

3 Zwischenstand

Die Techniken die bisher gezeigt wurden (Helferklasse, Vererbung, Namensräume) ermöglichen es, einzelne Aspekte auf Klassen (und auch Klassentemplates) anzuwenden.
Der momentane Stand erlaubt jedoch noch nicht Aspekte mit mehrere Klassen oder mehrere Aspekte mit einer Klasse zu verküpfen. Ausserdem stellt sich noch die Frage wie man Aspektcode angibt, der vor einem Konstruktor oder nach einem Destruktor ausgeführt werden soll.
Techniken für die angesprochnen Probleme werden im Folgenden vorgestellt.

4 Techniken zur Lösung spezieller Probleme

4.1 Ausführung von Code vor einem Konstruktor oder nach einem Destruktor

Bei der Kompostition mit Vererbung (siehe Kapitel 2.3) hat man die Einschränkung, keinen Code spezifizieren zu können, der vor dem Konstruktor oder nach dem Destruktor der Fachklasse ausgeführt werden soll.
Um diese Limitierung zu beseitigen, werden zwei weitere Hilfsklassen BeforeConstructor und AfterConstructor eingeführt. Die Aspektklasse IntStackWithTracing wird nun von den Klassen BeforeConstructor, AfterConstructor und IntStack abgeleitet; und zwar in genau diese Reihenfolge ! So wird bei einer Instanzierung der Aspektklasse zuerst der Konstruktor von BeforeConstructor, dann der Konstruktor der Fachklasse und zuletzt der Konstruktor der Aspektklasse abgearbeitet. Ein ähnliches Verhalten für die Destruktoren findet man bei der Zerstörung einer Instanz der Aspektklasse.

Listing 6: Konstruktoren und Destruktoren

```
namespace original{
  // Klasse IntStack wie in Listing 1
}
namespace aspects{

  typedef original::IntStack IntStack;

  class BeforeConstructor{
    public:
      BeforeConstructor(){
        cout << "Before constructor" << endl;
      }
  };

  struct AfterDestructor{
    ~AfterDestructor(){
      cout << "After destructor" << endl;
    }
  };
  class IntStackWithTracing: private BeforeConstructor, private AfterDestructor,public IntStack{
    public:
      Tracing(){
        cout << "After constructor" << endl;
      }

      void push(const int& i) {
        IntStack::push(i);
      }

      valueType pop() {
        return IntStack::pop();
      }
```

```
        ~Tracing(){
          cout << "Before destructor" << endl;
        }
  };
}
```

Die **private**-Vererbung von BeforeConstructor und AfterConstructor dient dazu, die Verwendung von Methoden der Hilfsklassen in anderen Klassen, die eventuell von IntStack abgeleitet werden, zu verhindern.

4.2 Anwendung von einem Aspekt auf mehrere Klassen

Wenn man einen Aspekt mit mehreren Fachklassen verweben will, darf man den Aspektcode nicht abhängig von einer Fachklassen machen, so wie es bisher geschah. Um den Aspektcode flexibler zu gestalten, besteht die Möglichkeit Aspekte in Form von Klassen-Templates zu realisieren.
Wie in Listing 7 zu sehen ist, wird die Aspektklasse von der Klasse abgeleitet, die ihr als Template-Parameter übergeben wurde. Probleme treten nun bei den Funktionen auf, deren Paramerter- oder Rückgabetypen von der konkreten Fachklasse abhängen; in unseren Fall **double** oder **int**. Mit *Traits Templates* ist es jedoch möglich den Typ zur Übersetzungszeit herauszufinden. *Traits Classes* und *Member Traits* erscheinen hier nicht als geeignet, da man hierfür Code der Fachklasse verändern müsste. Mehr zu Thema Metainformation findet man in [1], [3] und in einschlägiger Fachliteratur. Für jeden verwendeten Stack-Typ wird im Namensraum aspects ein spezialisiertes Klassen-Template von ValueType angelegt, welches die Information über den entsprechenden Stack-Typ beinhaltet.

