Einführung in Objekt-Orientiertes-Programmieren

von Borg Enders

Benötigte Vorkenntnisse:

- grundlegende Kenntnisse des vom Leser verwendeten Betriebssystems
- grundlegende Kenntnisse des vom Leser verwendeten Compilers
- grundlegende Kenntnisse der vom Leser verwendeten Programmiersprache (bevorzugt C++ oder Java)
- elementare Begriffe von Programmierkonzepten:
  - Parameterlisten von Funktionen
  - globale und lokale Programmkonstrukte
  - Namensraum
  - dynamische Speicherverwaltung
  - Kapselung
- grundlegende Kenntnisse der englischen Sprache

Die hier verwendete Programmiersprache ist C++. Aber es ist auch jede andere objekt-orientierte Programmiersprache geeignet, um die Beispiele nach zu vollziehen. Der einzige Nachteil ist, daß es hier nur für C++ Musterlösungen gibt.
Der hier verwendete Compiler ist der gcc 2.95.2 für Windows.

Inhaltsverzeichnis

1 Begriffsbildung

Was ist ein Objekt?
Ein Objekt ist ein in sich abgeschlossenes Programmkonstrukt bestehend aus betimmten Eigenschaften und Methoden, das wie ein selbstdeklarieter Datentyp zur Deklaration von Variablen verwendet werden kann.

Was ist ein abgeschlossenes Programmkonstrukt?
Ein abgeschlossenes Programmkonstrukt ist ein in sich geschlossener Namensraum mit sowohl global als auch nur lokal verfügbaren Eigenschaften und Methoden.

Was ist eine Eigenschaft?
Eine Eigenschaft ist ein Wert, der ein Teil des aktuellen Zustandes einer Instanz eines Objektes ist.

Was ist der Zustand einer Instanz eines Objektes?
Ein Zustand sind alle Eigenschaften und damit alle Werte einer Instanz eines Objektes zu einem bestimmten wohldefinierten Zeitpunkt der Programmausführung.

Was ist eine Instanz eines Objektes?
Eine Instanz eines Objektes ist eine Variable die anstelle eines Datentyps ein Objekt als Typ hat und die durch einen Konstruktor initialisiert und durch einen Destruktor deinitialisert werden muß.

Was ist eine Methode?
Eine Methode ist eine Möglichkeit Eigenschaften einer Instanz eines Objektes zu verändern oder mit einer Instanz eines Objektes zu kommunizieren.

Was ist ein Konstruktor?
Ein Konstruktor ist eine spezielle Methode eines Objektes, die für die Initialiserung aller Eigenschaften einer Instanz eines Objektes verantwortlich ist. Insbesondere sorgt ein Konstruktor für ausreichend Speicherplatz für alle beim Initialiseren einer Instanz benötigten dynamisch erzeugten Eigenschaften. Viele Compiler rufen für jede Instanz implizit einen Standardkonstruktor auf, da dies aber nicht für alle Compiler gilt, ist es immer sinnvoll explizit für jede Instanz auch einen Konstruktor aufzurufen.

Was ist ein Destruktor?

Ein Destruktor ist wie der Konstruktor eine spezielle Methode eines Objektes, die dann aufzurufen ist, wenn eine Instanz eines Objektes nicht mehr benötigt wird. Deswegen obliegt es auch dem Destruktor alle von der Instanz dynamisch allozierten Speicherbereiche wieder freizugeben. In einigen Programmiersprachen wird der Aufruf des Destruktors einer Instanz implizit durch den Compiler erledigt. Dies ist zum Beispiel in Java und C++ der Fall.

Was bedeutet objekt-orientiertes?
Dieses bedeutet, daß ein Programm modular aus einzelnen Objekten aufgebaut wird. Aber nur in einer sinnvollen Granularität. Eine zu feine Granularität wäre zum Beispiel ein eigenes Objekt für jede benötigte Variable.

Was bedeutet programmieren?
Programmieren bedeutet ein gegebenes Problem rein formal in Teilprobleme aufzubrechen und diese Teilprobleme als einzelne Probleme zu lösen. Im Fall der Objekt-Orientierten-Programmierung werden hierzu Teilprobleme auf eine geeignete Objekthierarchie abgebildet.

Was ist eine Objekthierarchie?
Eine Objekthierarchie ist eine geschickte Abbildung der Umwelt des Programmes auf Objekte beginnend bei abstrakten Objekten bishin zu konkreten Instanzen.

Was ist ein abstraktes Objekt?
Ein abstraktes Objekt ist die grundlegende Definition für eine Teilmenge der Objekthierarchie, ohne selber mehr als eine elementare Funktionalität zur Verfügung zu stellen.

2 Grundlagen der Programmierung

Der hier erklärte Syntax ist der von C++. In anderen Programmiersprachen gibt es geringfügig andere syntaktische Konstrukte.

- Objekt Deklaration:

```
class <name>
{
 public:
  name();  // standard constructor
```

```
  ~name(); // destructor
  <methods declarations>
  <property declarations>
};
```

- Eigenschaften Deklaration:

```
  <type> <name>;
```

- Methoden Deklaration:

```
  <return_type> <name>(<param_list>);
```

- Methoden Definition:

```
<return_type> <class_name>::<method_name>(<param_list of method>)
{
 \\ ...
}
```

- Zugriff auf Methoden und Eigenschaften einer Instanz:

```
<instance_name>.<name of property>
<instance_name>.<name of method>(<param_list>)
```

- Instanzierung:

```
class Cexample
{
   \\ ...
};
\\ ...
  Cexample instance1=Cexample();
  \\or
  Cexapmple instance2();
\\ ...
```

Zu bemerken wäre hier noch, daß ein Objekt neben seinem Standardkonstruktor noch weitere Konstruktoren mit unterschiedlichen Parameterlisten haben kann. Aber jedes Objekt sollte einen Standardkonstruktor haben.
Desweiteren finden Deklarationen in der Regel in C++ immer in `header`-Dateien statt und Spezifikationen in `cpp`-Dateien.

