Portierung einer Graph-Bibliothek von Java nach C++

Heinrich Drobin Galina Engelmann Manuel Webersen

25. Juli 2011

Inhaltsverzeichnis

1. Einleitung

Im vorliegenden Projekt geht es um die Portierung der Java-Bibliothek jGraphT in eine äquivalent funktionierende C++-Bibliothek. Mit jGraphT ist es möglich, Graphen (im Sinne der Graphentheorie [1]) zu modellieren und in eigenen Programmen zu nutzen. Diese Funktionalität soll hier für C++-Programme zugänglich gemacht werden.

Während der Portierung gilt es, die spezifischen Eigenschaften der beiden Sprachen Java und C++ zu beachten. Auf den ersten Blick scheinen sie sich auf Grund ihrer syntaktischen Ähnlichkeit nur leicht zu unterscheiden, doch gerade beim Zugriff auf Variablen und besonders Objekte verhalten sie sich anders. Die während der Portierung vorgenommenen Codeänderungen und die dabei aufgetretenen Probleme sind in Kapitel 4 beschrieben.

2. Rahmenbedingungen

2.1. Graphentheorie

“Die Graphentheorie ist ein Teilgebiet der Mathematik, das die Eigenschaften von Graphen und ihre Beziehungen zueinander untersucht”[2].

Graphentheorie findet überall dort Anwendung, wo es ein algorithmischen Problem zu lösen gibt. Das Grundmodell ist ein Graph, “in der Graphentheorie eine abstrakte Struktur, die eine Menge von Objekten zusammen mit den zwischen diesen Objekten bestehenden Verbindungen repräsentiert.”[3]. Über ein Graphmodell lassen sie Probleme veranschaulichen und algorithmische Ansätze zur Lösung aufstellen. Ein klassisches Beispiel ist z. B. die Suche nach dem kürzesten Weg zwischen zwei Orten. Abb. 2.1 zeigt den zugehörigen Graphen. Die Kanten des Graphen sind mit Weglängen gewichtet und über geeignete Algorithmen wird daraus der kürzeste Weg berechnet. Nach einem ähnlichen Prinzip werden im Bauwesen (Bauablaufplanung), Verkehrs- und Strromnetz (Abstände), in der Chemie (Molekülstrukturen) und vielen anderen Bereichen Probleme gelöst.

2.2. jGraphT

JGraphT ist eine freie Java-Bibliothek, die im Rahmen der mathematischen Graphentheorie Algorithmen und Klassen zur Verfügung stellt. JGraphT unterstützt verschiedene Graphentypen, beginnend mit ungerichteten sowie gerichteten bis hin zu Graphen mit gewichteten, benutzerdefinierten Kanten. Auch Multigraphen (Graphen mit mehr als einer Kante zwischen zwei Knoden), Pseudographen (erlaubt Mehrfachkanten und Schleifen), unveränderliche Graphen (“read-only”), sowie die Möglichkeiten zum Umgang mit Subgraphen oder Graphen im Umfeld der ereignisorientierten Programmierung stehen zur Verfügung. Die erzeugten Graphen können außerdem in zahlreiche Formate (z. B.

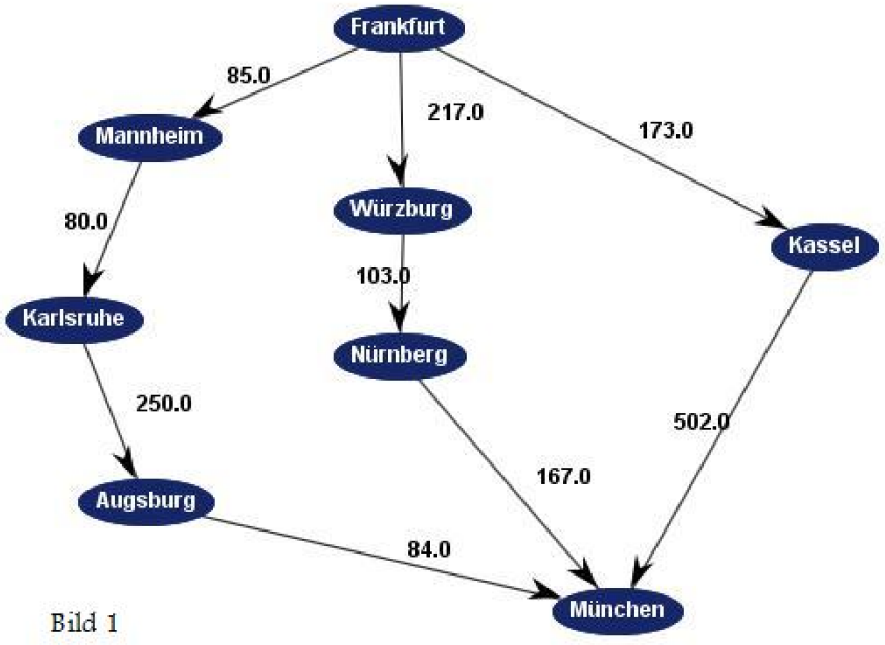

Abbildung 2.1.: Gewichteter Graph [4]

Matlab, CSV, XML) exportiert oder mit der zusätzlichen Bibliothek jGraph (ohne "T") visualisiert werden. Die jGraphT-Bibliothek kann unter [5] heruntergeladen werden.

