Oberseminar Softwareentwicklung

Class – Design (Christian Wanka)

„Yet nothing is more important than clear, intuitive interfaces in assuring that code will be readily understood and correctly maintained.“

Stephen C. Dewhurst

```
template <class T>
class UnusableStack {
  public :
    UnusableStack();
    ~UnusableStack();
    T *getStack();
    void setStack( T * );
    int getTop();
    void setTop( int );

  private :
    T *s_;
    int top_;
};
```

schlechtes Beispiel:
- keine Abstraktion
- dünn gesähte Datenmenge
- keine Implementationsunabhängigkeit

saubere Lösung:

```
template <class T>
class Stack {
  public:
    Stack();
    ~Stack();
    void push( const T & );
    T &top();
    void pop();
    bool empty() const;
  private:
    T *s_;
    int top_;
};
```

positives Beispiel:
- saubere Abstraktion
- Implementationsunabhängig

Get/set Interfaces

viele Klassen – Interfaces nur aus get/set (sog. accessor – Funktionen)
Möglichkeiten der Gestaltung:

```
class C {
  public:
    int getValue1() const        // get/set style 1
        { return value_; }
    void set Value1( int value )
        { value_ = value; }
    int &value2()   // get/set style 2
        { return value_; }
    int setValue3( int value )// get/set style 3
        { return value_ = value; }
    int value4( int value ) {   // get/set style 4
        int old = value_;
        value_ = value;
        return old;
    }
  private:
    int value_;
};
```

- 2. Style flexibelster aber auch gefährlichster → Rückgabe eines veränderbaren privaten Datums (besser als public)
- eventl. Abhängigkeiten der aktuellen Impl. und dem direkten Zugriff auf direkte Daten

Betrachtung am Beispiel einer Klasse unter Nutzung eines Standard - containers:

```
class Users {
  public:
    const std::map<std::string,User> &getUserContainer() const
        { return users_; }
    // . . .
  private:
    std::map<std::string,User> users_;
};
```

Die get – Funktion kennt die ziemlich private Information, dass der user – container mit einer standard map implementiert wurde.
Jeder die öffentliche Funktion aufrufender Code kann eine Abhängigkeit in der Implementierung von users verursachen.
Wird ein Vektor zur Implementierung verwendet, müssen alle user der Users – Klasse neu geschrieben werden!

Der 3. Style ist etwas ungewöhnlich, er ermöglicht nicht den Zugriff auf den aktuellen Wert des Datums. Aber er ermöglicht gleichzeitiges Setzen und Rückgeben des neu geschriebenen Wertes.

```
        a += setValue3(12)
→       setValue1(12); a += getValue1();
```

! mögliche Fehlinterpretation des Rückgabewertes

Der 4. Style ermöglicht das Rückgeben des aktuellen Wertes und das Setzen eines neuen Wertes.

- Rückgabe aktueller Wert mit Trick:

```
int current = c.value4( 0 );  // get and set
c.value4( current );  // restore
```

<u>Anwendung:</u> Callback – Funktionen

- generelle Anwendung nicht empfehlenswert
- Erhöhung der Komplexität und Kosten

Fazit:

Der erste get/set – Style ist zu bevorzugen.

- simpelster zur Verfügung stehender Mechanismus
- effizient und leicht zu verstehen

```
int a = c.getValue1(); // get, of course
c.setValue1( 12 );  // set, of course
```

Const and Reference Data Members

Zwei generelle Ratschläge:

„Anything that can be const should be const.“
“If something is not always used as a const, don’t declare it to be const.”

→”as const as possible, but no more so”

+ : kann gefährliche Änderungen verhindern

- : Klassen schwerer zu händeln
schwierige Copy – Operationen

```
class C {
  public:
    C();
    // . . .
  private:
    int a_;
    const int b_;
    int &ra_;
};
```

Der Konstruktor muss Konstanten und Referenzparameter initialisieren:

```
C::C()
        : a_( 12 ), b_( 12 ), ra_( a_ )
        {}
```

Objekte deklarieren und initialisieren:

```
C x; // default ctor
C y( x ); // copy ctor
```

! Kopiekonstruktor

- vom Compiler erzeugt
- member-by-member – Initialisierung

→ setzen der ra_ Referenz in y auf a_ in x → Copyoperationen selber schreiben

```
C::C( const C &that )
        : a_( that.a_ ), b_( that.b_  ), ra_( a_ )
        {}
```

```
x = y; // error !
```

Selbes Prinzip: b_ und ra_ können nicht zugewiesen werden

```
C &C::operator =( const C &that ) {
  a_ = that.a_;  // OK
  b_ = that.b_;  // error!
  return *this;
}
```

- Zuweisung zu einer Konstanten verboten

mögliche Auswege:

```
int *pb = const_cast<int *>(&b_);
*pb = that.b_;  // b in read-only segment, in einem non-constant C object !
```

Hinweis: der Referenzparameter von C muss nicht neu belegt werden, da er schon auf a_ des eigenen Objekts verweist.

```
C( const C & ); // copy – Konstruktor verbieten
C &operator = ( const C & ); // assignment verbieten
```

Fazit: Konstanten oder Referenzparameter als member sind zu vermeiden !