2.1 Übungsaufgabe

Zu Implementieren sind die folgenden Objekte mit der beschriebenen Funktionalität:
Die Datei `firstgraphics.h`:

```
class Cpoint
{
  public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing to do for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

   int x_coordinate;
   int y_coordinate;
};

class Cline
{
  public:
   /// creates line (x1,y1)---(x2,y2)
   Cline(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Cline();

    /// writes to cout: (x1,y1) --- (x2,y2)
   void draw();

   Cpoint start_point;
   Cpoint end_point;
};

class Clinelist
{
  public:
  /// creates list with n-entries
  Clinelist(int n=1);
```

```
  /// gives the allocated memory free
  ~Clinelist();

  /// adds one line to list if there is still an empty place
  /// returns true on success, false otherwise
  bool add(Cline line);

  /// writes to cout: ... all lines in list each followed by "\n"...
  void draw();

  Cline *list;   /// list of lines
  int size;      /// size of list
  int position;  /// position for adding of next line

};
```

zu der die Datei `firstgraphics.cpp` zu implementieren ist.

Das Hauptprogramm `main.cpp`:

```
#include "firstgraphics.h"

int main()
{
  Clinelist list1=Clinelist(3);
  Cline elem1=Cline(50,50,100,100);
  Cline elem2=Cline(100,100,100,50);
  Cline elem3=Cline(100,50,50,50);
  list1.add(elem1);
  list1.add(elem2);
  list1.add(elem3);
  list1.draw();

  return 0;
}
```

Welches folgende Bildschirmausgabe erzeugen soll (oder zumindestens etwas sehr ähnliches):

```
 (50,50) --- (100,100)
 (100,100) --- (100,50)
 (100,50) --- (50,100)
```

2.2 Musterlösung

```
#include <fstream.h>
#include <iostream.h>

#include "firstgraphics.h"

/// creates point (x,y)
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}
```

In C++ erfolgen alle Ausgaben über streams, wobei der Standardausgabestream `cout` heißt. Auszugebende Zeichenketten werden hierbei mit dem speziellen Operator `<<` konkateniert. Insbesondere ist dieser Operator auch für alle Basisdatentypen definiert.

`Cline` macht hier sinnvollerweise von den bereits vorhanden Methoden des Objektes `Cpoint` gebrauch, anstelle für jeden Punkt die Eigenschaften zu betrachten.

```
/// creates line (x1,y1)---(x2,y2)
Cline::Cline(int x1, int y1, int x2, int y2)
{
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}
```

```
/// nothing todo for this class
Cline::~Cline()
{
}

/// writes to cout: (x1,y1) --- (x2,y2)
void Cline::draw()
{
  start_point.draw();
  cout<<" --- ";
  end_point.draw();
}
```

In C++ werden die Routinen `new` und `delete` zur dynamischen Speicherverwaltung verwendet. `new` legt dabei Speicherplatz für die Anzahl der angeforderten Objekte an und `delete` gibt diesen wieder frei. Wobei die `[]` hinter `delete` angeben, daß diese Eigenschaft ein Feld ist.

```
/// creates list with n-entries
Clinelist::Clinelist(int n)
{
  size=n;
  position=0;
  list = new Cline[size];
}

/// gives the allocated memory free
Clinelist::~Clinelist()
{
  delete[] list;
}

/// adds one line to list if there is still an empty place
/// returns true on success, false otherwise
bool Clinelist::add(Cline line)
{
  if (position<size)
  {
    list[position]=line;
    position++;
    return true;
```

```
  } else
  {
    return false;
  }
}

/// writes to cout: ... all lines in list each followed by "\n"...
void Clinelist::draw()
{
  for( int i=0; i<position;i++)
  {
    list[i].draw();
    cout<<"\n";
  }
}
```

3 Vererbung

Eine Besonderheit von Objekten ist, daß ein Objekt von einem oder mehreren anderen Objekten abgeleitet werden kann. Die Objekte von denen abgeleitet werden, heißen Vorfahren und die abgeleiteten Kinder.
Bei diesem Verfahren erben die Kinder alle Methoden und Eigenschaften ihrer Vorfahren. Dies bedeutet, daß ein Kind auf alle Eigenschaften seiner Vorfahren zugreifen kann und insbesondere für das Kind alle Methoden der Vorfahren aufgerufen werden können.
Desweiteren haben Kinder die Möglichkeit Methoden und Eigenschaften ihrer Vorfahren neu zu definieren. Diese neu definierten Eigenschaften und Methoden überdecken dann implizit die Definition in den Vorfahrenklassen. Dieses Verfahren heißt auch Überladung.
Eine Besonderheit bei der Überladung von Konstruktoren und Destruktoren ist allerdings zu beachten. In diesen speziellen Methoden sollten immer die entsprechenden Methoden der Vorfahren aufgerufen werden.
In C++ ist hier zu beachten, daß die Aufrufe des Destruktors für alle Objekte automatisch erfolgt und der Aufruf des Konstruktors des Vorfahrs als Wertvorbelegung des Objektes aufgerufen werden muß und nicht im eigentlichen Rumpf des Konstruktors. Wenn ein Aufruf eines Konstruktors des Vorfahren fehlt wird implizit der Standardkonstruktor des Vorfahren aufgerufen.
Ein weiterer Vorteil der Vererebung ist, daß jeder Instanz eines Vorfahren eine Instanz eines Kindes zugewiesen werden kann. In diesem Fall sollte dann die Instanz des Vorfahren die Methoden des Kindes verwenden. Dieses Verwenden der Methode des Kindes heißt späte Bindung, da erst zur Laufzeit