Abb. 2.2 zeigt die Homepage des jGraphT-Projektes, die in einer Tabelle die Auflistung aller enthaltenen Pakete enthält und deren Funktion kurz beschreibt. Für jedes Paket können über die Menüleiste, oder duch unmittelbares Anklicken des Pakets, Informationen zu enthaltenen Schnittstellen (in der Leiste "Package"), deren Methoden (Method Summary) und den Klassen, die diese Schnittstellen nutzen ("Class"), angesehen werden. Jede Methode und Klasse erhält wiederum eine detailierte Funktionsbeschreibung und Dokumentation der Parameter, Ausgaben und Abhängigkeiten (Method Detail). Unter dem Menüpunkt "Tree" sind die hierarchischen Abhängigkeiten der Pakete und der Klassen angegeben.

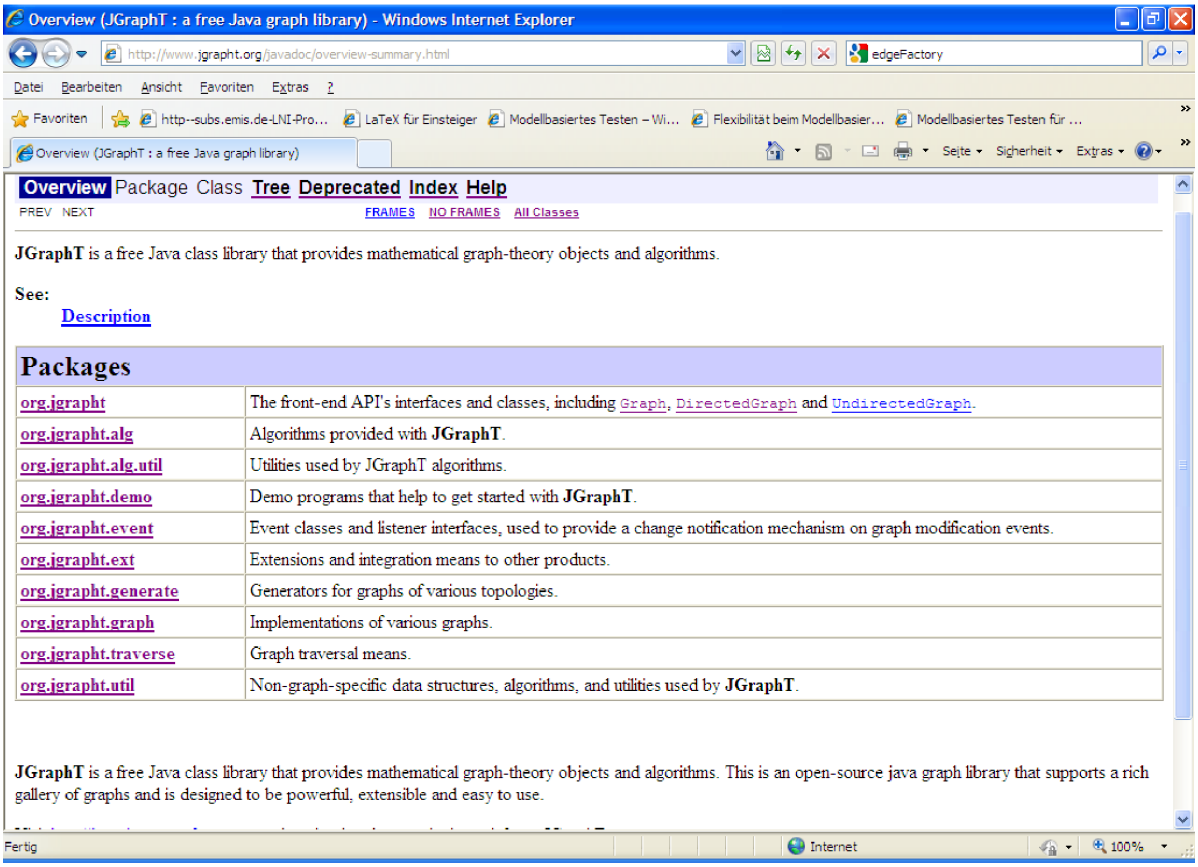

Abbildung 2.2.: Startseite der jGraphT- Bibliothek

2.3. Arbeitsumgebung

2.3.1. Mercurial

Für den Austausch von Daten innerhalb der Projektgruppe wurde ein Repository angelegt. Eine geeignetes Tool für diesen Zweck ist das plattformunabhängige Versionskontrollsystem zur Softwareentwicklung Mercurial [6]. Die Funktion dieses Systems kann im Wesentlichen durch die drei Begriffe “clone”, “branching” und “merging” beschrieben werden. Mercurial erlaubt den Zugriff auf Daten eines anderen Repositories, sodass jeder Beteiligter sich dessen Inhalt als Kopie auf den lokalen Rechner holen, also die Projektdaten “klonen” kann. Nach der Bearbeitung werden die Daten wieder mit den Repositories der anderen Beteiligten abgeglichen, wobei die Änderungen zusammengeführt werden (“merging”). Das “branching” ist das Erzeugen einer Verzweigung und wird automatisch bei der Übergabe eines Klons im lokalen Repository erzeugt. Eine ausführliche Beschrei-

bung der Installationsschritte und Benutzung von Mercurial gibt es unter [7]. Bedient wird Mercurial meist über die Kommandozeilenbefehle oder altenativ für Windows Betriebssysteme auch über eine einfache graphische Oberfläche namens `TortoiseHg` [8].

