



Aspekt-Orientierte Programmierung mit C++

Georg Blaschke

`georg.blaschke@stud.informatik.uni-erlangen.de`



Übersicht

- Randbedingungen
- Basistechniken
- spezielle Techniken
- nicht behandelte Themen



Motivation

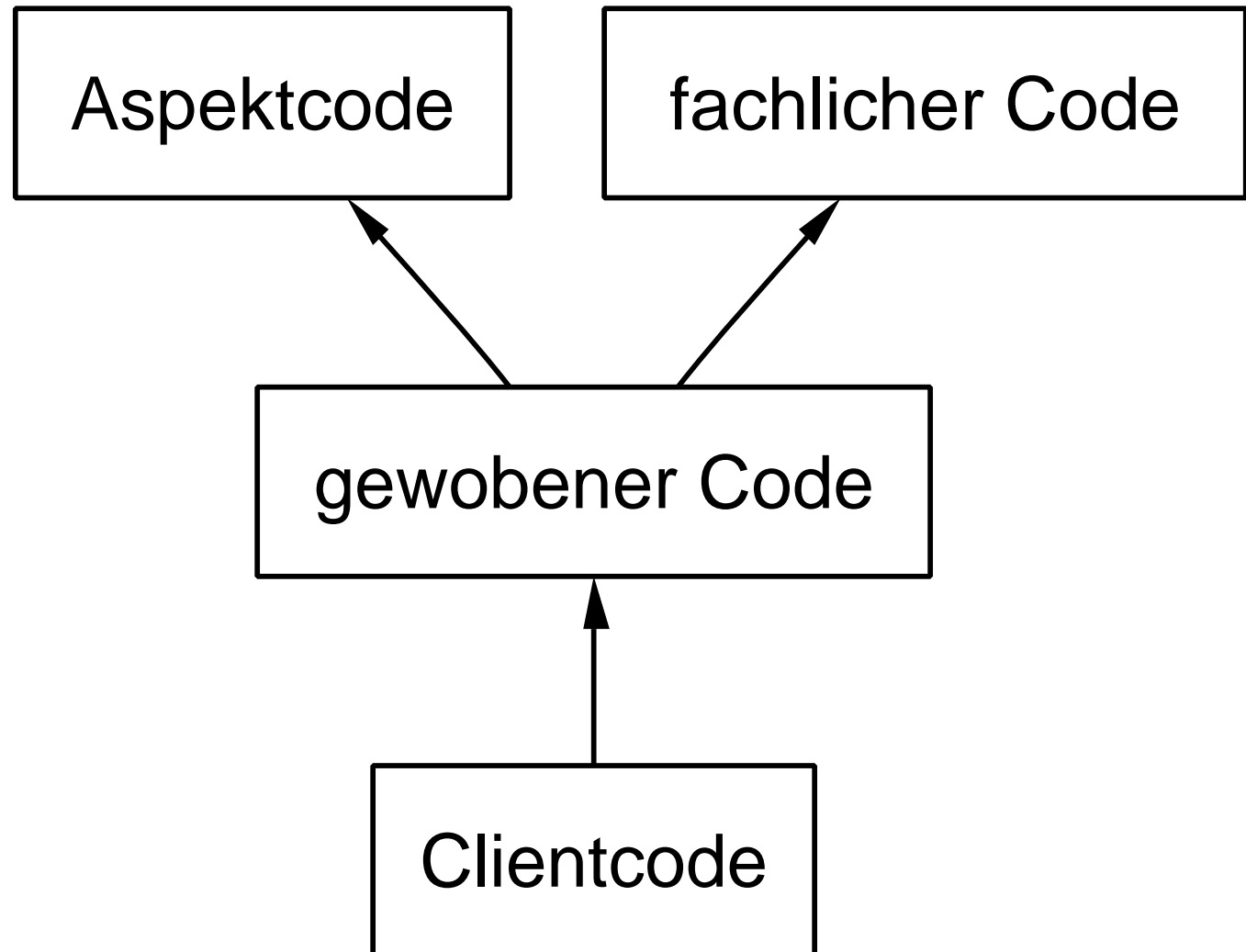
- AOP beeinflußt positiv
 - Wartbarkeit
 - Wiederverwendbarkeit
 - Erweiterbarkeit
- AOP mit C++
 - bisher keine funktionierende bekannte Lösung
 - keine Tools \Rightarrow schwierige Programmierung