entschieden werden kann, welche Methode aufzurufen ist.
Leider muß hierzu in C++ einem Zeiger auf eine Instanz des Vorfahren die Adresse eines Kindes zugewiesen werden. Dies ist eine der wenigen Stellen, wo C++ das objekt-orientierte Konzept nicht implizit unterstützt.
Mittels solcher Zuweisungen kann es vorallem oft sinnvoll sein, ein abstraktes Objekt als Basis eines Zweiges der Objekthierarchie zu definieren. Die Aufgabe dieses abstrakten Objektes ist es dann, von allen Kindern eine Basisfunktionalität zu fordern, ohne diese selber in der Regel näher zu definieren. Insbesondere können mittels dem Urobjekt das jede Programmiersprache zur Verfügung stellt ohne Probleme die unterscheidlichsten Instanzen von Objekten in Feldern verwaltet werden.

4 Programmierung von Vererbung

- Vererbung:

```
class <child>: public <ancestor_1>, ..., public <ancestor_n>
{
  // ...
};
```

- Späte Bindung:

```
class <ancestor>
{
  virtual <return_type> <method_overloadind_name>(<param_list>);

  virtual ~<ancestor>;
};
class <child>: public <ancestor>
{
  <return_type> <method_overloading_name>(<param_list>);
};
```

  Sobald eine einzige Methode `virtual` deklariert wurde, muß auch der Standarddestruktor `virtual` deklariert werden. Dies gewährleistet, daß mittels später Bindung zur Laufzeit immer der korrekte Destruktor aufgerufen wird.

- Zugriff auf einen Zeiger auf eine Instanz eines Objektes:

```
class Cexample
{
  //...
};
// ...
 Cexample* example;
// ...
  example=new Cexample();
//...
  example-><method_name>(<param_list>);
//...
```

- Wertvorbelegung von Eigenschaften eines Objektes mittels eines Konstruktors

```
Cexample::Cexample(<param_list>):
  <property1>(<preset_value>),
  <property2>(<preset_value>)
{
  //...
}
```

4.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `secondgraphics.h`:

```
class Cgraphicobject
{
  public:
    virtual ~Cgraphicobject();
    virtual void draw();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();
```

```
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();

   Cpoint start_point;
   Cpoint end_point;
};

class Csquare: public Crectangle
{
  public:
   /// creates square
   /// S(x1,y1)-
   /// | (x2,y2)
   Csquare(int x1=0, int y1=0, int x2=0, int y2=0);


  /// writes to cout:
  /// S(x1,y1)-
  /// | (x2,y2)
  virtual void draw();
};

class Cpicture {
 public:
  /// creates list with n-entries
```

```
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
  /// returns true on success, false otherwise
  bool add(Cgraphicobject* graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

  Cgraphicobject** list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line

};
```

Hierzu ist die Datei `secondgraphics.cpp` zu implementieren. Aufgrund der Ähnlichkeiten zur letzten Übungsaufgabe ist es durchaus sinnvoll, die dort implementierten Objekte wiederzuverwenden.

Das Hauptprogramm `main.cpp`:

```
#include "secondgraphics.h"

int main() {
  Cpicture picture(2);
  Crectangle rect(50,50,150,100);
  Csquare square(200,200,300,300);
  picture.add(&rect);
  picture.add(&square);
  picture.draw();

  return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
(50,50)-
| (150,100)
```

```
S(200,200)-
| (300,300)
```

4.2 Musterlösung

```
#include <fstream.h>
#include <iostream.h>

#include "secondgraphics.h"

Cgraphicobject::~Cgraphicobject()
{
}

void Cgraphicobject::draw()
{
  cout<<"Cgraphicobject::draw()\n";
}

/// creates point (x,y)
Cpoint::Cpoint(int x, int y):
  x_coordinate(x),
  y_coordinate(y)
{
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(int x1, int y1, int x2, int y2)
{
```

```
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::draw()
{
  start_point.draw();
  cout<<"-\n";
  cout<<"| ";
  end_point.draw();
  cout<<"\n";
}
```

Hier wird korrekterweise zuerst der Konstruktor von `Crectangle` aufgerufen. Bei Destruktoren wird der vererbte Destruktor immer als letztes aufgerufen.
Beim Konstruktor führt dies dazu, daß die Aufrufe der vererbten Konstruktoren alles initialisieren, was nicht speziell für dieses Objekt ist, und erst dann werden die für dieses Objekt speziellen Eigenschaften initialisert oder bestimmte Werte vererbten Eigenschaften zugewiesen. Beim Destruktor erfolgt die Aufrufsreihenfolge genau umgekehrt, da ein Aufruf des Destruktors des Vorfahren zu einem Aufruf des Destruktors des Urobjektes führt, der die Instanz dieses Objektes frei gibt, wodurch es keine Möglichkeit mehr gibt auf die speziellen Eigenschaften der Instanz dieses Objektes zuzugreifen.

```
/// creates square
/// (x1,y1)-
/// | (x2,y2)
Csquare::Csquare(int x1, int y1, int x2, int y2):
  Crectangle(x1,y1,x2,y2)
{
}

/// writes to cout:
```

```
/// S(x1,y1)-
/// | (x2,y2)
void Csquare::draw()
{
  cout<<"S";
  start_point.draw();
  cout<<"-\n";
  cout<<"| ";
  end_point.draw();
  cout<<"\n";
}

/// creates list with n-entries
Cpicture::Cpicture(int n)
{
  size=n;
  position=0;
  list = new Cgraphicobject*[size];
}

/// gives the allocated memory free
Cpicture::~Cpicture()
{
  delete[] list;
}

/// adds one graphicobject to list if there is still an empty place
/// returns true on success, false otherwise
bool Cpicture::add(Cgraphicobject* graphic)
{
  if (position<size)
  {
    list[position]=graphic;
    position++;
    return true;
  } else
  {
    return false;
  }
}

/// writes to cout: ... all graphicobjects in list each followed by "\n"...
void Cpicture::draw()
```

```
{
  for( int i=0; i<position;i++)
  {
    list[i]->draw();
    cout<<"\n";
  }
}
```

Eine kleine Zusatzaufgabe für Interessierte wäre, anstelle des Feldes von Zeigern in `Cpicture` eine Liste von Objekten zuverwenden, d.h. `Cgraphicobject**` durch `Cgraphicobject*` zuersetzen und den restlichen Code entsprechenden anzupassen.

