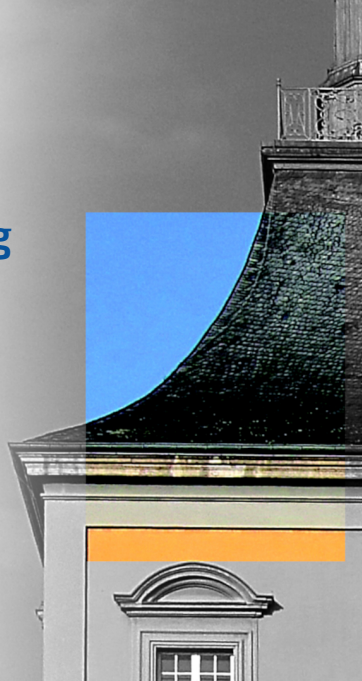
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



Objektorientierte Modellierung

Offene Frage

Was ist Objektorientierung?

Zu welchem Zweck wird objektorientierte Modellierung verwendet?

Was ist ein nichttriviales Beispiel?

UNIVERSITÄT BONN

Literaturempfehlungen

Ausführliche Diskussion der Objektorientierung im Allgemeinen

- Grady Booch et al.. Object-Oriented Analysis and Design with Applications. Addison-Wesley Professional, 2007.

Objektorientierte Programmierung in C++

- Bjarne Stroustrup. The C++ Programming Language, 4th Edition Pearson Education, 2013.
- Bjarne Stroustrup und Herb Sutter. C++ Core Guidelines. `https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md`

Objektorientierte Modellierung

Gründe für Objektorientierung

UNIVERSITÄT BONN

Komplexität

Die Wirklichkeit ist zu komplex um sie vollumfänglich zu verstehen. Deshalb verwenden kluge Menschen starke Vereinfachungen um Modelle zu konstruieren.

Innerhalb dieser Modelle gibt es wiederkehrende Probleme, die mit erprobten oder mit neuen Strategien gelöst werden.

Großen Projekte bilden hier natürlich einen Spezialfall. Es ist üblich, ein großes Projekt in handhabbare Projektbestandteile aufzuteilen. Hier stellt die „geschickte“, „dienliche“ oder „richtige“ Aufteilung das Hauptproblem dar. Es gibt sehr viele Teilstrategien, um möglichst hilfreiche Aufteilungen zu erreichen. Eine moderne Teilstrategie ist die **Objektorientierte Modellierung**.

UNIVERSITÄT BONN

Objektorientierte Modellierung und Objektorientierte Programmierung

Objektorientierte Modellierung wird maßgeblich beim Modellieren und Implementieren von Softwareprojekten verwendet. Erst durch objektorientierte Modellierung wurde die Handhabung von vielen größeren bis sehr großen, verteilt entwickelten Projekten möglich.

Objektorientierte Programmierung ist ein prominentens Beispiel von Objektorientierter Modellierung. Objektorientierte Programmiersprachen erlaubt es, die Denkweise, Methoden und Zusammenhänge der Objektorientierten Modellierung klar und kompakt auszudrücken.

Objektorientierte Programmieung wird besonders bei GUI Programmierung verwendet, weil eine gut zu bedienende GUI autoamtisch die Prinzipien der Objektorientierten Modellierung umsetzt. Ebenso ist die meiste, große Nutzersoftware objektorientiert entwickelt[1].

[1] Es gibt natürlich gute Ausnahmen, wie die Hauptkomponenten der Mediaplayer VLC und MVP.

UNIVERSITÄT BONN

Typische Entwicklungsziele

Einige, wesentliche Entwicklungsziele bei größeren Projekten sind die Folgenden.

- Verteilte, asynchrone (Weiter-)Entwicklung ermöglichen
- Hohe Nachvollziehbarkeit und Wartbarkeit durch klare, konkrete Struktur und Funktionsweise (aus Entwicklung- und Anwendungssicht)
- Allgemeine Schnittstellen um mit anderen Diensten zu interagieren
- Einfache Erweiterbarkeit durch Komponenten
- Allgemeiner Entwurf um flexibel auf unerwartete Voraussetzungen oder Anwensungsgebiete zu reagieren
- Hohe Effizienz und Skalierbarkeit in Enwticklung und Ausführung

Das vollständige Erreichen aller Ziele ist unmöglich. Objektorientierte Modellierung umfasst eine Reihe von erprobten Methodiken und Mustern, um der Erreichnung dieser Zielen nahe zu kommen.

