# 5 Objektorientierte Programmierung

= Objektbasierte Programmierung + Vererbung

## 5.1 Vererbung

```
class A : public B { ... }
```

Jedes „A“ ist ein „B“ (und das ist auch nach aussen sichtbar)



### 5.1.1 Beispiel: Sender-Klassen

```
class AnonymousSender {
public:
  point position;
  double range;
  AnonymousSender(point p, double r) :
     position(p), range(r) { }
  ...
}

class Sender : public AnonymousSender{
public:
  string name;
  Sender(string n, point p, double r) :
    AnonymousSender(p, r), name(n) { }
  ...
}
```



```
class AmateurSender : public Sender {
public:
  AmateurSender(string n, point p, double r) :
    Sender(n, p, r) { };
  AmateurSender(string n,Locator l,double r):
    Sender(n, l.position(), r) { };
  Locator getLocator() {
    return Locator(position);
  };
  void setLocator(Locator l) {
    position = l.position();
  }
}
```



## 5.2 Implementierung

```
class A { int a1: double a2; }
class B : public A { char b; }
```

Eine Instanz ist aufgebaut wie

```
struct AB { int a1; double a2; char b }
```

Wichtig: Jedes Member hat in jeder Klasse, in der es auftritt den gleichen Offset.

## 5.3 Virtuelle Funktionen

```
class AnonymousSender {
...
  virtual void print() {
    cout << position << " " << range << endl;
  };
}

class NumberedSender : public AnonymousSender {
public:
  int id;
...
  void print() {
    cout << id << " ";
    AnonymousSender::print();
  }
}
```



```
class NamedSender : public AnonymousSender {
public:
  string name;
...
  void print() {
    cout << name << " ";
    AnonymousSender::print();
  }
```



### Virtuelle Funktionen versus Overloading

```
Sender *s[3] = {NamedSender(...),
                NumberedSender(...),
                Sender(...)}
...
for (i = 0;  i < 3;  i++) {
  s[i]->print();
}
```



### Implementierung

Sei `K` Klasse mit $k > 0$ virtuellen Funktionen

- Der Compiler legt eine Tabelle $T$ der Größe $k$ mit Zeigern auf die in `K` zu verwendenden Funktionen an. (Einmal für jede Klasse.)
- Jede Instanz von `K` bekommt $\&T$ als zusätzliches Member
- Aufruf der $i$-ten Funktion wird umgesetzt in: `*(this->t[i])(...)`
- Kann man in C von Hand machen!
- Aufruf relativ teuer
- Inlining fast unmöglich
- Unterlegene Alternative: Typfeld und `switch`

## 5.4 Abstrakte Klassen

(C++-speak: *virtuelle Basisklassen*)

```
class AbstractSender {
  ...
  virtual void print() = 0;
}
```

Jede Klasse mit mindestens einer *rein virtuellen* Funktion ist abstrakt.

- Alle nichtabstrakten Unterklassen müssen die rein virtuellen Funktionen implementieren
- Es gibt keine Instanzen von abstrakten Klassen



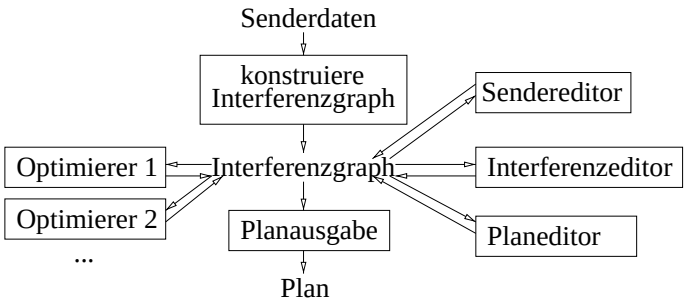
## 5.5 Wozu Vererbung

- Code-Wiederverwendung
- Erweitern von Basisklassen. Beispiel: Behandlung von Locators
- Spezialisierung
- Memberfunktionen redefinieren
- Polymorphie zur Vermeidung von `switch` (nur hier virtuelle Klassen nötig)
- Tradoff: Buy or inherit? Mittelding: `private` Vererbung



# 6 Grobentwurf für Frequenzzuordnung

## 6.1 Datenfluss






## 6.2 Ablauf

1. $\geq 6$ unabhängige Programme. (Je nach Anzahl Optimierer). Aber zunehmende Harmonisierung:
   - Gemeinsame Klassen für Sender, Interferenzgraph
   - Ein Code für Datei Ein/Ausgabe
   - Ein Mechanismus für grafische Benutzeroberflächen
2. Integration aller Komponenten unter gemeinsamer Benutzeroberfläche – Integrationsgruppe beginnt sofort
3. Erweiterung der Funktionalität

## 6.3 Teilprojekt Geometrie

Mindestumfang des ersten Schritts

- Datei mit Senderbeschreibungen lesen
- Grafische Darstellung d. Senderninformation
- Editieren von Senderinformation bei anklicken
- Interferenzgraph erzeugen und abspeichern
- Umwandlung von Längen- und Breitengraden

Inkrementelle Erweiterungen

- Koexistenz mit Editoren für Interferenzgraph und Planeditor
- Effiziente Algorithmen für Berechnung der Kanten des Interferenzgraphen (LEDA)
- Locatorsystem für Amateurfunker
- Darstellung der Senderinformation bei Berühren eines Senders?
- Mitwirkung bei einem kombinierten Geometrie/Grapheditor?