5 Gültigkeitsbereiche von Eigenschaften und Methoden

Um die Modularität von Objekten zu verstärken können explizit für Eigenschaften und Methoden Gültigkeitsbereiche definiert werden. Die beiden wichtigsten sind der bisher implizit verwendete Gültigkeitsbereich öffentlich (`public`) und der Gültigkeitsbereich privat (`private`).
Der Gültigkeitsbereich öffentlich erlaubt den Zugriff von außerhalb des Objektes auf die so deklarierten Methoden und Eigenschaften. Im Gegensatz hierzu dürfen auf Methoden und Eigenschaften des Gültigkeitsbereiches privat nur Methoden des Objektes selber und all seiner Kinder zugreifen, in C++ dürfen dies allerdings die Kinder nicht.
Insbesondere ist es deshalb sinnvoll, nahezu alle Eigenschaften eines Objektes als privat zu deklarieren und alle Zugriffe auf diese Eigenschaften mittels öffentlicher Methoden zu kapseln. Dies gewährleistet, daß eine Instanz eines Objektes die Korrektheit all seiner Eigenschaften über die Aufrufe der kapselnden Methoden garantieren kann und ermöglicht so eine Fehlerbehandlung, sollten diesen Eigenschaften unzulässige Werte zugewiesen werden.
In einigen Programmiersprachen gibt es noch einen dritten Gültigkeitsbereich. Dieser heißt geschützt (`protected`). Auf Eigenschaften und Methoden eines Objektes mit diesem Gültigkeitsbereich dürfen nur Methoden desselben oder befreundeter Objekte zugreifen. Hierbei müssen alle befreundeten Objekte eines bestimmten Objektes explizit deklariert werden.
In C++ dürfen auch alle Kinder auf geschützt deklarierte Eigenschaften und Methoden zugreifen.

6 Programmierung mit Gültigkeitsbereichen

Wenn explizit kein Gültigekitsbereich angegeben wird, verwendet C++ implizit den Gültigkeitsbereich privat. In C++ kann auch beim Ableiten eines Kindes von einem Vorfahren noch ein Gültigkeitsbereich angegeben werden. Dieser beeinflusst dann die Gültigkeitsbereiche der Eigenschaften und Methoden des Vorfahren beim Kind.

```
<child_class>: <public/protected/private> <ancestor_class>
{
  friend class <friend_class>;

  public:
    /// all methods and properties, which should be public

  protected:
    /// all methods and properties, which should be protected

  private:
    /// all methods and properties, which should be private

};
```

Gültigkeitsbereiche in Kindern abhängig von der Ableitungsart:

Vorfahr	Art der Ableitung		
	private	protected	pubic
public	private	protected	public
protected	private	protected	protected
private	-	-	-

6.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `thirdgraphics.h`:

```
class Cgraphicobject
{
  public:
```

```
    virtual ~Cgraphicobject();
    virtual void draw();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();

  protected:
   Cpoint start_point;
   Cpoint end_point;
};

class Csquare: public Crectangle
```

```
{
  public:
   /// creates square
   /// S(x1,y1)-
   /// | (x2,y2)
   /// and controls that x2-x1=y2-y1 is
   /// and if not decreases the larger side
   /// to the size of the smaller side
   Csquare(int x1=0, int y1=0, int x2=0, int y2=0);

  /// writes to cout:
  /// S(x1,y1)-
  /// | (x2,y2)
  virtual void draw();
};

class Cpicture {
 public:
  /// creates list with n-entries
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
  /// returns true on success, false otherwise
  bool add(Cgraphicobject* graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

 private:
  Cgraphicobject** list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line
};
```

Hierzu ist die Datei `thirdgraphics.cpp` zu implementieren. Aufgrund der Ähnlichkeiten zur letzten Übungsaufgabe ist es durchaus sinnvoll, die dort implementierten Objekte wiederzuverwenden.

Das Hauptprogramm `main.cpp`:

```
#include "thirdgraphics.h"

int main() {
  Cpicture picture=Cpicture(2);
  Crectangle rect=Crectangle(50,50,150,100);
  Csquare square=Csquare(200,200,300,300);
  picture.add(&rect);
  picture.add(&square);
  picture.draw();

  return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
WARNING
S(200,200)-
| (250,300)
is not a square!
correcting endpoint to (250,250)


(50,50)-
| (150,100)

S(200,200)-
| (250,250)
```

6.2 Musterlösung

Hier nur die Änderungen gegenüber der Musterlösung `secondgraphics.cpp`.

```
// ...
#include "thirdgraphics.h"
// ...

/// creates square
/// (x1,y1)-
/// | (x2,y2)
/// and controls that x2-x1=y2-y1 is
/// and if not decreases the larger side
```

```
/// to the size of the smaller side
Csquare::Csquare(int x1, int y1, int x2, int y2):
  Crectangle(x1,y1,x2,y2)
{
    int dx=x2-x1;
    int dy=y2-y1;

    if (dx!=dy)
    {
      cout<<"WARNING"<<"\n";
      draw();
      cout<<"is not a square!"<<"\n";

      if (dx<dy)
      {
        y2=y1+dx;
      } else
      {
        x2=x1+dy;
      }
      end_point=Cpoint(x2,y2);

      cout<<"correcting endpoint to ";
      end_point.draw();
      cout<<"\n\n\n";
    }
}
// ...
```

Eine kleine Zusatzaufgabe für Interessierte: deklarieren sie die Eigenschaften `start_point` und `end_point` des Objektes `Crectangle` als `private` und passen sie den restlichen Code entsprechend an.