UNIVERSITÄT BONN

Geschichtlicher Verlauf der Softwareentwicklung

Imperativ und Funktionsorientiert (Abstraktion von Operationen):

- Globale Daten und Programmcode ohne Funktionsparameter in einer Datei.
- Globale Daten und Programmcode mit Parametern in einer Datei.
- Globale Daten und Programmcode mit Parametern in mehreren Dateien.

Methodisch geeignet für sehr kleine bis kleine Projekte oder größere Projekte mit sehr wenigen Datentypen.

Objektorientiert (Abstraktion von Datentypen):

- Praktisch keine globalen Daten. Module die aus mehreren Klassen bestehen. Die Instanzen der Klassen interagieren miteinander.

Methodisch geeignet für kleine und große Projekte. Bei sehr großen Projekten werden ganze Applikationen und Dienste nach den Prinzipien der Objektorientierten Modellierung zusammengesetzt.

Geschichtlicher Verlauf der Softwareentwicklung

Impera

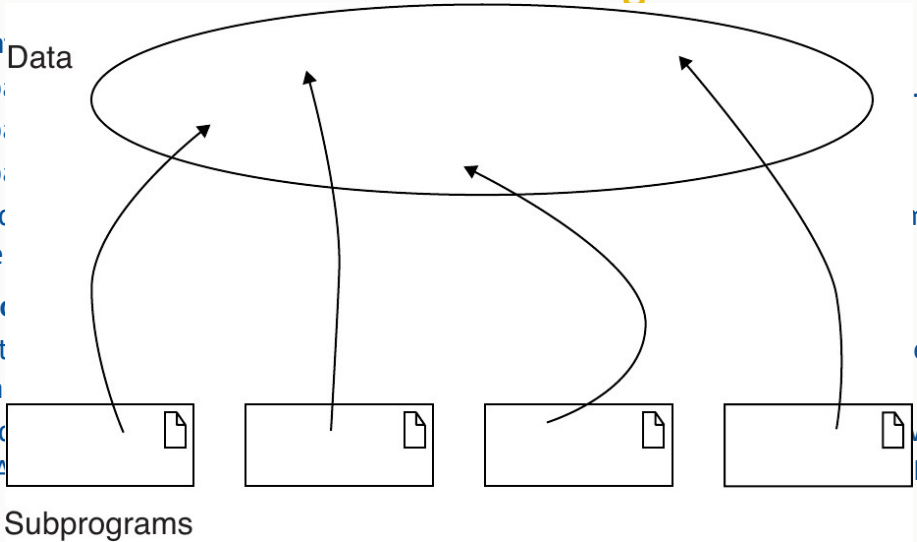
- Glob