2.3.2. Mantis

Zur Fehlerverfolgung und Verwaltung der einzelnen Aufgaben wurde der Web-basierte Mantis Bug Tracker verwendet. In diesem werden zu lösende Aufgaben aufgelistet. Die Projektentwickler haben dann die Möglichkeit, über Status-Felder Aufgaben für die Bearbeitung auszuwählen und mit Kommentaren, zum Beispiel über den Fortschritt, auftretende Fehler und ähnliches zu berichten. Mantis Bug Tracker ist gleichfalls eine freie Software [9].

2.3.3. GCov

Bei der Entwicklung von Testfällen ist es wünschenswert, einen möglichst hohen Überdeckungsgrad des zu testenden Codes zu erreichen. Bei der Analyse der Überdeckung (Code Coverage) wird ermittelt, welche Zeilen eines Programms ausgeführt wurden und wie oft diese ausgeführt wurden. Deswegen ist es möglich die Anweisungs- und Zweigüberdeckung von C/C++-Programmen zu messen (C0 und C1 Test). Fehlt die Überdeckung einer Codekomponente, kann der Programmierer daraus schließen, dass das Stück Code oder Segment eines Programms fälschlicherweise nicht ausgeführt oder nicht ausreichend betrachtet bzw. getestet wurde. Daraufhin kann er den Auslöser suchen oder eine entsprechende Test-Methode schreiben. Gcov (GNU COVerage tool) dient der Umsetzung von Code Coverage-Tests für C und C++ Code und es gibt ein Plugin für Eclipse. Die Nutzung von Gcov ist sehr einfach, es müssen lediglich zwei Optionen beim Kompilieren zusätzlich angegeben werden (`-fprofile-arcs -ftest-coverage`) und schon lassen sich Programme mit Gcov auswerten. Während des Compilierens werden *.gcno - Files erstellt. Diese enthalten Source-Referenzen. Bei der Ausführung werden dann *.gcda - Files generiert. Diese enhalten die Laufzeitauswertung. Das Eclipse-Plugin parst dann (nach einem erneuten Build-Anstoss) die Ausgaben von “GCOV” und färbt die Zeilen in Eclipse rot oder grün ein. Gcov ist unter [10] zu finden.

```
/* gcov_test.cpp */

int main()
{
    int var;
    for(int i=0; i<10; i++)
        if (i==100)
            int var = 0;
        else
            int var = 1;
    return 0;

}
```

Abbildung 2.3.: Screenshot eines mit Gcov getesteten Programms

2.3.4. Doxygen

Zu Dokumentationszwecken wurde das Open-Source-Dokumentationstool Doxygen [11] verwendet. Dieses erstellt automatisch eine Dokumentation zu einem kommentierten Code und gibt einen Überblick über alle Klassenelemente sowie Abhängigkeiten der Klassen und Methoden untereinander. Weitere Informationen werden aus Kommentaren mit bestimmten Stichworten generiert, z. B. Angaben zum Autor oder zum Erstellungsdatum.

3. Planung

3.1. Klassendiagramm

Klassendiagramme bieten eine erste Übersicht über ein System, dessen interne Klassenabhängigkeiten, Methoden und Schnittstellen. Auf Basis eines Klassendiagramms lassen sich leichter organisatorische Aspekte wie Planung, Aufgabeneinteilung durchführen oder der zeitliche Aufwand einschätzen.

Die UML-Diagramme aus Abb. 3.1 und Abb. 3.2 zeigen die im Projekt bearbeiteten Aufgaben. Darin sieht man in Abb. 3.1 alle Schnitstellen des Pakets `org.jgrapht` und in Abb. 3.2 alle Klassen und Schnittstellen der beiden Pakete, `org.jgrapht` und `org.jgraph.jgrapht`, die portiert wurden. In Abb. 3.1 sind neben der Art der Verbindungen auch Methoden (und Konstanten) der Schnittstellen angegeben. Für die Abb. 3.2 wurde aus Gründen der Übersichtlichkeit auf die Auflistung der Methoden verzichtet. Die Pfeilbeschriftungen zeigen die Verbindungsart an: Mit "Import" (Modelle, Klassen, ganze Packete an den Verwendungsort einfügen) und "Implement" (z. B. mit Methoden anderer Schnittstellen erweitern) wird die Verwendung, mit "Derive" die Abstammung und mit "Instantiate" die Erzeugung eines neuen Objektes gekennzeichnet.

Die Wahl der zu portierenden Klassen ergab sich teilweise aus den Abhängigkeiten der Klassen und Schnittstellen untereinander. Eine detailiertere Darstellung der Zusammenhänge kann in der Doxygen-Dokumentation nachgeschlagen werden.