7 Design Patterns

Design Patterns sind, wie der Name schon sagt, Muster für oft wiederkehrende Probleme der Programmierung.
Hier sollen insgesamt vier nützliche Muster vorgestellt werden. Aber es gibt noch eine ganze Reihe weiterer sinnvoller Design Patterns die man im Buch "Design Patterns: Elements Of Reusable Object-Oriented Software" (Addison-Wesley) von Erich Gamma nachlesen kann.
Die hier vorgestellten Muster sind:

Composite Pattern: hierbei wird ein Objekt aus einer Anzahl ähnlicher Objekte zusammengesetzt, wie dies zum Beispiel oft bei Datenbanken der Fall ist.

Template Method Pattern: hierbei wird innerhalb eines abstrakten Objektes ein bestimmter Algorithmus implementiert und alle hierzu benötigten Methoden abstrakt definiert. Somit können alle Kinder dieses Objektes den Algorithmus nutzen. Ein gängiges Beispiel hierfür ist zum Beispiel ein Sortieralgorithmus.

Factory Pattern: Hierbei kann eins von mehreren angeforderten Objekten, das Ergebnis einer Methode sein. Die aufrufenden Objekte entscheiden dabei, welches Objekt sie benötigen. Dieses kann zum Beispiel verwendet werden, um ähnliche Objekte mit gleichen Eigenschaften aber unterschiedlichen Methoden zu definieren.

Chain Of Responsibility: Hierbei wird eine Kette von Objekten gebildet, die für ein bestimmtes Ereignis bei der Programmausführung zuständig sind. Dies kann zum Beispiel verwendet werden, um bei einer graphischen Oberfläche auf Eingaben der Maus zu reagieren.

8 Composite Pattern

Das Composite Pattern repräsentiert eine rekursive Teil-Ganzes-Beziehung zwischen Objekten und ermöglicht hierbei eine einheitliche Behandlung von atomaren Bestandteilen des Objektes.
Hierfür fordert eine abstrakte Basisklasse folgende Funktionalität:

- Definition aller benötigten Methoden der abgeleiteten Objekte
- Hinzufügen einer weiteren Komponente
- Entfernen einer Komponente
- Zugriff auf eine beliebige hinzugefügte Komponente

Der öffentliche Teil der Objektdefinition sieht dann wie folgt aus:

```
class CComponent
{
  public:
   // definition of needed methods
   virtual void example_method();

   virtual int add(CComponent* c);

   virtual int remove(CComponet* c);

   virtual CComponent* get_child(int index);

   virtual ~CComponent();
};
```

Von diesem Objekt lassen sich dann weitere zusammengesetze Objekte oder atomare Objekte ableiten, deren öffentlicher Teil wie folgt aussieht:

```
class CComposite: public CComponent
{
  public:
   // definition of needed methods
   void example_method();

   int add(CComponent* c);

   int remove(CComponet* c);

   CComponent* get_child(int index);
};

class Catom: public CComponent
{
  public:
   // definition of needed methods
   void example_method();
};
```

8.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `fourthgraphics.h`:

```
class Cgraphicobject
```

```
{
  public:
    virtual void draw();

    virtual int add(Cgraphicobject g);

    virtual Cgraphicobject* get_child(int index);

    virtual ~Cgraphicobject();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();
```

```
 protected:
   Cpoint start_point;
   Cpoint end_point;
};

class Cpicture : public Cgraphicobject
{
 public:
  /// creates list with n-entries
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
  /// returns true on success, false otherwise
  int add(Cgraphicobject graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

  // result the element at position index in list
  Cgraphicobject* get_child(int index);

 private:
  Cgraphicobject* list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line
};
```

Auf `remove` von `Cgraphicobject` wurde hier verzichtet, da hierfür zusätzlich noch ein Gleichheitstest für alle Objekte nötig wäre.
Hierzu ist die Datei `fourthgraphics.cpp` zu implementieren. Aufgrund der Ähnlichkeiten zur letzten Übungsaufgabe ist es durchaus sinnvoll, die dort implementierten Objekte wiederzuverwenden.

Das Hauptprogramm `main.cpp`:

```
#include "fourthgraphics.h"

int main() {
```

```
  Cpicture picture=Cpicture(2);
  Crectangle rect=Crectangle(50,50,150,100);
  Crectangle square=Crectangle(200,200,300,300);
  picture.add(&rect);
  picture.add(&square);
  picture.add(&rect);
  picture.draw();

  return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
Error: list is full

(50,50)-
| (150,100)

(200,200)-
| (300,300)
```

8.2 Musterlösung

Hier nur die Änderungen gegenüber der Musterlösung `thirdgraphics.cpp`.

```
//...
#include "fourthgraphics.h"
//...

int Cgraphicobject::add(Cgraphicobject* g)
{
  cout<<"Cgraphicobject::add";
  return -1;
}

Cgraphicobject* Cgraphicobject::get_child(int index)
{
  cout<<"Cgraphicobject::get_child(int index)";
  return NULL;
}
//...
```

```
/// adds one graphicobject to list if there is still an empty place
/// returns true on success, false otherwise
int Cpicture::add(Cgraphicobject* graphic)
{
  if (position<size)
  {
    list[position]=graphic;
    position++;
    return position-1;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return -1;
  }
}
//...
```