- Glob
- Glob

Method nit sehr wenige

Objekt

- Prakt en. Die Insta

Method werden ganze A Model-lierung



Geschichtlicher Verlauf der Softwareentwicklung

Impera

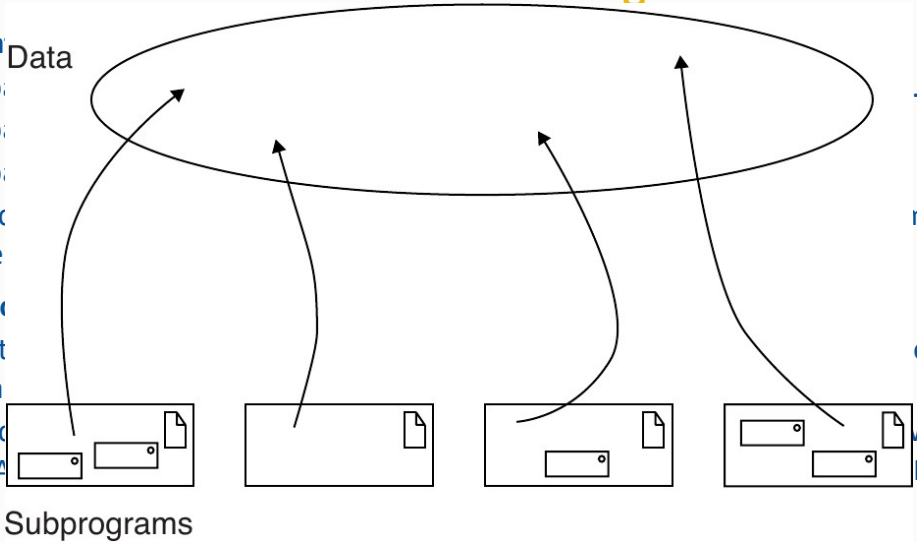
- Glob
- Glob
- Glob

Method nit sehr wenige

Objekt

- Prakt en. Die Insta

Method werden ganze A Model-lierung



Geschichtlicher Verlauf

Impera

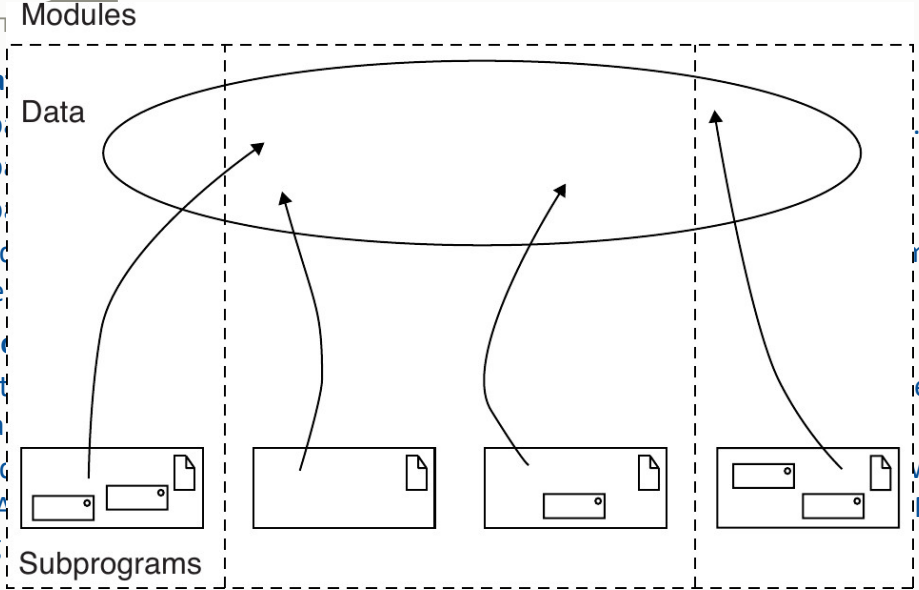
- Glob
- Glob
- Glob

Method nit sehr wenige

Objekt

- Prakt en. Die Insta

Method werden ganze A Model- lierung



Geschichtlicher Verlauf

UNIVERSIT

Impera

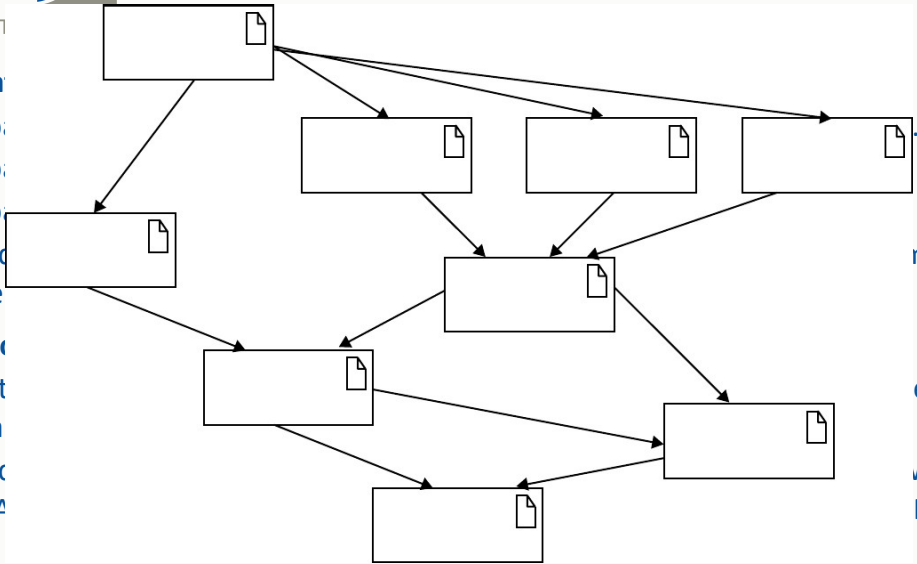
- Glob
- Glob
- Glob

Method nit sehr
wenige

Objekt

- Prakt en. Die
  Insta

Method werden
ganze A Model-
lierung



Zwischenfazit

Große, komplexe Softwareprojekte werden durch die objektorientierte Modellierung in handhabbare Komponenten und Abläufe zerlegt

Das verbessert die Projektqualität ernorm

Haben Sie Fragen?