Bedeutung von Const Member Functions

- const hinter der Deklaration

```
class BoundedString {
  public:
    explicit BoundedString( int len );
    // . . .
    size_t length() const;
    void set( char c );
    void wipe() const;
  private:
    char * const buf_;
    int len_;
    size_t maxLen_;
};
```

- privates member buf_ als konstanter Pointer auf character
- Pointer ist konstant, nicht der character auf welchen er zeigt
- const Angabe hinter Zeigersyntax

- gleiches Prizip bei const member function length
- const hinter dem Funktionskopf bedeutet, dass die Funktion const ist, nicht der Rückgabewert

Schlussfolgerung für die member – Funktion ?

Eine const member – Funktion darf sein Objekt nicht verändern

Jede nicht statische member – Funktion hat ein impizites Argument, ein Pointer auf das Objekt, welches die member – Funktion aufruft.

- this Schlüsselwort beinhaltet Wert

```
BoundedString bs( 12 );
cout << bs.length(); // “this” is &bs
BoundedString *bsp = &bs;
cout << bsp->length(); // “this” is bsp
```

- für non-const member – Funktionen von Klasse X Typ des this – Pointer: X * const
  → konstanter Pointer auf non-constant X
- Zeiger verweist immer auf das eine Objekt
- Werte des Objektes dürfen sich ändern

- für const member – Funktionen von Klasse X Typ des this – Pointer: const X * const
  → konstanter Pointer auf constant X
- weder Zeiger noch Objekt dürfen geändert werden

```
size_t BoundedString::length() const
  { return strlen( buf_ ); }
```

const member – Funktionen ermöglichen das implizite Argument, den this Pointer, als konstant in einer member – Funktion zu deklarieren.

non-member – Funktion:
```
  bool operator ==( const BoundedString &lhs, const BoudedString &rhs );
```

- ändert nicht ihre Argumente
→ deklarieren als const

Analog member – Funktionen:

```
clas BoundedString {
  // . . .
  bool operator <( const BoundedString &rhs );
  bool operator >=( const BoundedString &rhs ) const;
};
```

linkes Argument = this // implizit bei überladener member operator – Funktion
rechtes Argument = rhs

>= : sauber deklarierte Funktion

< : garantiert die Unveränderbarkeit des rechten Arguments
    linkes Arguments nicht konstant

→ Beispiel mit Compile – Fehler

```
bool BoundedString::operator >=( const BoundedString &rhs ) const
        { return !(*this < rhs); }
```

*this – Ausdruck bei < - Operator: Versuch des Initialisieren des this – Pointer einer non-const member – Funktion mit der Adresse eines konstanten Objekts

unsauberes Überladen eines Operators

"It's possible to get by without operator overloading":

```
class Complex {
  public:
    Complex( double real = 0.0, double imag = 0.0 );
    friend Complex add( const Complex &, const Complex & );
    friend Complex div( const Complex &, const Complex & );
    friend Complex mul( const Complex &, const Complex & );
   // . . .
};
// . . .
Z = add( add( R, mul( mul( j, omega ), L ) ), div( 1, mul( j, omega ), C ) )
);
```

→ Operator überladen:

- syntakischer Zucker
- leichter zu lesen
- richtig eingesetzt Intension besser erkennbar

```
class Complex {
  public:
    Complex( double real = 0.0, double imag = 0.0 );
    friend Complex operator +( const Complex &, const Complex & );
    friend Complex operator *( const Complex &, const Complex & );
    friend Complex operator /( const Complex &, const Complex & );
   // . . .
};

// . . .
Z = R + j*omega*L + 1/(j*omega*C);
```

Infix – Version ist korrekt
Präfix – Version ist es nicht !

Fehler ist schwer zu sehen bzw. zu korrigieren

gerechtfertigt auch bei Standard template libraries und iostream:

```
ostream &operator <<( ostream &os, const Complex &c )
        { return os << '(' << c.r_ << ", " << c.i_ << ')'; }
```

Designer durch erfolgreiche Anwendung oft zu übereifrig mit der Anwendung des Operanden überladen:

```
template <typename T>
class Stack {
  public:
    Stack();
    ~Stack();
    void operator +( const T & ); // push
    T &operator *(); // top
    void operator –(); // pop
    operator bool() const; // not empty?
    // . . .
};
// . . .
Stack<int> s;
s + 12;
s + 13;
if( s ) {
  int a = *s;
  -s;
  // . . .
```

Clever ?

NEIN !

saubere Implementierung eines Stacks:
allgemein erwartete nicht-Operator – Namen für die Stackoperationen

Fazit:

Die Bedeutung eines überladenen Operators muss allgemein verständlich sein.
Auch wenn die Bedeutung 75% der Benutzer verständlich ist, ist eine Missinterpretation unakzeptabel.
→ mehr Probleme als Vorteile durch überladen der Operatoren