In C++ ist es üblich Fehlermeldung auf dem vordefinierten stream `cerr` auszugeben. Dieser wird genauso gehandhabt wie der Standardausgabestream.

```
// result the element at position index in list
Cgraphicobject* Cpicture::get_child(int index)
{
  if (index<position)
  {
    return list[index];
  } else
  {
    return NULL;
  }
}
```

Eine kleine Aufgabe für Interessierte: ändern sie das Hauptprogramm so ab, daß nicht mehr die Methode `draw` von `picture` aufgerufen wird, sondern jedes Element von `picture` mittels `get_child` abgerufen und dann das `draw` dieses Elements aufgerufen wird.

9 Template Method Pattern

Mittels des Template Method Patterns wird in einem abstrakten Objekt ein nützlicher Algortihmus implementiert. Hierfür definiert das Template Objekt konkret

den Algorithmus und abstrakt die für den Algorithmus benötigten Methoden. Diese Methoden müssen dann in den jeweiligen Objekten definiert werden, die den Algorithmus nutzen möchten.
Der öffentliche Teil eines Template Objekts sieht dann wie folgt aus:

```
class Ctemplate
{
  public:
    // nice algorithm
    void templateMethod();

    // for templateMethod needed methods
    virtual void example_method1();

    virtual void example_method2();

    virtual ~Ctemplate();
};
```

Der öffentliche Teil eines konkreten Objekts sieht dann wie folgt aus:

```
class Cconcrete: public Ctemplate
{
  public:

    // for templateMethod needed methods
    void example_method1();

    void example_method2();
};
```

9.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `fifthgraphics.h`:

```
class Cpoint
{
  friend class Clist;

public:
  /// creates point (x,y)
  Cpoint(int x=0, int y=0);
```

```
   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 protected:
   int x_coordinate;
   int y_coordinate;
};

class Clisttemplate
{
  public:
    /// algorithm sorts points into list
    /// depending on distance to (0,0)
    void insert(Cpoint p);

    /// needed operations for algorithm

    ///adds one point to a pointlist
    virtual void add(Cpoint p);

    /// swaps the points with index from and to
    virtual void swap(int from, int to);

    /// size of list
    virtual int size();

    /// distance of point i to (0,0)
    virtual int distance(int i);

    virtual ~Clisttemplate();
};

class Clist : public Clisttemplate
{
 public:
  /// creates list with size n
  Clist(int n);

  /// gives the allocated memory free
```

```
  ~Clist();


  /// writes to cout: ... all points in list each followed by "\n"...
  void draw();

  ///adds one point to a pointlist if there is still en ampty place
  void add(Cpoint p);

  /// swaps the points with index from and to
  void swap(int from, int to);

  /// size of list
  int size();

   /// distance of point i to (0,0)
  int distance(int i);

 private:
  Cpoint* list;   /// list of points
  int max_size;       /// size of list
  int position;   /// position for next empty place in list
};
```

Das Hauptprogramm `main.cpp`:

```
#include "fifthgraphics.h"

int main() {
   Clist list(5);
   Cpoint p1=Cpoint(10,20);
   list.insert(p1);
   Cpoint p2=Cpoint(5,15);
   list.insert(p2);
   Cpoint p3=Cpoint(50,50);
   list.insert(p3);
   Cpoint p4=Cpoint(10,10);
   list.insert(p4);
   Cpoint p5=Cpoint(12,13);
   list.insert(p5);
   list.draw();
```

```
   return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
(10,10)
(5,15)
(12,13)
(10,20)
(50,50)
```

9.2 Musterlösung

Hier die Musterlösung:

```
#include <fstream.h>
#include <iostream.h>

#include "fifthgraphics.h"

/// creates point (x,y)
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

Clisttemplate::~Clisttemplate()
{
}
```

```
/// algorithm sorts points into list
/// depending on distance to (0,0)
void Clisttemplate::insert(Cpoint p)
{
  add(p);

  for(int i=size()-1;i>0;i--)
  {
    if (distance(i)<distance(i-1))
      swap(i-1,i);
  }
}
```

Dieser Algorithmus ist eine rudimentäre Form des InsertionSort-Algorithmus. Der Vorteil dieses Algorithmus liegt darin, daß er auf sortierten Listen nur linearen Aufwand hat.

```
///adds one point to a pointlist
void Clisttemplate::add(Cpoint p)
{
 cout<<"Clisttemplate::add(Cpoint p)"<<"\n";
}

/// swaps the points with index from and to
void Clisttemplate::swap(int from, int to)
{
  cout<<"Clisttemplate::swap(int from, int to)\n";
}

int Clisttemplate::size()
{
  cout<<"Clisttemplate::size()\n";
  return 0;
}

int Clisttemplate::distance(int i)
{
  cout<<"Clisttemplate::distance(int i)\n";
  return 0;
}
```

```
/// creates list with size n
Clist::Clist(int n)
{
  max_size=n;
  position=0;
  list = new Cpoint[n];
}

/// gives the allocated memory free
Clist::~Clist()
{
  delete[] list;
}

/// adds one point to the end of the list
void Clist::add(Cpoint p)
{
  if (position<max_size)
  {
    list[position]=p;
    position++;
    return;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return;
  }
}

void Clist::swap(int from, int to)
{
  if ((from>=0) && (from<position) &&
      (to>=0)   && (to<position)    )
  {
    Cpoint h=list[from];
    list[from]=list[to];
    list[to]=h;
  } else
  {
    cerr<<"Error: list has size "<<size()<<"\n";
    cerr<<"swap("<<from<<","<<to<<") out of index\n";
  }
}
```

```
/// writes to cout: ... all graphicobjects in list each followed by "\n"...
void Clist::draw()
{
  for( int i=0; i<position;i++)
  {
    list[i].draw();
    cout<<"\n";
  }
}

int Clist::size()
{
  return position;
}

int Clist::distance(int i)
{
  return list[i].x_coordinate*list[i].x_coordinate+
         list[i].y_coordinate*list[i].y_coordinate;
}
```