Objektorientierte Modellierung

Kernprinzipien

Offene Frage

Was sind die Kernprinzipien der Objektorientierten Modellierung?

UNIVERSITÄT BONN

Objektorientierten Modellierung Kernprinzipien

Ansatz: Zerlegung des Projekts in **miteinander interagierende Objekte**.

Zwei **wesentliche Sichtweisen** auf Objekte.

- **Zusammensetzung**: Aus welchen Objekten besteht X?
- **Subtypenbildung**: Welche Objekten verhalten sich allgemeiner / spezieller als X?

Beispiel:

- Gerichtete Graphen besteht aus Knoten und gerichteten Kanten, sodass zwei Knoten durch höchstens eine Kante verbunden werden.
- Gerichtete Graphen sind allgemeiner als gerichtete Graphen mit Kantengewichten und spezieller als gerichtete Graphen mit Mehrfachkanten zwischen Knoten.

UNIVERSITÄT BONN

Konsequenzen der Subtypenbildung

Gewünschte Konsequenz aus **Subtypenbildung**:

- Wenn ein Objekt X mit einem Objekt Y interagieren kann, dann sollen auch Objekte die spezieller sind als X mit Objekten interagieren können, die spezieller als Y sind.

Beispiele:

- Menschen können mit Menschen sprechen. Dann können auch Briefzusteller:innen mit Landwirt:innen sprechen.
- Die Breitensuche für allgemeine Graphen soll ebenso für Graphen möglich sein, die gerichtete oder gewichtete Kanten haben.

Zwischenfazit

Objekteorientierte Modellierung zerlegt ein Projekt in miteinander interagierende Objekte

Beim Zusammenspiel der Objekte sind Zusammensetzung und Subtypenbildung grundlegend

Haben Sie Fragen?

Objektorientierte Modellierung

Objektorientierte Sprachen

Offene Frage

Was sind wichtige, objektorientierte Sprachen?

UNIVERSITÄT BONN

Objektorientierte Modellierungssprachen

Objektorientierte Modellierungssprachen werden entworfen, um die Prinzipien der objektorientierten Modellierung klar und kompakt auszudrücken.

Die visuelle angelegte Sprache **Universal Modelling Language** (UML) ist der Quasisstandard um objektorientierte Modellierung zu kommunizieren.

Im Laufe der Module **Algorithmen und Programmierung** sowie **Praktikum Objektorientierte Softwareentwicklung** werden Sie die Grundlagen von UML erlernen.

Objektorientierte Programmiersprachen

Objektorientierte Programmiersprachen werden entworfen, um die Prinzipien der objektorientierten Modellierung bei der Implementierung von Software klar und kompakt auszudrücken und umzusetzen.

Java Es gibt ausschließlich Objekte[2] und es existieren keine alleinstehenden Funktionen und keine direkte Interaktionsschnittstelle mit zugrundeliegender Hardware

C++ Es gibt hauptsächlich Objekte und auch alleinstehende Funktionen und Interaktionsschnittstellen mit zugrundeliegender Hardware

Python Es gibt viele Objekte und alleinstehende Funktionen und keine direkte Interaktionsschnittstelle mit zugrundeliegender Hardware. Python wird seltener für große Projekte verwendet, da keine starke Typisierung erzwungen wird und Python nicht an sich effizient ist.

[2] Aus Optimierungsgründen verhalten sich primitive Datentypen etwas anders als allgemeine Objekte.

Zwischenfazit

Objektorientierte Modellierung wird in UML ausgedrückt

Objektorientierte Programmierung wird in C++, Java, Python und weiteren ausgedrückt

Haben Sie Fragen?