Listing 7: flexibler Aspektcode

```
namespace original{
  // Klasse IntStack wie in Listing 1
  class DoubleStack
     //gleicher Aufbau wie IntStack
}
namespace aspects{
  template <class T>
     struct ValueType
     {};

  template<>
     struct ValueType<original::IntStack>
     {
       typedef int RET;
     };

  template<>
     struct ValueType<original::DoubleStack>
     {
       typedef double RET;
     };


  // ReportTracing wie in Listing 2

  template <class Base>
     class Tracing: public Base{
        public:
           typedef typename ValueType<Base>::RET valueType;

           void push(const valueType& i){
             ReportTracing t("push");
```

```
            Base::push(i);
          }

          valueType pop(){
            ReportTracing t("pop");
            return Base::pop();
          }
      }
}

namespace composed
{
  // typedef original::IntStack IntStack;
  typedef aspects::Tracing<original::IntStack> IntStack;

  typedef original::DoubleStack DoubleStack;
  //typedef aspects::Tracing<original::DoubleStack> DoubleStack;
}
```

4.3 Verweben von mehreren Aspekten mit mehreren Klassen

Bisher wurde zu einer Fachklasse immer nur ein Aspekt hinzugefügt. Nun wird eine Möglichkeit vorgestellt, mehrere Aspekte mit einer Klasse zu verküpfen; in unserem Beispiel (Listing 8) werden die Aspekte Checking und/oder Tracing den Fachklassen DoubleStack und/oder IntStack hinzugefügt. Wie in Kapitel 4.2 werden die Aspekte als Klassen-Templates implementiert. Wenn zwei Aspekte auf eine Klasse angewendet werden sollen, hat man bei der zweiten Anwendung das Problem, dass für die übergebene Klasse keine Typ-Information vorhanden ist. Eine Lösung wäre somit für alle vorkommenden Verküpfungs-Kombinationen(IntStack, IntStack+Tracing, IntStack+Checking, IntStack+Checking+Tracing, IntStack+Tracing+Checking, DoubleStack,...) die Typ-Information bereit zu stellen, was jedoch schnell ausufern würde. Eine bessere Möglichkeit bietet der *inheritance detector* der zur Übersetzungszeit Vererbungsbeziehungen zwischen zwei Klassen feststellt. Mit dem *inheritance detector* und der Meta-Kontrollstruktur IF kann nun die richtige Typ-Information (ValueTyp) ermittelt werden. Nähere Informationen zum Thema Meta-Funktionen findet man in [3]. Jede Fachklasse, auf die irgendein Aspekt angewendet werden soll, muss mit der Klasse, die der Aspektklasse als Template-Parameter übergeben wurde, bezüglich ihrer Vererbungsbeziehung hin überprüft werden. Fällt dieser Test positiv aus, wird die Typ-Information der positiv getesten Fachklasse zurückgegeben.