10 Factory Pattern

Mittels dem Factory Pattern wird ein einheitliches Interface zur Erzeugung von Instanzen definiert, wobei die Instanzen von unterschiedlichen Objekten sein können. Die Aufrufenden Instanzen entscheiden, welche Objektinstanzen dabei zurückgeliefert werden sollen.
Der öffentliche Teil der hierfür benötigten Objekte sieht dann wie folgt aus:

```
class Ccreator
{
  public:
    Cproduct* factoryMethod(<params>);

  //...
};

class Cproduct
{
  //...
};
```

```
class Cproduct1: public Cproduct
{
  //...
};

class Cproduct2: public Cproduct
{
 //...
};
```

Dieses könnte dann zum Beispiel wie folgt aufgerufen werden:

```
//...
  Ccreator factory();
//...
  Cproduct1* example=(Cproduct1*)factory.factoryMethod(<value for
                                                        product1 as result>);
//...
```

Hierbei ist `(Cproduct1*)` ein sogenannter `typecast`. Dieses bedeutet, daß eine Variable eines bestimmten Types so behandelt werden soll, als ob sie den in den Klammern vorkommenden Typ besitzt.

10.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `sixthgraphics.h`:

```
class Cgraphicobject
{
  public:
    virtual void draw();
    virtual ~Cgraphicobject();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates (x,y)
   Cpoint(int x=0,int y=0);

   /// creates point (x,y)
```

```
   Cpoint(const Cpoint& init);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(Cpoint start,Cpoint end);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
    void draw();

 private:
   Cpoint start_point;
   Cpoint end_point;
};

class Ctriangle : public Cgraphicobject
{
 public:
   /// creates triangle
   /// (x1,y1)-(x2,y2)
   /// | (x3,y3)
   Ctriangle(Cpoint a, Cpoint b, Cpoint c);

   /// nothing todo for this class
```

```
   ~Ctriangle();

   /// writes to cout:
   /// (x1,y1)-(x2,y2)
   /// | (x3,y3)
   void draw();

 private:
   Cpoint a_point;
   Cpoint b_point;
   Cpoint c_point;
};

class Clist
{
 public:
  /// creates list with size n
  Clist(int n);

  /// gives the allocated memory free
  ~Clist();

  ///adds one point to a pointlist if there is still en ampty place
  void add(Cpoint p);

  /// creates a Ctriangle for type==0 and a Crectangle for type==1

 Cgraphicobject* factoryMethod(int type);

 private:
  Cpoint* list;   /// list of points
  int max_size;       /// size of list
  int position;   /// position for next empty place in list
};
```

Das Hauptprogramm `main.cpp`:

```
#include "sixthgraphics.h"

int main() {
   Clist list(3);
   Cpoint p1=Cpoint(10,20);
```

```
   list.add(p1);
   Cpoint p2=Cpoint(5,15);
   list.add(p2);
   Cpoint p3=Cpoint(50,50);
   list.add(p3);

   Ctriangle* t=(Ctriangle*)list.factoryMethod(0);
   t->draw();

   Crectangle* r=(Crectangle*)list.factoryMethod(1);
   r->draw();

   return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
(10,20)-(5,15)
| (50,50)
(10,20)-
| (5,15)
```

10.2 Musterlösung

Hier die Musterlösung:

```
#include <fstream.h>
#include <iostream.h>

#include "sixthgraphics.h"

Cgraphicobject::~Cgraphicobject()
{
}

void Cgraphicobject::draw()
{
  cout<<"Cgraphicobject::draw()";
}

///creates (x,y)
Cpoint::Cpoint(int x,int y)
```

```
{
  x_coordinate=x;
  y_coordinate=y;
}

/// creates point (x,y)
Cpoint::Cpoint(const Cpoint& init)
{
  x_coordinate=init.x_coordinate;
  y_coordinate=init.y_coordinate;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(Cpoint start,Cpoint end)
{
  start_point=Cpoint(start);
  end_point=Cpoint(end);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::draw()
{
  start_point.draw();
```

```
  cout<<"-\n";
  cout<<"| ";
  end_point.draw();
  cout<<"\n";
}

 /// creates triangle
 /// (x1,y1)-(x2,y2)
 /// | (x3,y3)
Ctriangle::Ctriangle(Cpoint a,Cpoint b,Cpoint c)
{
  a_point=Cpoint(a);
  b_point=Cpoint(b);
  c_point=Cpoint(c);
}

/// nothing todo for this class
Ctriangle::~Ctriangle()
{
}

/// writes to cout:
/// (x1,y1)-(x2,y2)
/// | (x3,y3)
void Ctriangle::draw()
{
  a_point.draw();
  cout<<"-";
  b_point.draw();
  cout<<"\n";
  cout<<"| ";
  c_point.draw();
  cout<<"\n";
}

/// creates list with size n
Clist::Clist(int n)
{
  max_size=n;
  position=0;
  list = new Cpoint[n];
}
```

```
/// gives the allocated memory free
Clist::~Clist()
{
  delete[] list;
}

/// adds one point to the end of the list
void Clist::add(Cpoint p)
{
  if (position<max_size)
  {
    list[position]=p;
    position++;
    return;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return;
  }
}

Cgraphicobject* Clist::factoryMethod(int type)
{
   if (type==0)
   {
      if (position>=3)
      {
        Ctriangle* t=new Ctriangle(list[0],list[1],list[2]);
        return t;
      } else
      {
        cerr<<"Not enough points to create triangle\n";
        Cpoint p=Cpoint();
        Ctriangle* t=new Ctriangle(p,p,p);
        return t;
      }
   }

   if (type==1)
   {
      if (position>=1)
      {
        Crectangle* r=new Crectangle(list[0],list[1]);
```

```
        return r;
      } else
      {
        cerr<<"Not enough points to create rectangle\n";
        Cpoint p=Cpoint();
        Crectangle* r=new Crectangle(p,p);
        return r;
      }
   }

  return NULL;
}
```