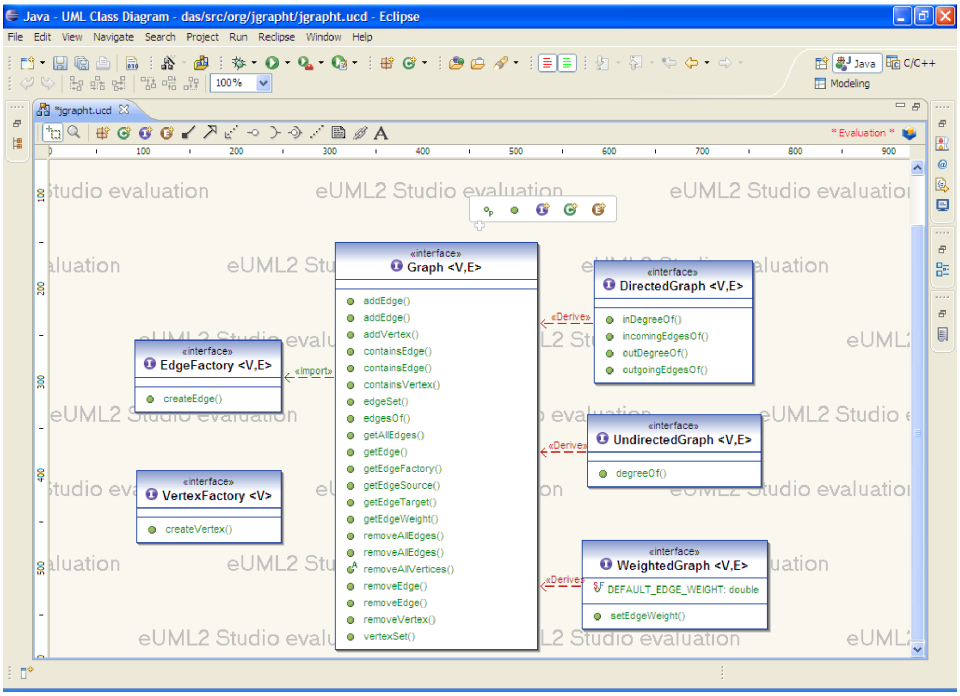

Abbildung 3.1.: Screenshot UML-Klassendiagramm des `org.jgrapht`

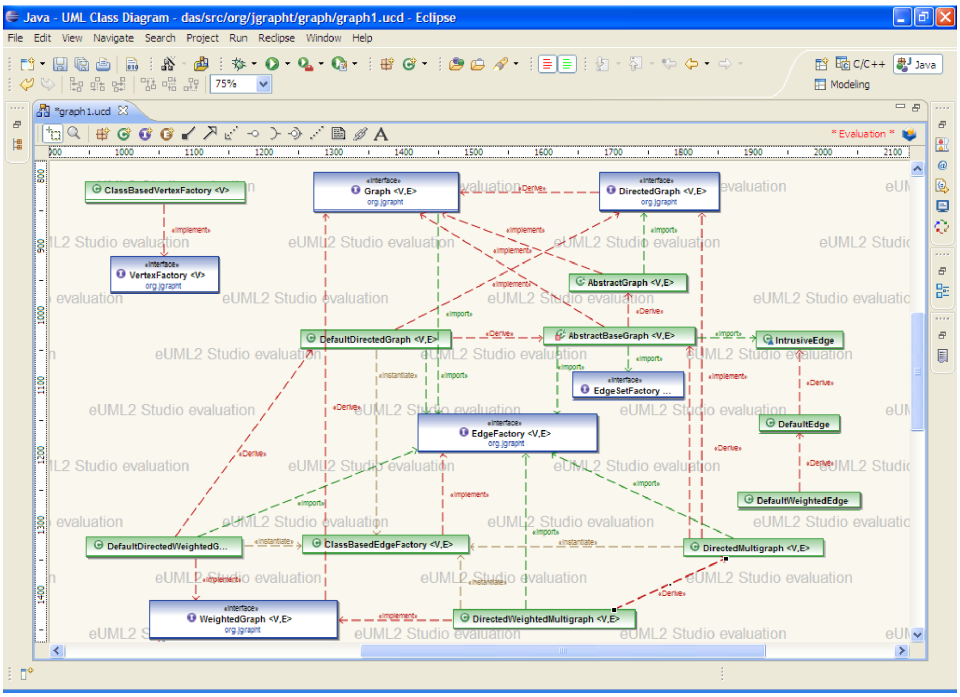

Abbildung 3.2.: Screenshot UML-Klassendiagramm

3.2. Aufgabeneinteilung

Aufgaben	**Bearbeitung durch**
Klassenportierung	Heinrich Drobin Galina Engelmann Manuel Webersen
Gcov-Test	Heinrich Drobin
Fehlerkorrektur Implementierung Beispielprogramm	Manuel Webersen
Codedokumentation	siehe Anhang A
UML-Klassendiagramm Doxygen- und Abschlussdokumentation	Galina Engelmann

