23. Graphen in Java

1

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

Technische Universität Dresden

Version 15-1.2, 01.06.15

1) Entwurfsmuster Fabrikmethode
2) Das Graph Framework JGraphT
    1) Struktur
    2) Iteratoren
    3) Kürzeste Pfade
    4) Generatore

Obligatorische Literatur

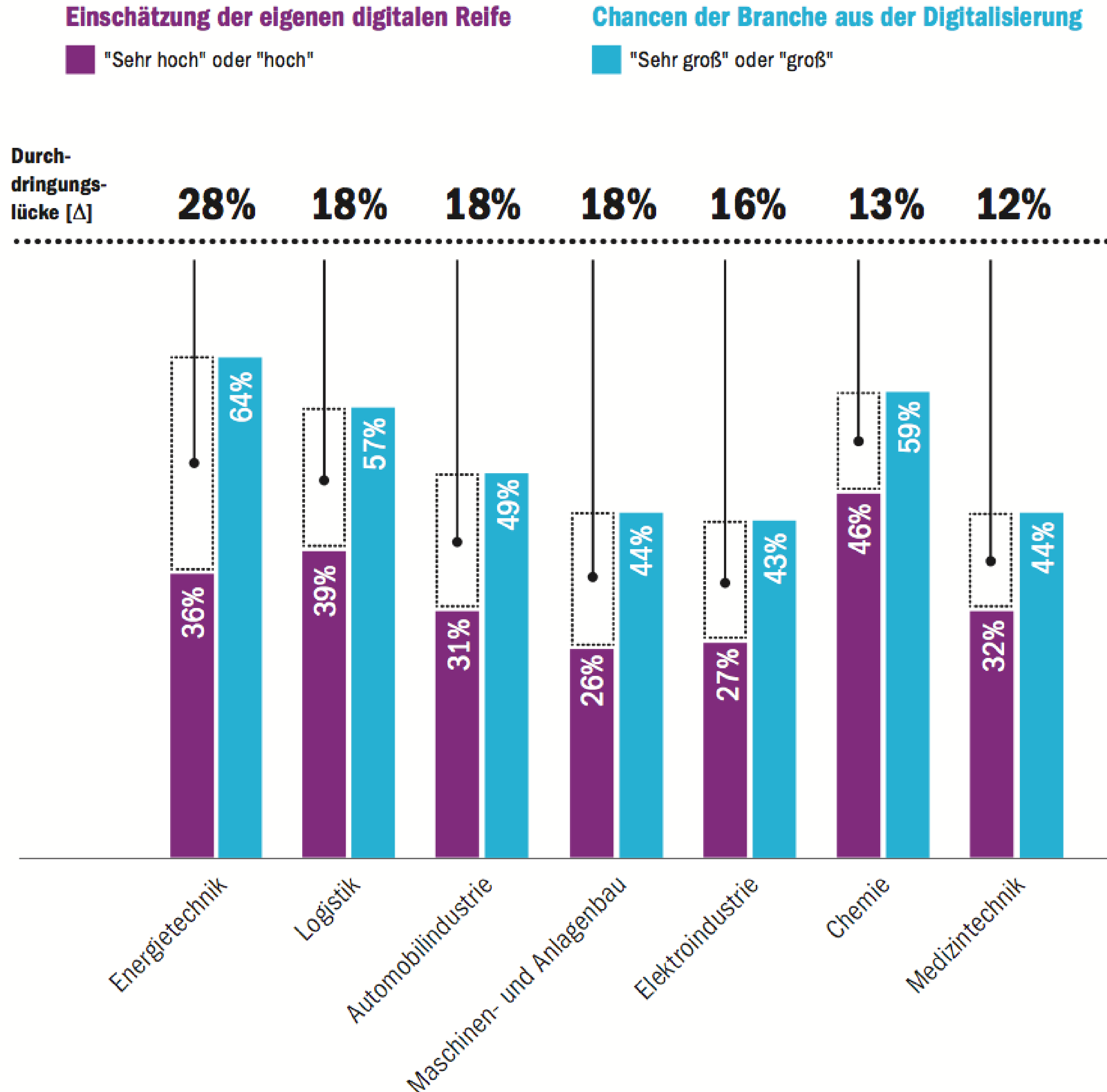
- JDK Tutorial für J2SE oder J2EE, www.java.sun.com
- Dokumentation der Jgrapht library http://www.jgrapht.org/
  - Javadoc http://www.jgrapht.org/javadoc
  - http://sourceforge.net/apps/mediawiki/jgrapht/index.php?title=jgrapht:Docs
- Dokumentation der Library für verteilte Graphen GELLY (Teil von Apache Flink)
  - http://ci.apache.org/projects/flink/flink-docs-master/gelly_guide.html