Hier wird der Operator `new` verwendet um explizit Speicherplatz zu reservieren. Damit werden diese Instanzen nicht als lokal betrachtet und nicht nach Verlassen der Methode abgeräumt. Dies sollte explizit erfolgen.
Eine kleine Aufgabe für Interessierte: lassen sie den Operator `new` weg und passen sie die restliche Methode entsprehend an.

11 Chain of Responsibilty

Das Pattern Chain of Responsibility ist motiviert dadurch, daß ein Objekt Ereignisse oder Nachrichten erzeugt, welche von verschieden anderen Instanzen von Objekten bearbeitet werden sollen. Um jetzt zu garantieren, daß jedes Ereignis oder jede Nachricht von der korrekten Instanz bearbeitet wird, wird eine Chain of Responsibility eingerichtet. Durch diese Kette werden dann die Ereignisse oder Nachrichten weitergereicht, und jede Instanz bearbeitet dabei die für sie relevanten Ereignisse oder Nachrichten.
Hierfür ist als erstes eine Objektdefinition zur Behandlung von Ereignissen oder Nachrichten nötig und ein Objekt das ein Ereignis oder eine Nachricht darstellt. Die öffentlichen Teile dieser Objekte sehen dann wie folgt aus:

```
class Cevent
{
  //...
};

class Chandler
{
  public:
    virtual void handle(Cevent e);
```

```
    void setsuccessor(Chandler* h);

    virtual ~Chandler();
};
```

Im privaten von `Chandler` gibt es natürlich die Eigenschaft `Chandler* successor`, über welche die Kette zur Ereignisbehandelung aufgebaut wird.
Hierbei muß bei der Deklaration am Ende, sofern vorhanden, die Methode `handle` der Mittels `setsuccesor` gesetzen Instanz aufgerufen werden, um das Ergeinis gegebenfalls in der Kette weiterzureichen.
Von `Chandler` lassen sich jetzt konkrete Objekte zur Behandlung von Objekten ableiten, deren öffentlicher Teil sieht dann wie folgt aus:

```
class Chandlerspec: public Chandler
{
  public:
    void handle(Cevent e);
};
```

11.1 Übungsaufgabe

Zu implementieren sind die folgenden Objekte, deklariert in der Datei `seventhgraphics.h`:

```
class Cevent
{
   public:
     Cevent(int t=-1);

     ~Cevent();

     // type==0
     bool isDrawEvent();

   private:
     int type;
};

class Chandler
{
  public:
    Chandler();

    virtual ~Chandler();
```

```
    virtual void handle(Cevent e);

    void setsuccessor(Chandler* h);

  private:
    Chandler* succ;
};

class Cpoint
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Chandler
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
    /// if e is drawevent
   void handle(Cevent e);
```

```
 protected:
   Cpoint start_point;
   Cpoint end_point;
};

class Cpicture : public Chandler
{
 public:
  Cpicture();

  /// nothing to do
  ~Cpicture();

  void handle(Cevent e);

  /// creates draw event
  void draw();
};
```

Das Hauptprogramm `main.cpp`:

```
#include "seventhgraphics.h"

int main() {
  Cpicture picture=Cpicture();
  Crectangle rect=Crectangle(50,50,150,100);
  picture.setsuccessor(&rect);
  Crectangle rect2=Crectangle(200,200,250,300);
  rect.setsuccessor(&rect2);
  picture.draw();

  return 0;
}
```

Dieses soll die folgende Bildschirmausgabe (oder eine sehr ähnliche) erzeugen:

```
Drawevent created
(50,50)-
| (150,100)
(200,200)-
| (250,300)
Event finished
```

11.2 Musterlösung

Hier die Musterlösung:

```
#include <fstream.h>
#include <iostream.h>

#include "seventhgraphics.h"

Cevent::Cevent(int t=-1)
{
  type=t;
}

Cevent::~Cevent()
{
}

bool Cevent::isDrawEvent()
{
  return type==0;
}

Chandler::Chandler()
{
  succ=NULL;
}

Chandler::~Chandler()
{
  delete succ;
}

void Chandler::handle(Cevent e)
{
  if (succ!=NULL)
  {
    succ->handle(e);
  } else
  {
    cout<<"Event finished\n";
  }
}
```

```
void Chandler::setsuccessor(Chandler* h)
{
  succ=h;
}

/// creates point (x,y)
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(int x1, int y1, int x2, int y2)
{
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::handle(Cevent e)
{
  if (e.isDrawEvent())
```

```
  {
     start_point.draw();
     cout<<"-\n";
     cout<<"| ";
     end_point.draw();
     cout<<"\n";
  }

  Chandler::handle(e);
}

Cpicture::Cpicture()
{
}

/// nothing to do
Cpicture::~Cpicture()
{
}

void Cpicture::handle(Cevent e)
{
  Chandler::handle(e);
}

/// creates drawEvent
void Cpicture::draw()
{
  Cevent e=Cevent(0);
  cout<<"Drawevent created\n";
  handle(e);
}
```