Objektorientierte Modellierung am Beispiel

Ziel

Um erste Prinzipien der objektorientierten Modellierung zu demonstieren, entwerfen wir ein kleines Projekt

Der Fokus liegt auf interagierenden Objekten und Objektzusammensetzung

UNIVERSITÄT BONN

Projektbeschreibung

Projektziel: Typische Graphen- und Graphenalgorithmen zu Demonstrationszwecken implementieren.

Das hier erklärte Projekt ist klein und es gibt aktuell keine erklärten Langzeitziele. Die größten Stärken der objektorientierten Programmierung kommen erst bei größeren Projekten zu tragen.

Wir beginnen mit der Modellierung von gerichteten Graphen und beachten im Besonderen den **Hauptansatz** (welche Interaktionsschnittstellen sollen gerichtete Graphen anbieten?) und die **Zusammensetzung** (aus welchen Objekten sind gerichtete Graphen zusammgesetzt?).

Objektorientierte Modellierung am Beispiel

Planung und Modellierung

Modellierung

Interaktionsschnittstellen und Zusammensetzung

UNIVERSITÄT BONN

Gerichtete Graphen
Interaktionsschnittstelle

Wie **interagiert** man mit gerichteten Graphen?

- Grapherzeugung
- Knoten hinzufügen / entfernen
- Erfragen ob ein Knoten existiert
- Menge aller Knoten erhalten
- Kante hinzufügen / entfernen
- Erfragen ob eine Kante existiert
- Menge aller Kanten erhalten
- Menge aller Kanten erhalten, die an einem Knoten hinauslaufen / hineinlaufen
- Textrepräsentation (ausdrucken)
- Knotenfolge via Breitensuche / Tiefensuche erhalten
- . . .

UNIVERSITÄT BONN

Gerichtete Graphen
Zusammensetzung

Aus welchen Objekten sind gerichtete Graphen **zusammengesetzt**?

- Knotenmenge (was sind Knotenobjekte?)
- Kantenmenge (was sind Kantenobjekte?)

UNIVERSITÄT BONN

Knotenobjekte

Modellierungsansatz: Uns reicht es hier aus, die Knoten (irgendwie) zu indizieren.

Interaktionsschnittstelle:

- Knotenerzeugung
- Index nennen
- Vergleichsoperator

Zusammensetzung:

- Knoten besteht aus einer festen Zahl oder
- Knoten IST eine feste Zahl (diese Möglichkeit verfolgen wir nicht)

UNIVERSITÄT BONN

Kantenobjekte

Modellierungsansatz: Kanten sind Knotenpaare.

Interaktionsschnittstelle:

- Kantenerzeugung
- Startknoten nennen
- Zielknoten nennen
- Vergleichsoperator

Zusammensetzung:

- Fester Startknoten
- Fester Endknoten

Objektorientierte Modellierung am Beispiel

Implementierung - Projektsstruktur und Headerdateien

Modellierung

Implementierung der Skizze in C++

UNIVERSITÄT BONN

Projektaufbau

Wir legen die übliche Projektstruktur an.

```
.
|
+-- build           // Hier soll das Projekt gebaut werden
|
+-- examples        // Demoprogramme
|
+-- external        // Externe Bibliotheken, z.B. die {fmt}-Bibliothek
|     |
|     +-- fmt       // Namespace der {fmt}-Bibliothek
|
+-- include         // Eigene Includes. Pro Namespace ein Verzeichnis
|     |
|     +-- simpler_gerichteter_graph // Namespace des Graphenprojekts
|
+-- src             // Eigene Quelldateien
|
+-- CMakeLists.txt  // Konfiguration des Buildsystems
```

UNIVERSITÄT BONN

Das Buildsystem CMake im Überblick

Um Softwareprojekte auf möglichst vielen System zuverlässig zu bauen, verwendet man typischerweise **Buildsysteme**.