Nicht-obligatorische Literatur

- [HB01] Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In Jan Bosch, editor, GCSE, volume 2186 of Lecture Notes in Computer Science, pages 10-23. Springer, 2001.
  - Facetten von Graphen und wie man sie systematisch, noch besser in einem Framework anordnet
  - Siehe Vorlesung “Design Patterns and Frameworks”

Ziele

- Eine komplexe Java-Bibliothek aus dritter Hand, eine Graphen-Bibliothek, kennenlernen
- Graphen als spezielle Kollaborationen verstehen
- Fabrikmethoden, Iteratoren und Streams in der Anwendung bei Graphen
- Generische Graphalgorithmen kennenlernen
  - Delegatoren
  - Generatoren
  - Graphanalysen



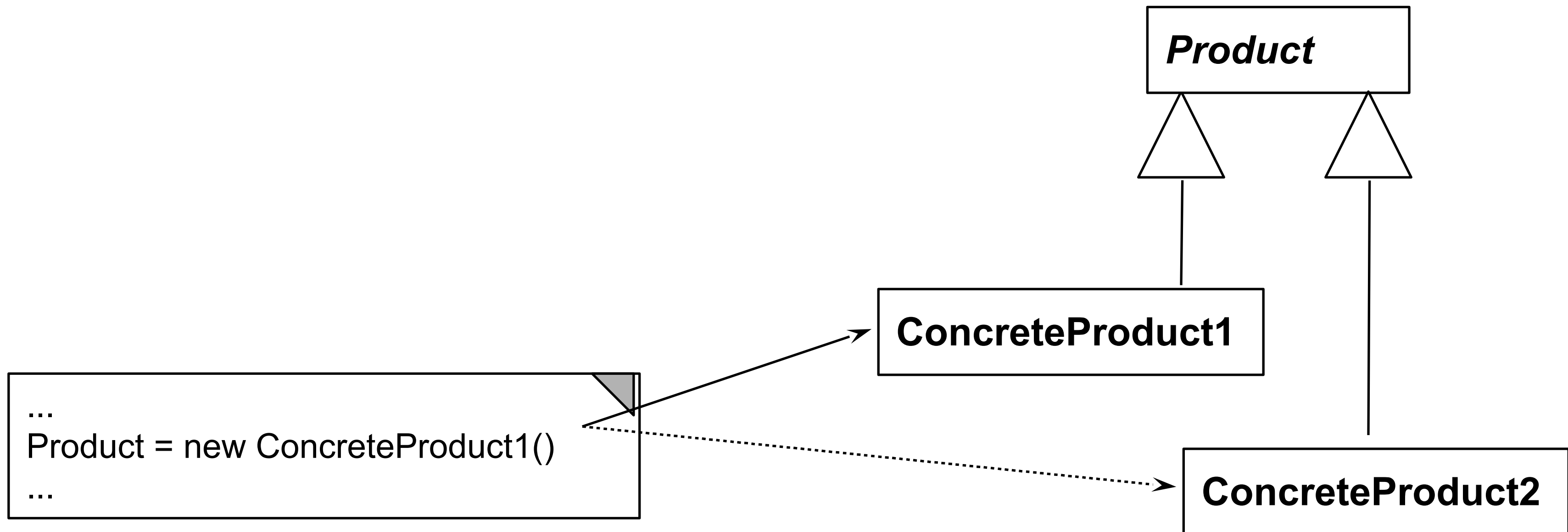
http://www.rolandberger.de/media/pdf/Roland_Berger_Die_digitale_Transformation_der_Industrie_20150315.pdf

Foto: Detlef W. Schmalow / BASF

zur polymorphen Variation von Komponenten (Produkten) und zum Verbergen von Produkt-Arten