Randbedingungen

- kein spezieller Präprozessor
- Implementierung nur in Standard-C++
- möglichst wenig Änderungen an bereits existierendem Quellcode
- verwendete Compiler
 - GNU C++ 2.95.2
 - V++ 6.0

Code Arten





Basistechniken

- verschiedenen Techniken zur Code-Komposition
 - Manipulation des Quellcodes
 - Funktionsaufruf-Weiterleitung
 - Vererbung
- Verwendung von Namenräumen



Beispiel-Klasse: IntStack

```
1  class IntStack{
2      public :
3          IntStack (): size_(0){
4              }
5          void push(const int & i){
6              elements_[size_++] = i;
7          }
8          int pop() {
9              return elements_[--size_];
10         }
11     private :
12         enum { max_size_ = 3 };
13         int elements_[max_size_];
14         int size_;
15 };
```



Q-Manipulation (1)

- hat nichts mit AOP zu tun
- in C++ keine Spezifikation von Code möglich, dessen Ausführung nach einem return stattfindet.
- Aber: lokale Objekte werden nach abarbeitung des Funktionsrumpfes zerstört
- Helferklasse ReportTracing



ReportTracing

```
1  class ReportTracing {
2      public :
3          ReportTracing(const char* function ): function_(function){
4              cout << "Before " << function_ << endl;
5          }
6          ~ReportTracing(){
7              cout << "After " << function_ << endl;
8          }
9      private :
10         const char* function_;
11 };
```

Quellcode-Manipulation(2)

```
1  class IntStack{
2      public : IntStack (): size_(0){
3          ReportTracing t("constructor ");
4      }
5      void push(const int & i){
6          ReportTracing t("push");
7          elements_[size_++] = i;
8          return ;
9      }
10     int pop(){
11         ReportTracing t("pop");
12         return elements_[--size_];
13     }
14     private :
15         enum { max_size_ = 3 };
16         int elements_[max_size_];
17         int size_;
18 };
```



Weiterleitung (1)

- Vorteile

- Leichte Spezifikation von Code, der vor dem Fachklassen-Konstruktor ausgeführt wird
- Funktions-Aufruf Semantik (AspektJ: Call)

- Nachteile

- implementiert alle Methoden der Fachklasse
- schlechte Erweiterbarkeit der Fachklasse

Weiterleitung (2)

```
1  class IntStackWithTracing{
2      public :
3          IntStackWithTracing(){
4              ReportTracing t( "constructor");
5              myStack = new IntStack();
6          }
7          void push(const int & i){
8              ReportTracing t( "push");
9              myStack->push(i);
10         }
11         int pop(){
12             ReportTracing t( "pop");
13             return myStack->pop();
14         }
15     private :
16         IntStack *myStack;
17 };
```



Vererbung (1)

- Nachteile
 - Konstruktor-Problem
- Vorteile
 - Fachklasse veränderbar
 - Aspektklasse muss nicht alle Methoden implementieren

Vererbung (2)

```
1  class IntStackWithTracing: public IntStack{
2      public :
3          IntStackWithTracing(){
4              ReportTracing t( "constructor");
5          }
6          void push(const int & i ) {
7              ReportTracing t( "push");
8              IntStack :: push(i);
9          }
10         int pop(){
11             ReportTracing t( "pop");
12             return IntStack :: pop();
13         }
14     };
```



Namensräume (1)

- Bisher: Clientcode muss gewünschte Aspektklasse instanziiieren
 - nicht transparent
 - Veränderung von existentem Quellcode
- Einführung von verschiedenen Namensräumen
 - original: fachlicher Code
 - aspects: Aspektcode
 - composed: Auswahl der Aspekte
- Client importiert Bezeichner aus dem Namensraum composed