## 6.4 Interferenzeditor

(z.B. mit LEDA GraphWin)

- Grafische Darstellung des Interferenzgraphen (z.B. mit LEDA GraphWin)
- Einfügen von Kanten
- Löschen von Kanten

Inkrementelle Erweiterungen

- Koexistenz mit Editoren für Interferenzgraph und Planeditor
- Mitwirkung bei einem kombinierten Geometrie/Grapheditor?



## 6.5 Planeditor/Ausgabe

Mindestumfang des ersten Schritts

- Darstellung/Ausgabe einer Liste der Kanalzuordnungen
- Grafische Dartstellung der Kanalzuordnung
- Manuelle Eingabe von Kanälen für einzelne Sender (anklicken)
- Fixieren/Defixieren von Kanälen für einzelne Sender,

Inkrementelle Erweiterungen

- Unterstützung bei der Eingabe von Kanälen welche sind erlaubt, ...
- Koexistenz mit Editoren für Interferenzgraph und Sender
- Mitwirkung bei einem kombinierten Geometrie/Grapheditor?
- Export HTML/LaTeX/Postscript?



## 6.6 Optimierer

Mindestumfang des ersten Schritts

- In irgendeiner Reihenfolge suche für jeden Sender den ersten passenden Kanal
- Experimente mit verschiedenen Reihenfolgen: grösster Grad zuerst, zufällig, ...

„Fortgeschrittene“ Algorithmen

- Hill climbing
- Simulated annealing?
- Taboo search?
- Genetische Algorithmen?
- ...?

## 6.7 Integrationsgruppe

Schlüsselaufgaben

- Einheitlicher Repräsentationen für zentrale Datenstrukturen durchsetzen
- Wiederverwendung durchsetzen (Datei-Ein/Ausgabe), Fensteroperationen, ...
- Entwicklung einer zentralen Steuerung aller Komponenten in einem Programm
- Integration der Komponenten

Weitere Ziele

- Testkonzept
- Dokumentationsstandards
- Quelltextstandards
- Versionskontrolle
- Koordination bei der Entwicklung von kombinierten Plan/Geometrie/Interferenzeditoren?



## 6.8 Interessenprofile

**Geometrie:** LEDA, grafische Benutzeroberflächen, geometrisches Vorstellungsvermögen

**Integration/Interferenzeditor:** Interesse an Softwaretechnik, Kommunikationsfähigkeit, Sorgfalt, LEDA GraphWin

**Planeditor** Grafische Benutzeroberflächen, ggf. LaTeX, HTML, Unix

**Optimierer** Abstraktes Denken, ggf. Mathematik, KI



## 6.9 Klasse Sender

Entwurfsentscheidung: Ein Sender ein Kanal.

Aber: Meherere Sender auf gleicher Position erlaubt.

```
class Sender {
public:
  string name; // same rules as C++ identifiers
  point position;
  double range; // >= 0.0
  int channel; // -1 == unknown
  bool fixed; // channel changable?
  ...
}
```

LEDA Paramtertyp mit hash-Funktion (Handbuch Abschnitt 1.3, 1.3.1)



**Frequenzbereichsinformation**

```
class Sender {
  ...
  static double base;
  static double step;
  static int numChannels;
  ...
}
```

**Koordinateninfo**

```
class Sender {
  ...
  static double latitudeFactor;
  static double longitudeFactor;
  ...
}
```



## Einschub: Ungerichtete Graphen

$G = (V, E)$

- Knotenmenge $V$
- Kantenmenge $E$
- Kante $e \in E$, $e \in \{\{u, v\} : u \in V, v \in V\}$
- Knoteninformation: $V \to S$ bei uns $S =$Sender
- Kanteninformation: $E \to K$ bei uns $K = \mathbb{Z}$



## 6.10 Klasse Interferenzgraph

Parametrisierter ungerichter LEDA-Graph

```
class IntfGraph : public UGRAPH<Sender,int> {
public:
  bool consistent(edge e)
  { return inf(source(e)) == unknownChannel ||
           inf(target(e)) == unknownChannel ||
           abs(inf(source(e)).channel -
               inf(target(e)).channel) <=
               inf(e);
  }
  bool consistent()
  { edge e;
    forall_edges(e, *this) {
      if (!consistent(e)) return 0;
    }
    return 1;
  }
 ...
```