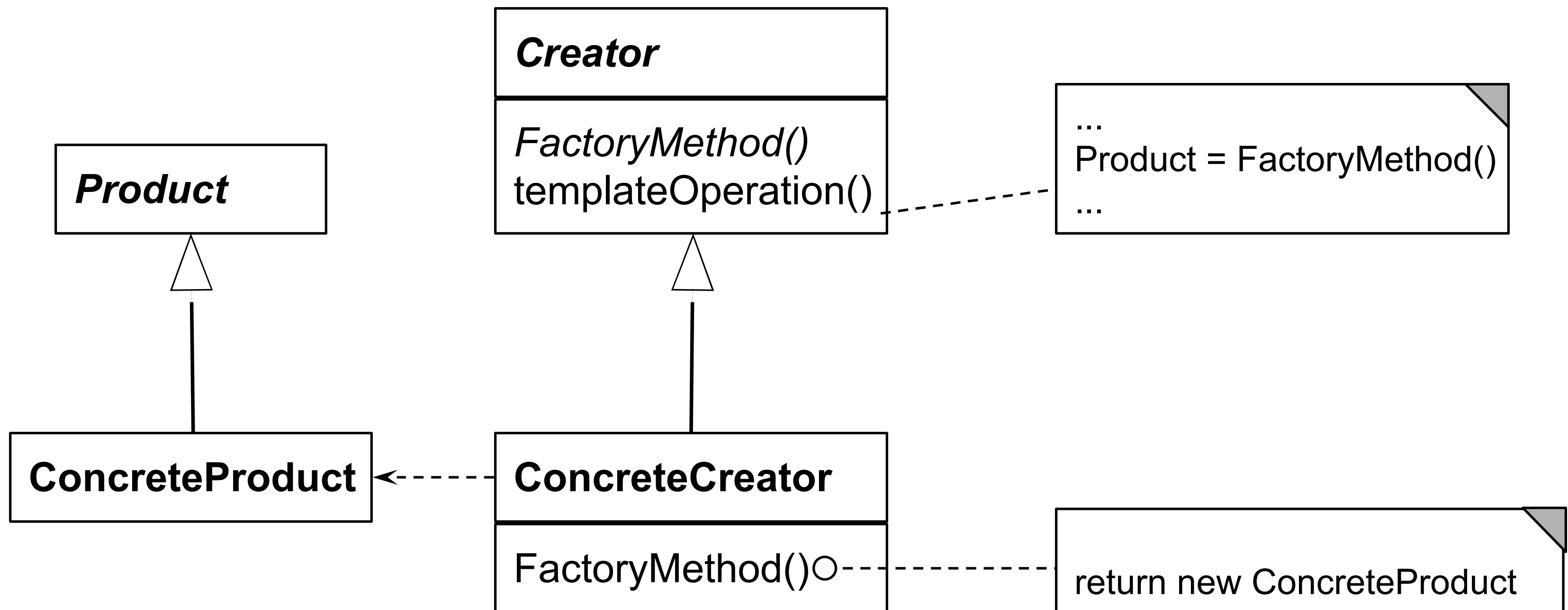
Problem der Fabrikmethode

- Wie variiert man die Erzeugung für eine polymorphe Hierarchie von Produkten?
- Problem: Konstruktoren sind nicht polymorph!



Struktur Fabrikmethode

- FactoryMethod ist eine Variante von TemplateMethod, zur Produkterzeugung [Gamma95]



Fabrikmethode (Factory Method)

- Allokatoren in einer abstrakten Oberklasse nennt man *Fabrikmethoden (polymorphe Konstruktoren)*
  - Konkrete Unterklassen spezialisieren den Allokator
  - Template-Methoden rufen die Fabrikmethode auf