Namensräume (2)

```
1 namespace original{
2     // Fachklasse: hier IntStack
3 }
4 namespace aspects{
5     typedef original :: IntStack IntStack;
6     // Aspektklasse: hier IntStackwithTracing
7 }
8 namespace composed{
9     // typedef aspects::IntStack IntStack ; // " pure" IntStack
10    // typedef aspects::IntStackWithChecking Stack; // IntStack with Checking
11    typedef aspects::IntStackWithTracing IntStack; // Stack with Tracing
12 }
```




Zwischenstand

- Basistechniken
 - Helferklassen für Ausführungsabfolge
 - Komposition mit Vererbung
 - Verwendung von Namensräumen
- Möglichkeit Aspekte auf Klassen anzuwenden



spezielle Probleme

- Codeausführung vor und nach einem Konstruktor/Destruktor
- Verweben eines Aspekts mit mehreren Klassen
- Verweben mehrerer Aspekte mit mehreren Klassen




Konstruktor / Destruktor (1)

- zwei Helferklassen
 - BeforeKonstruktor
 - AfterDestructor
- Aspektklasse wird von Helferklassen abgeleitet

Konstruktor / Destruktor (2)

```
1  class BeforeConstructor{
2      public :
3          BeforeConstructor()
4              {cout << "Before constructor" << endl;}
5  };
6  struct AfterDestructor{
7      ~AfterDestructor()
8          {cout << "After destructor" << endl;}
9  };
10 class IntStackWithTracing: private BeforeConstructor,
11                             private AfterDestructor, public IntStack{
12     public : Tracing()
13         {cout << "After constructor" << endl;}
14         ~Tracing()
15         {cout << "Before destructor" << endl;}
16         /* ... */
17 };
```



Asp.:Kl. = 1:n (1)

- flexiblere Aspektklasse
- Einsatz von Klassen-Templates
- Problem: Parameter- und Rückgabetypen von Funktionen
- Lösung: Traits-Templates
- Nachteil: erfordert pro Fachklasse ein spezialisiertes Klassen-Template

Asp.:Kl. = 1:n (2)


```
1 namespace aspects{
2     template <class T>
3         struct ValueType
4             {};
5     template<>
6         struct ValueType<original::IntStack>
7         {
8             typedef int RET;
9         };
10    template<>
11        struct ValueType<original::DoubleStack>
12        {
13            typedef double RET;
14        };
15    /* ... */
16 }
```

Asp.:Kl. = 1:n (3)

```
1  template <class Base>
2  class Tracing: public Base{
3      public :
4          typedef typename ValueType<Base>::RET valueType;
5
6          void push(const valueType& i){
7              ReportTracing t( "push");
8              Base::push(i);
9          }
10
11         valueType pop(){
12             ReportTracing t( "pop");
13             return Base::pop();
14         }
15 }
```

Asp.:Kl. = 1:n (4)

```
1 namespace composed
2 {
3     // typedef original :: IntStack IntStack;
4     typedef aspects::Tracing<original :: IntStack> IntStack;
5
6     typedef original :: DoubleStack DoubleStack;
7     //typedef aspects::Tracing<original :: DoubleStack> DoubleStack;
8 }
```

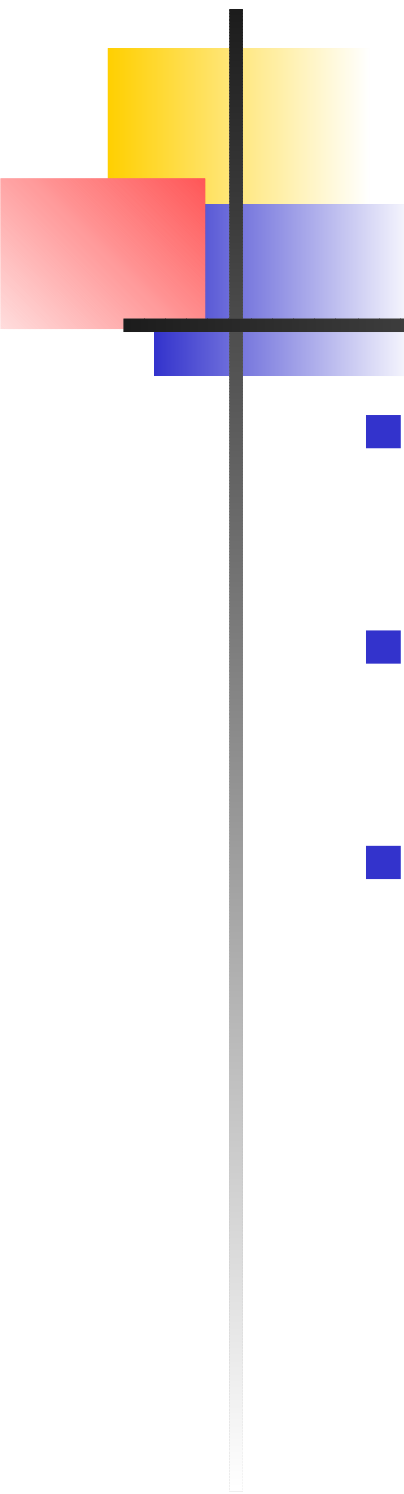
Asp.:Kl. = n:n (1)