Die wichtigsten Aufgaben von Buildsystemen sind:

- Prüfung ob die nötigen Voraussetzungen erfüllt sind (sind notwendige Compiler, Bibliotheken und Hilfsprogramme verfügbar?)
- Systembezogene Konfiguration aus allgemein gegebener Definition ableiten
- Softwareprojekt entsprechend der allgemeinen Konfiguration bauen
- Korrekte Funktionsweise des gebauten Softwareprojekt testen

Für C++-Projekte verwenden wir das Buildsystem **CMake**

```cpp
// include/simpler_gerichteter_graph/knoten.hpp
#pragma once
#include <cstddef>

namespace SimplerGerichteterGraph {
// Wir definieren durch welchen Datentyp der Knotenindex angegeben werden soll.
typedef size_t index_t;

// Klasse Knoten
class Knoten {
public:
  // Konstruktur
  Knoten(const index_t knotenindex);

  // Index erhalten
  index_t get_index_t() const;

  // Vergleichsoperator der die Indizes zweier Knoten vergleicht
  bool operator== (const Knoten& other) const;

private:
  const index_t idx; // Der Index wird zur Instanziierung festgelegt.
};

}
```

```cpp
// include/simpler_gerichteter_graph/kante.hpp
#pragma once
#include <simpler_gerichteter_graph/knoten.hpp>

namespace SimplerGerichteterGraph {
// Klasse Kanten
class Kante {
public:
  // Konstruktur
  Kante(const Knoten& startknoten, const Knoten& zielknoten);

  // Erhalte den Startknoten (als Referenz)
  const Knoten& get_startknoten() const;

  // Erhalte den Zielknoten (als Referenz)
  const Knoten& get_zielknoten() const;

  // Vergleichsoperator der die Indizes zweier Knoten vergleicht
  bool operator== (const Kante& other) const;

private:
  const Knoten start;
  const Knoten ziel;
};

}
```

```cpp
// include/simpler_gerichteter_graph/gerichteter_graph.hpp
#pragma once
#include <simpler_gerichteter_graph/knoten.hpp>
#include <simpler_gerichteter_graph/kante.hpp>
...

namespace SimplerGerichteterGraph {

// Knotenmenge mit hashender Menge
class KnotenHash {
public:
  size_t operator() (const Knoten& k) const;
};
typedef std::unordered_set<Knoten, KnotenHash> Knotenmenge;

// Kantenmenge mit hashender Menge
class KantenHash {
public:
  size_t operator() (const Kante& k) const;
};
typedef std::unordered_set<Kante, KantenHash> Kantenmenge;

// Pro Knoten: Menge der auslaufenden / einlaufenden KantenHash
typedef std::unordered_map<Knoten, Kantenmenge, KnotenHash> AuslaufendeKanten;
typedef std::unordered_map<Knoten, Kantenmenge, KnotenHash> EinlaufendeKanten;
```

```
// Klasse Gerichteter Graph
class GerichteterGraph {
public:
  // Konstruktur
  GerichteterGraph();

  // Existierenden Knoten hinzufügen
  void knoten_einfuegen(const Knoten& k);

  // Knoten erstellen und hinzufügen
  void knoten_einfuegen(const index_t& idx);

  // Knoten entfernen (durch Angabe eines existierenden Knoten)
  void knoten_entfernen(const Knoten& k);

  // Knoten entfernen (durch Angabe eines Index)
  void knoten_entfernen(const index_t& idx);

  // Fragt an ob existierender Knoten enthalten ist
  bool existiert_knoten(const Knoten& k) const;

  // Fragt an ob Knoten zu gegebenem Index enthalten ist
  bool existiert_knoten(const index_t& k) const;
```

```
// Gibt Referenz auf enthaltene Knotenmenge zurück
const Knotenmenge& get_knotenmenge() const;

// Existierende Kante hinzufügen
void kante_einfuegen(const Kante& k);

// Kante erstellen und hinzufügen
void kante_einfuegen(const index_t& start, const index_t& ziel);

// Kante entfernen (durch Angabe einer existierenden Kante)
void kante_entfernen(const Kante& k);

// Knoten entfernen (durch Angabe eines Indexpaars)
void kante_entfernen(const index_t& start, const index_t& ziel);

// Fragt an ob existierende Kante enthalten ist
bool existiert_kante(const Kante& k) const;

// Fragt an ob Kante zu gegebenem Indexpaar enthalten ist
bool existiert_kante(const index_t& start, const index_t& ziel) const;

// Gibt Referenz auf enthaltene auslaufende Kantenmenge zurück
Kantenmenge get_auslaufende(const Knoten& start) const;

// Gibt Referenz auf enthaltene auslaufende Kantenmenge zurück
Kantenmenge get_auslaufende(const index_t& start) const;
```

```
  // Gibt Referenz auf enthaltene einlaufende Kantenmenge zurück
  Kantenmenge get_einlaufende(const Knoten& start) const;

  // Gibt Referenz auf enthaltene einlaufende Kantenmenge zurück
  Kantenmenge get_einlaufende(const index_t& start) const;

  // Druckt Graphen aus
  void drucke() const;

  // Breitensuche
  std::vector<Knoten> breitensuche(const Knoten& start) const;

  // Tiefensuche
  std::vector<Knoten> tiefensuche(const Knoten& start) const;

private:
  Knotenmenge knotenmenge;
  Kantenmenge kantenmenge;
  AuslaufendeKanten auslaufendekanten;
  EinlaufendeKanten einlaufendekanten;
};

}
```