Listing 8: Weben mehrerer Aspekte

```
namespace original{
  // Klasse IntStack wie in Listing 1
  class DoubleStack
     // gleicher Aufbau wie IntStack
}
namespace aspects{
  // ValueType wie in Listing 7

  // inheritance detector: siehe naechsten kommentar
  template<class Base>
     struct InheritDetector
     {
       typedef char (&no)[1];
       typedef char (&yes)[2];
       static yes test( Base* );
       static no  test( ... );
```

```
    };

// inheritance detector: ermittelt ob die Klasse Derived von Base abgeleitet ist
template<class Derived, class Base>
    struct Inherits
    {
      typedef Derived* DP;
      enum
      {
        RET =
            sizeof( InheritDetector <Base>::test(DP()) ) ==
            sizeof( InheritDetector <Base>::yes ) };
    };

// meta-IF
struct SelectThen
{
  template<class Then, class Else>
      struct Result
      {
        typedef Then RET;
      };
};

//meta-IF
struct SelectElse
{
  template<class Then, class Else>
      struct Result
      {
        typedef Else RET;
      };
};

//meta-IF
template<bool condition>
    struct Selector
    {
      typedef SelectThen RET;
    };

//meta-IF
template<>
    struct Selector<false>
    {
      typedef SelectElse RET;
    };

//meta-IF
template<bool condition, class Then, class Else>
    struct IF
    {
      typedef typename Selector<condition>::RET Selector_;
      typedef typename Selector_::Result<Then,Else>::RET RET;
    };

//Ermittelt Typ-Information der Klasse T
template <class T>
    struct GetType
```

```
    {
      typedef typename
        IF<(Inherits<T,original :: IntStack>::RET),
           ValueType<original :: IntStack>::RET,
           typename IF<(Inherits<T,original::DoubleStack>::RET),
                       ValueType<original :: DoubleStack>::RET,
                       void
                      >::RET
         >::RET RET;
    };
```

```
// ReportTracing wie in Listing 2

//Tracing Aspekt
template <class Base>
    class Tracing: public Base{
      public:
        typedef typename GetType<Base>::RET valueType;

        void push(const valueType& i){
          ReportTracing t("push");
          Base::push(i);
        }

        valueType pop(){
          ReportTracing t("pop");
          return Base::pop();
        }
    };

//Hilsklasse fuer den Checking Aspekt
class CheckingException: public exception
{
    public:
        CheckingException(const char* msg):msg_(msg)
        {}
        const char* what() const throw()
        {
          return msg_;
        }
    private:
        const char* msg_;
};

//Checking Aspekt
template <class Base>
    class Checking: public Base
    {
        public:
            typedef typename GetType<Base>::RET valueType;
            void push(const valueType& i)
            {
              if (size() == capacity())
                throw CheckingException("stack overflow");
              Base::push(i);
            }
            valueType pop()
            {
```

```
            if ( size () <= 0)
              throw CheckingException("stack underrun");
            Base::pop();
          }
      };

}

namespace composed
{
  // typedef  original :: IntStack  IntStack;
  // typedef  aspects :: Tracing< original :: IntStack> IntStack;
  // typedef  aspects :: Checking< original :: IntStack> IntStack;
  // typedef  aspects :: Checking<aspects::Tracing< original :: IntStack> > IntStack;
  typedef aspects :: Tracing<aspects::Checking<original :: IntStack> > IntStack;

  // typedef  original :: DoubleStack DoubleStack;
  // typedef  aspects :: Tracing< original :: DoubleStack> DoubleStack;
  // typedef  aspects :: Checking< original :: DoubleStack> DoubleStack;
  // typedef  aspects :: Checking<aspects::Tracing< original :: DoubleStack> > DoubleStack;
  typedef aspects :: Tracing<aspects::Checking<original :: DoubleStack> > DoubleStack;
}
```

5 Zusammenfassung

Viele anschauliche Beispiele haben gezeigt, dass Aspekt-orientierte Programmierung mit „reinem“C++ unter den geforderten Bedingungen durchaus möglich ist. Jedoch ist dieser Ansatz eher zu vergleichen mit Objekt-orientierter Programmierung in der Programmiersprache C, welcher einfach die Sprachmittel fehlen um OO-Konzepte übersichtlich und effizient zu realisieren.
Nun muss man sich die Frage stellen welche Sprachmittel nötig sind um effizient AOP zu betreiben. Mit AspectJ ist ja schon ein großer Schritt unternommen worden genau diese Sprachmittel zu identifizieren. Daran müssen sich jetzt andere Konzepte messen, was keine leichte Aufgabe darstellt.

Literatur

[1] Ulrich W. Eisenecker, Lutz Dominik, Krzysztof Czarnecki, *Aspektorierte Programmierung in C++* , iX, Ausgabe 8/01 bis 10/01, Heise-Verlag, 2001.

[2] Gregor Kiczales, et al., *Aspect-Oriented Programming*, Springer-Verlag LNCS 1241, 1997.

[3] Ulrich W. Eisenecker, Krzysztof Czarnecki, *Template-Metaprogrammierung. Eine Einführung*, OBJEKTspektrum, Ausgabe Mai/Juni, 1998.