- gewünscht:

```
typedef aspects::Tracing<aspects::Checking<original::IntStack> > IntStack;
```

- Aber:

Für `aspects::Checking<original::IntStack>`
gibt es keine Typinformation



Asp.:Kl. = n:n (1)

- Lösung 1: Typinformation für alle erdenklichen Typen bereitstellen
- Lösung 2: die Original-Klasse ermitteln (BESSER)
- Realisierung von Lösung 2 mit Meta-IF und *inheritance detector*

Asp.:Kl. = n:n (2)

```
1  template<class Base>
2      struct InheritDetector
3  {
4      typedef char (&no)[1];
5      typedef char (&yes)[2];
6      static yes test ( Base* );
7      static no test ( ... );
8  };
9  template<class Derived, class Base>
10     struct Inherits
11     {
12         typedef Derived* DP;
13         enum {RET = sizeof( InheritDetector<Base>::test(DP()) ) ==
14                 sizeof ( InheritDetector<Base>::yes )
15     };
16     };
```

Asp.:Kl. = n:n (3)

```
1  template <class T>
2      struct GetType
3      {
4          typedef typename
5              IF<( Inherits<T, original :: IntStack>::RET),
6                  ValueType<original::IntStack>::RET,
7                  typename IF<( Inherits<T, original :: DoubleStack>::RET),
8                      ValueType<original::DoubleStack>::RET,
9                          void
10                             >::RET
11                  >::RET RET;
12      };
```

Asp.:Kl. = n:n (4)

```
1  template <class Base>
2      class Tracing: public Base{
3      public :
4          typedef typename GetType<Base>::RET valueType;
5
6          void push(const valueType& i){
7              ReportTracing t( "push");
8              Base::push(i);
9          }
10
11         valueType pop(){
12             ReportTracing t( "pop");
13             return Base::pop();
14         }
15     };
```

Asp.:Kl. = n:n (5)

```
1 namespace composed
2 {
3     // typedef original :: IntStack IntStack;
4     // typedef aspects::Tracing<original :: IntStack> IntStack;
5     // typedef aspects::Checking<original::IntStack> IntStack;
6     // typedef aspects::Checking<aspects::Tracing<original::IntStack> > IntStack;
7     typedef aspects::Tracing<aspects::Checking<original::IntStack> > IntStack;
8
9     // typedef original :: DoubleStack DoubleStack;
10    // typedef aspects::Tracing<original :: DoubleStack> DoubleStack;
11    typedef aspects::Checking<original::DoubleStack> DoubleStack;
12    // typedef aspects::Checking<aspects::Tracing<original::DoubleStack> > DoubleStack;
13    // typedef aspects::Tracing<aspects::Checking<original::DoubleStack> > DoubleStack;
14 }
```



Hier nicht behandelt

- Unterscheidung zwischen
 - Ausführung einer Funktion
 - Aufruf einer Funktion
- Aspekte für freie Funktionen
- Ausführung von Aspektcode nur bei Fehlern



Zusammenfassung

- AOP unter den geforderten Bedingung möglich
- vergleichbar mit OOP unter C
- Messlatte wegen AspectJ sehr hoch