3.3. Zeitplan

Woche	Code	Test
Woche 1 16.05 - 22.05	Einarbeitung jGraphT Festlegung Coding Conventions Einrichtung Bugtracker	Einarbeitung CppUnit
Woche 2 23.05 - 29.05	Einrichtung Repository Auswahl der Klassen UML-Diagramm jGraphT	Repository Erstellung Test-Framework
Woche 3 30.05 - 05.06	UML-Diagramm Auswahl UML-Diagramm Portierung Port.[1] Graph-Interfaces Port.[1] Factory-Interfaces Port.[1] Utils Port.[1] AbstractGraph Port.[1] „Graphs“ & Abh.	-
Woche 4 06.06 - 12.06	Port.[1] Factories Port.[1] Edges Port.[1] AbstractBaseGraph	Testf. Graph-Interfaces[2] Testf. Factory-Interfaces[2] Testf. Utils Testf. AbstractGraph Testf. „Graphs“ & Abh.
Woche 5 13.06 - 19.06	Port.[1] Graph-Nachkommen Port.[1] Generators ggf. Port.[1] Algorithmen	Testf. Factories Testf. Edges Testf. AbstractBaseGraph
Woche 6 20.06 - 26.06	Konzeption Beispielprogramm ggf. Port.[1] Algorithmen (Forts.)	Testf. Graph-Nachkommen Testf. Generators ggf. Testf. Algorithmen
Woche 7 27.06 - 03.07	Umsetzung Beispielprogramm	Codeabdeckung
Woche 8 04.07 - 10.07	Problembehebung	Testf. Beispielprogramm Alle Lib-Testfälle erfolgreich
Woche 9 11.07 - 17.07	Pufferwoche Abgabefertigkeit	Alle Testfälle erfolgreich Abgabefertigkeit

Woche	Dokumentation
Woche 1 16.05 - 22.05	Einarbeitung Doxygen Festlegung Kommentar-Struktur
Woche 2 23.05 - 29.05	Repository Einarbeitung LaTeX
Woche 3 30.05 - 05.06	LaTeX-Grundgerüst Doku-Formatierung Klassenauswahl
Woche 4 06.06 - 12.06	Organisatorisches und Zeitplan
Woche 5 13.06 - 19.06	-
Woche 6 20.06 - 26.06	-
Woche 7 27.06 - 03.07	Beispielprogramm
Woche 8 04.07 - 10.07	Homogenisierung
Woche 9 11.07 - 17.07	Abgabefertigkeit

3.4. Auswahl der zu portierenden Klassen

Zur Portierung wurden Schnittstellen und Klassen aus den Paketen `org.jgrapht.graph`, das alles zur Implementation verschiedenartiger Graphen enthält, und aus `org.jgrapht`, das die wichtigsten Basisschnittstellen enthält, ausgewählt. `Graph` ist eine der Basisschnittstellen. Sie enthält Grundbausteine für die Modellbildung. Weitere gewählte Schnittstellen sind `EdgeFactory`, `VertexFactory`, `EdgeSetFactory` zur Erzeugung von neuen Kanten, Knoten und Hinzufügen von Kanten an bestimmte Knoten, sowie die Subschnittstellen `DirectedGraph`, `UndirectedGraph` und `WeightedGraph` für die jeweiligen Graphtypen.

[1] Port. meint die Portierung und den jeweils zugehörigen Beitrag zur Dokumentation.
[2] Es können nicht die Interfaces selbst getestet werden, sondern nur die davon abgeleiteten Klassen.

Die Klasse `AbstractGraph` erbt von `Graph` und bildet das Skelett eines Graphen. Die wichtigste Klasse ist `AbstractBaseGraph`, die von `AbstractGraph` erbt und den allgemeinen Aufbau eines Graphen implementiert. Die Klassen `ClassBasedEdgeFactory` und `ClassBasedVertexFactory` implementieren die Funktionalität zur Erzeugung neuer Kanten und Knoten und werden innerhalb von `AbstractBaseGraph` genutzt.

Schließlich stellen die Klassen

- `DefaultDirectedGraph`
- `DefaultDirectedWeightedGraph`
- `DirectedMultigraph`
- `DirectedWeightedMultigraph`

die konkreten Standard-Implementierungen der jeweiligen Graphtypen dar, die allesamt von `AbstractBaseGraph` abgeleitet werden.

4. Portierung

4.1. Interfaces

4.1.1. Graph

Das Interface `Graph` konnte relativ einfach als abstrakte Klasse in C++ umgesetzt werden. Zu beachten war hierbei neben der Deklaration aller Methoden als "pure virtual" die Tatsache, dass die Methoden `removeAllEdges()` und `removeAllVertices` in jGraphT als Parameter ein Objekt erwarten, das das Interface `Collection` implementiert (z. B. `Set`, `ArrayList` usw.). Die C++-Entsprechungen dieser Klassen werden jedoch nicht von einer gemeinsamen Basisklasse abgleitet, sodass hier nur eine Lösung durch überladen der Methoden in Frage kommt. Zunächst wurde jedoch nur ein Prototyp festgelegt, der als Parameter ein `set` erwartet. Zusätzlich wurden in `Graph` die Kommentare derart angepasst, dass sie statt der Standard-Exceptions von Java auf die von C++ verweisen.

4.1.2. DirectedGraph, UndirectedGraph und WeightedGraph

Die Interfaces `DirectedGraph`, `UndirectedGraph` sowie `WeightedGraph` konnten ohne weitere Änderungen als abstrakte Klasse in C++ umgesetzt werden. Zu beachten war hierbei lediglich die Deklaration aller Methoden als "pure virtual".

4.1.3. EdgeFactory

Das Interface `EdgeFactory` konnte ohne weitere Änderungen als abstrakte Klasse in C++ umgesetzt werden. Zu beachten war hierbei lediglich die Deklaration aller Methoden als "pure virtual".

4.1.4. EdgeSetFactory und VertexFactory

Die abstrakte Klassen `EdgeSetFactory` und `VertexFactory` entsprechen der "pure virtual" Funktion in C++ , die mit "0" initialisiert werden müssen.