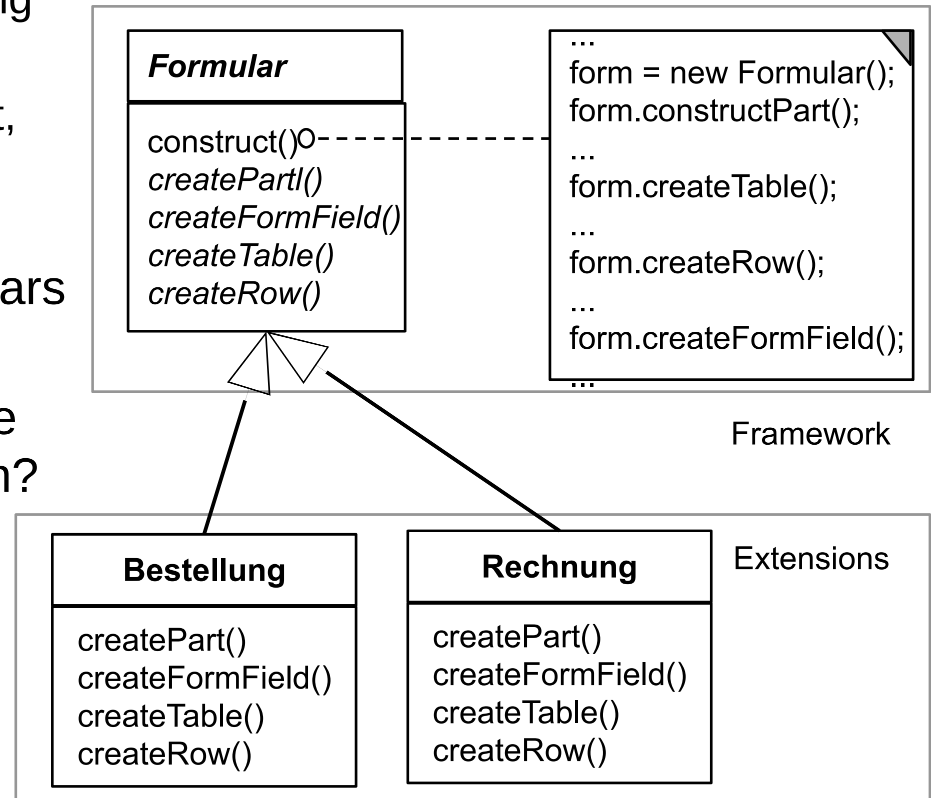
```
public class Client {
  ...
  Creator cr = new ConcreteCreator();
  cr.collect();

}
```

```
// Abstract creator class
public abstract class Creator  {
  public void collect() {
    Set mySet = createSet(10);
    Set secondSet = createSet(20);
    ....
  }
  // factory method
  public abstract Set createSet(int n);
}
```

```
// Concrete creator class
public class ConcreteCreator
    extends Creator {
   public Set createSet(int n) {
      return new ListBasedSet(n);
   }
  ...
}
```

Beispiel FactoryMethod für Formulare

- Framework (Rahmenwerk) für Formulare
  - Klasse Formular hat eine Schablonenmethode zur Planung der Struktur von Formularen
  - Abstrakte Methoden: createPart, createFormField, createTable, createRow
- Benutzer können Art des Formulars verfeinern
- Wie kann das Rahmenwerk neue Arten von Formularen behandeln?



Lösung mit FactoryMethod

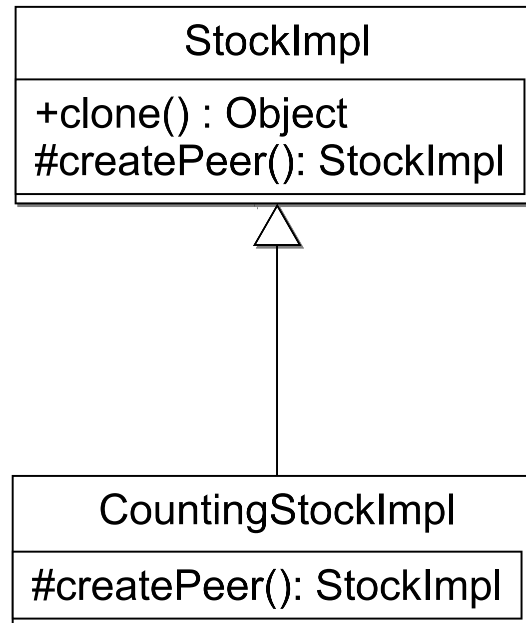
- Bilde createFormular() als Fabrikmethode aus

```
// abstract creator class
public abstract class Formular  {
   public abstract
       Formular createFormular();
   ...
}
```

```
// concrete creator class
public class Bestellung extends Formular {
   Bestellung() {
       ...
   }
   public Formular createFormular() {
      ... fill in more info ...
      return new Bestellung();
  }
   ...
}
```