UNIVERSITÄT BONN

Demonstration Dummy

Um die syntaktische Korrektheit der Header zu testen, legen wir zunächst eine sehr einfache Demo an.

```cpp
// examples/demo1.cpp
#include <simpler_gerichteter_graph/gerichteter_graph.hpp>

#include <iostream>

int main() {
  std::cout << "Hier gibt es noch nichts zu sehen" << std::endl;
}
```

UNIVERSITÄT BONN

Projekt bauen und ausführen - Konsole

Wir testen ob das Projekt baut. Dazu öffnen wir die Konsole und beginnen im Wurzelverzeichnis des Projekts. Zunächst erstellen wir (falls nötig) den Ordner `build`.

```
mkdir -p build
```

Nun wechselen wir in das Buildverzeichnis.

```
cd build
```

Nun bereiten wir das Buildsystem vor.

```
cmake ..
```

Nun bauen wir das Projekt.

```
make
```

Nun rufen wir das gebaute Projekt auf.

```
./demo1
```

UNIVERSITÄT BONN

Projekt bauen und ausführen - IDE

Sofern eine korrekte CMake-Datei im Projektverzeichnis enthalten ist, stellen die meisten, modernen IDEs einen zur Verfügung, der diese Schritte in der Konsole automatisieren soll.

UNIVERSITÄT BONN

CMake im Detail

```
#
# Konfiguration des Buildsystems CMake
#

# Minimale Version des Buildsystems
cmake_minimum_required(VERSION 3.17)
# Name des Projekts
project(SimplerGerichteterGraph)

#
# Optionen
#

# Setzte verwendeten C++-Standard auf C++17
set(CMAKE_CXX_STANDARD 17)

# Prüfe ob Heapspeicher gefunden wird,
# der nicht freigegeben wurde
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

```
# Füge selbsgeschriebene Includes hinzu
include_directories(include)
# Füge externe Includes hinzu
# include_directories(external)

#
# Baue Programme
#

# Baue das Programm 'demo1' aus den
# genannten Quelldateien
add_executable(demo1
  examples/demo1.cpp
  src/knoten.cpp
  src/kante.cpp
  src/gerichteter_graph.cpp)
```

Objektorientierte Modellierung am Beispiel

Implementierung - Quelldateien

```cpp
// src/knoten.cpp
#include <simpler_gerichteter_graph/knoten.hpp>

namespace SimplerGerichteterGraph{
  // Konstruktur
  Knoten::Knoten(const index_t knotenindex) :
    idx(knotenindex)
  {}

  // Vergleichsoperator der die Indizes zweier Knoten vergleicht
  bool Knoten::operator== (const Knoten& other) const {
    return idx == other.idx;
  }

  // Index erhalten
  index_t Knoten::get_index_t() const {
    return idx;
  }
}
```

```cpp
// src/kante.cpp
#include <simpler_gerichteter_graph/kante.hpp>

namespace SimplerGerichteterGraph {
  // Konstruktur
  Kante::Kante(const Knoten& startknoten, const Knoten& zielknoten) :
    start(startknoten), ziel(zielknoten)
  {}

  // Erhalte den Startknoten (als Referenz)
  const Knoten& Kante::get_startknoten() const {
    return start;
  }

  // Erhalte den Zielknoten (als Referenz)
  const Knoten& Kante::get_zielknoten() const {
    return ziel;
  }