4.2. Klassen

4.2.1. AbstractGraph

Bei der Klasse `AbstractGraph` mussten einige Änderungen gegenüber der Vorlage vorgenommen werden:

Für die Methoden `removeAllEdges()`, `removeAllVertices()` sowie `toStringFromSets()` war die Nutzung eines Iterators für die als Parameter übergebenen Sets notwendig, die im Gegensatz zu Java etwas aufwendiger ist. Wichtig war hierbei insbesondere, dass eine Deklaration wie `set<V>::iterator it;` einen nicht eindeutigen Befehl darstellt und daher mit dem Schlüsselwort `typename` eingeleitet werden muss. Andernfalls wird der Befehl als Aufruf einer Methode interpretiert und führt somit zu einem Fehler.

Für die Methoden `toString()` und `toStringFromSets()` musste die Typüberprüfung angepasst werden, die in C++ mit dem Schlüsselwort `typeid` realisiert werden kann (nach Einbindung der Bibliotheksdatei typeinfo.h). Weitere Probleme im Zusammenhang mit der `toString()`-Methode werden im Abschnitt 4.2.5 beschrieben.

Außerdem mussten in der Methode `assertVertexExist()` die Java-Exceptions durch die in C++ verfügbare `invalid_argument`-Exception ersetzt werden.

4.2.2. AbstractBaseGraph

Mit rund 1200 Zeilen ist `AbstractBaseGraph` mit Abstand die größte der ausgewählten Klassen, was u. a. auf die zahlreichen inneren Klassen zurückzuführen ist. Von diesen ging auch bereits das erste Detail der Implementierung aus, nämlich eine Umsortierung der Klassen innerhalb der Datei. Da in C++ eine Klasse vor ihrer ersten Nutzung definiert werden muss, also zwingend weiter vorn im Dokument stehen muss, war dies unumgänglich.

In den inneren Klassen `DirectedEdgeContainer` und `UndirectedEdgeContainer` werden in einigen Methoden unveränderliche Sets zurückgegeben. Während es in Java hierfür offenbar eigene Klassen gibt, kann ein entsprechendes Verhalten relativ einfach durch die Deklaration der Variablen als `const set<E*>*`, d. h. als Zeiger auf ein konstantes Set umgesetzt werden.

Einige Details der Java-Implementierung könnten in C++ nicht vollwertig umgesetzt werden, etwa die Verwaltung von EdgeSets mit der `ArrayList`-Klasse - diese existiert nämlich nur in Java. Für die Portierung wurde hier auf die sonst üblichen `set`s zurückgegriffen, wodurch allerdings einige Bezeichner (z. B. bei der Klasse `ArrayListFactory`) nicht mehr zutreffend sind. Um eine möglichst hohe Übereinstimmung zwischen Original und Portierung zu erreichen, wurden diese jedoch nicht verändert.

Auch die gegenüber dem Original eingefügte Abhängigkeit aller Edge-Instanzen vom jeweiligen Vertex-Typ tritt bei der Portierung des `AbstractBaseGraph` zutage.

Des weiteren traten auch hier wieder die bereits zuvor beschriebenen Phänomene der zu ersetzenden Exception-Klassen, der Typvergleiche sowie der Nutzung von Iteratoren auf.

4.2.3. ClassBasedEdgeFactory und ClassBasedVertexFactory

Die abstrakte Klassen `ClassBasedVertexFactory` und `ClassBasedEdgeFactory` konnten ohne weitere Änderungen in C++ umgesetzt werden. Es wurden lediglich die Standard-Exceptions von Java auf die von C++ angepasst.

4.2.4. DefaultDirectedGraph und DefaultDirectedWeightedGraph

Die Basisklasse `DefaultDirectedGraph` und die Schnittstelle `WeightedGraph` bzw. die Basisklasse `AbstractBaseGraph` und die Schnittstelle `DirectedGraph` für `DefaultDirectedWeightedGraph` wurden durch Mehrfachvererbung eingebunden. Der Ausdruck Class<?extends E> liefert das Class-Exemplar, dass die Klasse des Objekts E repräsentiert und entspricht in C++ dem im template definierten Datentyp E. Im parametrisierten Konstruktor wurde ein Exemplar der Klasse ClassBasedEdgeFactory

erzeugt, die this-Referenz zeigt auf die Adresse der neuen Instanz. Im zweiten Konstruktor ruft der Java-Befehl super() den Default-Konstruktor der Oberklasse auf. In C++ entspricht dieser Befehl der Vererbung / Anhängen des Oberklassenkonstruktors. Zusätzlich wird noch ein Destruktor definiert.

4.2.5. DefaultWeightedEdge, DefaultEdge und IntrusiveEdge

Bei der Portierung der Edge-Klassen traten zwei Probleme durch die Unterschiede zwischen C++ und Java auf. Erstens hängen in Java alle Klassen schlussendlich von der Basisklasse `Object` ab, was die Nutzung beliebiger Objekte mit einer Variable vom Typ `Object` ermöglicht. In C++ ist dies nicht der Fall, sodass ein ähnliches Verhalten nur über void-Pointer oder über Templates erzeugt werden kann. Das zweite Problem liegt in der Verfügbarkeit der Methode `toString()`, die in Java von `Object` implementiert wird. Durch diese Methode ist es möglich, beliebige Objekte in einer String-Konkatenation zu verwenden, ohne dass es zu Typkonvertierungsfehlern kommt. In C++ fehlt diese Methode jedoch, allein schon auf Grund des Fehlens der Basisklasse.

Der gewählte Lösungsansatz besteht darin, die Edge-Klassen wiederum als Templates auszulegen, deren Parameter der Typ der zu verarbeitenden Knoten (Vertices) ist. Auf diese Weise kann sogar auf die relativ aufwendige Verarbeitung von void-Pointern verzichtet werden. Die Template-Klasse `IntrusiveEdge` implementiert somit ihre Attribute `source` und `target` als Variablen vom Typ "V*", also einem Zeiger auf den späteren Datentyp der Knoten.

Die Nachbildung der `toString()`-Methode ist dann relativ einfach durch Nutzung von `ostream`s und Überladung des `«`-Operators für eigene Vertex-Klassen möglich. Die Implementierung dieser Methode muss jedoch leider vorausgesetzt werden und kann (im Gegensatz zu Java) auch nicht implizit erfolgen. Allerdings bleibt auf diese Weise die Kompatibilität mit den primitiven Datentypen, die natürlich ebenfalls als Template-Parameter verfügbar sein sollten, erhalten.

Schließlich war bei der Klasse `DefaultWeightedEdge` zu beachten, dass für die Attribute einer Klasse keine Standardwerte zur Initialisierung festgelegt werden dürfen. Um dies zu umgehen wurde ein Konstruktor definiert, der das Attribut `weight` entsprechend auf den in `WeightedGraph` festgelegten Standardwert setzt.

4.2.6. DirectedMultigraph und DirectedWeightedMultigraph

Die abstrakten Klassen `DirectedMultigraph` und `DirectedWeightedMultigraph` konnten ohne weitere Änderungen in C++ umgesetzt werden. Zusätzlich wurden in der Klassen die virtuelle Destruktoren eingefügt.

4.3. Weitere Klassen

Die Umsetzung einiger weiterer Klassen, z. B. der konkreten Graph-Implementierungen

- Multigraph
- WeightedMultigraph
- SimpleGraph
- SimpleWeightedGraph

oder der Generatoren

- GraphGenerator
- CompleteGraphGenerator
- EmptyGraphGenerator

konnte auf Grund des vorzeitigen Ausscheidens einiger Teammitglieder nicht durchgeführt werden.

5. Test

5.1. Unit-Test

Der Unit-Test konnte auf Grund des Ausscheidens der damit beauftragten Teammitglieder nicht durchgeführt werden.

5.2. Codeabdeckung

Für die Gewährleistung einer gewissen Qualität des Programms wurden einige Tests mit Hilfe des Code-Coverage-Tools GCov durchgeführt. Als Testfälle wurden die verschiedenen Graphen modelliert (z. B. gerichtete oder ungerichtete Graphen usw.). Mit diesen Tests wurde 75% Programmcodeüberdeckung bei den geprüften Klassen erreicht, wobei jedoch nicht alle portierten Klassen geprüft wurden.

Da die Prüfung der Codeabdeckung hauptsächlich zur Ergänzung der Testfälle des Unit-Tests dient, dieser jedoch nicht durchgeführt werden konnte, können die Ergebnisse nicht zur Korrektur von Fehlern eingesetzt werden. Sie lassen lediglich darauf schließen, dass viele Komponenten zumindest ausgeführt werden. Grundlegende Fehler im Zusammenspiel der Klassen liegen also offenbar nicht vor.

5.3. Beispielprogramm

Zur Demonstration der Funktionalität sollte ein Beispielprogramm entwickelt werden, das für einen gegebenen Graphen prüft, ob dieser ein gültiger Ereignis-Sequenz-Graph (ESG) ist. Hierzu sollen drei einfache Kriterien geprüft werden:

1. Es gibt genau einen Knoten ohne eingehende Kanten (Startknoten)

2. Es gibt genau einen Knoten ohne ausgehende Kanten (Endknoten)
3. Jeder Knoten ist vom Startknoten aus (indirekt) erreichbar

Daraus folgt, dass alle Knoten außer Start- und Endknoten sowohl eingehende als auch ausgehende Kanten haben müssen, sodass jeder Knoten sowohl besucht als auch verlassen werden kann. In einer Breiten- oder Tiefensuche kann anschließend festgestellt werden, ob tatsächlich alle Knoten erreichbar sind.

Bei der Implementierung des Beispielprogramms treten logische Fehler im Zusammenhang mit den Methoden

- `inDegreeOf()`
- `outDegreeOf()`
- `incomingEdgesOf()`
- `outgoingEdgesOf()`

auf. Diese Methoden dienen der Verarbeitung von Kanten eines Knotens und sind somit zwingend erforderlich für die Funktionalität des Beispielprogramms. Es zeigt sich, dass die `sets`, die zur Rückgabe der Kanten genutzt werden, keine Elemente enthalten, also offenbar nicht korrekt befüllt werden. Die Lokalisierung des eigentlichen Fehlers ist jedoch kaum möglich, da es sich um zahlreiche geschachtelte Methodenaufrufe innerhalb der inneren Klassen von `AbstractBaseGraph` handelt. Auch hier ist der Unit-Test erforderlich, um die einzelnen Methoden unabhängig voneinander testen und den Suchbereich eingrenzen zu können.