  // Vergleichsoperator der die Indizes zweier Knoten vergleicht
  bool Kante::operator== (const Kante& other) const {
    return start == other.start and ziel == other.ziel;
  }
}
```

```cpp
// src/gerichteter_graph.cpp
#include <simpler_gerichteter_graph/gerichteter_graph.hpp>

#include <iostream>

namespace SimplerGerichteterGraph {
  // Naive Hashfunktion für Knoten
  size_t KnotenHash::operator()(const Knoten& k) const {
    return k.get_index_t();
  }

  // Naive Hashfunktion für Kanten (Hash(Start) XOR Hash(Ziel))
  size_t KantenHash::operator()(const Kante& k) const {
    return k.get_startknoten().get_index_t() ^ k.get_zielknoten().get_index_t();
  }
```

```cpp
// src/gerichteter_graph.cpp
  GerichteterGraph::GerichteterGraph()
  {}

  void GerichteterGraph::knoten_einfuegen(const Knoten& k) {
    knotenmenge.emplace(k);
  }

  void GerichteterGraph::knoten_einfuegen(const index_t& idx) {
    knoten_einfuegen(Knoten(idx));
  }
```

```
void GerichteterGraph::knoten_entfernen(const Knoten& k) {
  knotenmenge.erase(k);
  // Entferne alle Kanten, die k enthalten

  // Entferne alle auslaufenden Kanten
  if (auslaufendekanten.count(k) > 0) {
    for (const auto& e : auslaufendekanten[k]) {
      kantenmenge.erase(e);
    }
    auslaufendekanten.erase(k);
  }

  // Entferne alle einlaufenden Kanten
  if (einlaufendekanten.count(k) > 0) {
    for (const auto& e : einlaufendekanten[k]) {
      kantenmenge.erase(e);
    }
    einlaufendekanten.erase(k);
  }
}

void GerichteterGraph::knoten_entfernen(const index_t& idx) {
  knoten_entfernen(Knoten(idx));
}
```

```cpp
bool GerichteterGraph::existiert_knoten(const Knoten& k) const {
  return knotenmenge.count(k) > 0;
}

bool GerichteterGraph::existiert_knoten(const index_t& idx) const {
  return existiert_knoten(Knoten(idx));
}

const Knotenmenge& GerichteterGraph::get_knotenmenge() const {
  return knotenmenge;
}
```

```
  void GerichteterGraph::kante_einfuegen(const Kante& k) {
    const auto& startknoten = k.get_startknoten();
    const auto& zielknoten = k.get_zielknoten();
    knoten_einfuegen(startknoten);
    knoten_einfuegen(zielknoten);

    // Wir wollen keine Schlaufen
    if(not (startknoten == zielknoten)) {
      kantenmenge.emplace(k);
      auslaufendekanten[startknoten].emplace(k);
      einlaufendekanten[zielknoten].emplace(k);
    }
  }

  //
  // Und so weiter und so weiter...
  //
}
```

```cpp
// examples/demo1.cpp
#include <simpler_gerichteter_graph/gerichteter_graph.hpp>

#include <iostream>

int main() {
  SimplerGerichteterGraph::GerichteterGraph g;

  g.knoten_einfuegen(1);
  g.knoten_einfuegen(2);
  g.knoten_einfuegen(42);
  g.drucke();

  g.kante_einfuegen(1,2);
  g.kante_einfuegen(2,42);
  g.drucke();

  g.kante_einfuegen(2,5);
  g.kante_einfuegen(42,1);
  g.kante_einfuegen(42,2);
  g.kante_einfuegen(42,3);
  g.drucke();

  g.kante_entfernen(42,1);
  g.knoten_entfernen(1);
  g.drucke();
```

Zusammenfassung

Durch die objektorientierte Modellierung wird das Projekt in sehr kleine, handhabbare Bestandteile zerlegt

Die einzelnen Memberfunktionen enthalten simpelen Programmcode

Objektorientierte Modellierung am Beispiel

Zusammenfassung und Ausblick

Zusammenfassung

Gerichtete Graphen mit Kantengewichten sind konzeptionell mit gerichteten Graphen verwandt

Offene Frage

Wie werden verwandtschaftliche Beziehungen in objektorientierten Modellen behandelt?

Teilfrage: Wie wird wiederkehrender Code für verwandte Typen reduziert?