Da der Aufwand für eine manuelle Suche entsprechend hoch ist, konnte keine funktionierende Version des Beispielprogramms erstellt werden.

6. Ergebnisse

6.1. Schlussbetrachtung

Die generierte C++- Bibliothek ist imstande, einige Graphtypen zu erzeugen, beispielsweise gerichtete, gewichtete Graphen und Multigraphen, sowie ihnen Kanten und Knoten hinzuzufügen. Auf Grund des fehlenden Unit-Tests und den daher zurückgebliebenen Fehlern innerhalb der Bibliothek war es jedoch nicht möglich, wie geplant ein umfassendes Beispielprogramm zur Demonstration zu erstellen.

Alle Bestandteile der Bibliothek werden erfolgreich kompiliert, sodass lediglich logische Fehler vorliegen können. Offenbar liegen die Probleme hauptsächlich in den inneren Klassen von `AbstractBaseGraph`. Eine manuelle Fehlersuche gelingt auf Grund der Komplexität dieser Klasse jedoch nicht, sodass ein Unit-Test erforderlich ist, der insbesondere die einzelnen Methoden überprüft.

6.2. Zusammenfassung & Ausblick

Der wichtigste Schritt vor der weiteren Ausarbeitung der angelegten Bibliothek wäre, einen Unit-Test für das System zu entwickeln und damit die noch enthaltene Fehler zu finden. Als ein Test-Werkzeug kann beispielsweise das Unit-Test-Framework CppUnit eingesetzt werden. Mit CppUnit lassen sich automatisierte Tests implementieren. Es werden immer nur kleine Systemeinheiten abgearbeitet, beispielsweise einzelne Methoden von Klassen, für die Tests geschrieben und deren Ausgaben mit den zu erwartenden verglichen werden. Dadurch sind Fehler sofort lokalisiert. Zusätzlich werden die einzelne Tests zu größeren zusammengefasst, so dass sich auch komplette Softwaresysteme auf einmal prüfen lassen.

Im Anschluss könnte das ESG-Beispielprogramm weiterentwickelt werden, um einerseits zur Demonstration und andererseits als Grundlage für einen weiteren Codeabdeckungs-Test zu dienen.

Des Weiteren würde es sich anbieten, die Portierung fortzusetzen, beispielsweise um weitere Graph-Typen (Pseudograph, Subgraph), Exportfunktionen oder auch Algorithmen (zur "KürzestePfad"-Berechnung und Breiten- bzw. Tiefensuche). Diese sind in jGraphT bereits implementiert, stellen jedoch keine grundlegende Funktionalität dar und wurden deshalb in diesem ersten Schritt der Portierung nicht einbezogen.

A. Übersicht aller Aufgaben

Aufgabe	**Umsetzung durch**
Auswahl der zu portierenden Klassen	Manuel Webersen
Coding-Convenktions	Manuel Webersen
Doxygendokumentation	Galina Engelmann
Einrichtung Bugtracker	Manuel Webersen
Fehlerkorrektur	Manuel Webersen
Format für die Dokumentation	Galina Engelmann
GCov-Nutzung	Heinrich Drobin
Implementierung Beispielprogramm	Manuel Webersen
Organisatorisches und Zeitplan	Manuel Webersen
Portierung AbstractBaseGraph	Manuel Webersen
Portierung ArrayUninforcedSet	Manuel Webersen
Portierung ClassBasedEdgeFactory	Heinrich Drobin
Portierung ClassBasedVertexFactory	Heinrich Drobin
Portierung DefaultGraphs	Galina Engelmann
Portierung DirectedMultigraphs	Heinrich Drobin
Portierung EdgeFactory	Manuel Webersen
Portierung Edges	Manuel Webersen
Portierung EdgeSetFactory	Galina Engelmann
Portierung Generators	Manuel Webersen
Portierung Graph-Interfaces	Manuel Webersen
Portierung TypeUtil	Manuel Webersen
Portierung VertexFactory	Galina Engelmann
Testen der portierten Klassen mit Gcov	Heinrich Drobin
UML-Klassendiagramm	Galina Engelmann
Vervollständigung Dokumentation	Galina Engelmann

B. Elektronischer Anhang

Der Programmcode der in die C++- Programmiersprache portierten Klassen der jGraphT-Bibliothek sowie die Doxygen-Dokumentation sind der beiliegenden CD zu entnehmen.

Literaturverzeichnis

[1] Jonathan L. Gross, Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2004

[2] Wikipedia Graphentheorie:
http://de.wikipedia.org/wiki/Graphentheorie

[3] Wikipedia Graph:
http://de.wikipedia.org/wiki/Graph_(Graphentheorie)

[4] Gewichteter Graph:
http://upload.wikimedia.org/wikiversity/de/0/00/%C3%9Cberschrift.jpg

[5] JgraphT-Bibliothek:
http://www.jgrapht.org/javadoc/index.html?index-all.html

[6] Mercurial:
http://mercurial.selenic.com/

[7] Mercurial `Tutorial`:
http://mercurial.selenic.com/wiki/GermanTutorialFirstChange

[8] TortoiseHg:
http://tortoisehg.bitbucket.org/download/index.html

[9] Mantis:
http://www.mantisbt.org/

[10] Gcov-Testwerkzeug:
http://sourceforge.jp/projects/ginkgo/releases/

[11] Doxygen-Dokumentationstool:
http://www.doxygen.org