Factory Method im SalesPoint-Rahmenwerk

- Anwender von SalesPoint verfeinern die StockImpl-Klasse, die ein Produkt des Warenhauses im Lager repräsentiert
  - z.B. mit einem CountingStockImpl, der weiß, wieviele Produkte noch da sind



Einsatz in Komponentenarchitekturen

- In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von oberen Schichten (Anwendungsschichten) aus die Rahmenwerkschicht zu konfigurieren:

Rahmenwerk

Formular
create():Formular

Formular Stil
create():FormularStil

Anwendung

Lieferschein

Bestellung

Rechnung

Moderner Stil

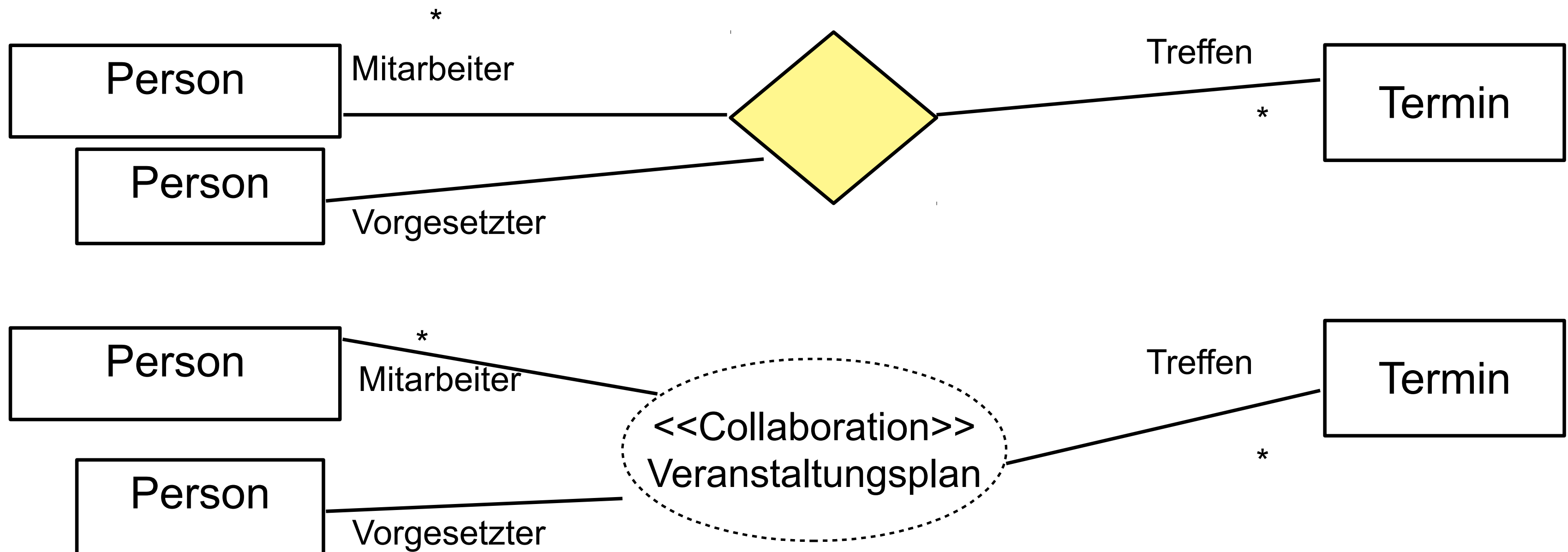
Klassischer Stil

23.2 Das JGraphT Framework

Fabriken überall

Graphen als spezielle Kollaborationen

- Eine nicht-fixe **Assoziation** oder **Relation** besteht aus einer dynamisch wachsenden Tabelle mit einer Menge von Tupeln
  - Ein **Graph** verknüpft zwei Mengen von Objekten (Knotenmengen) mit einer Assoziation und bietet Navigationsverhalten an
  - Ein **Hypergraph** verknüpft mehrere Knotenmengen mit einer n-stelligen Relation
- Graphen bilden spezielle binäre Kollaborationen; Hypergraphen spezielle n-stellige Kollaborationen, die Navigationen als Verhalten anbieten



Ziele einer Graph-Bibliothek

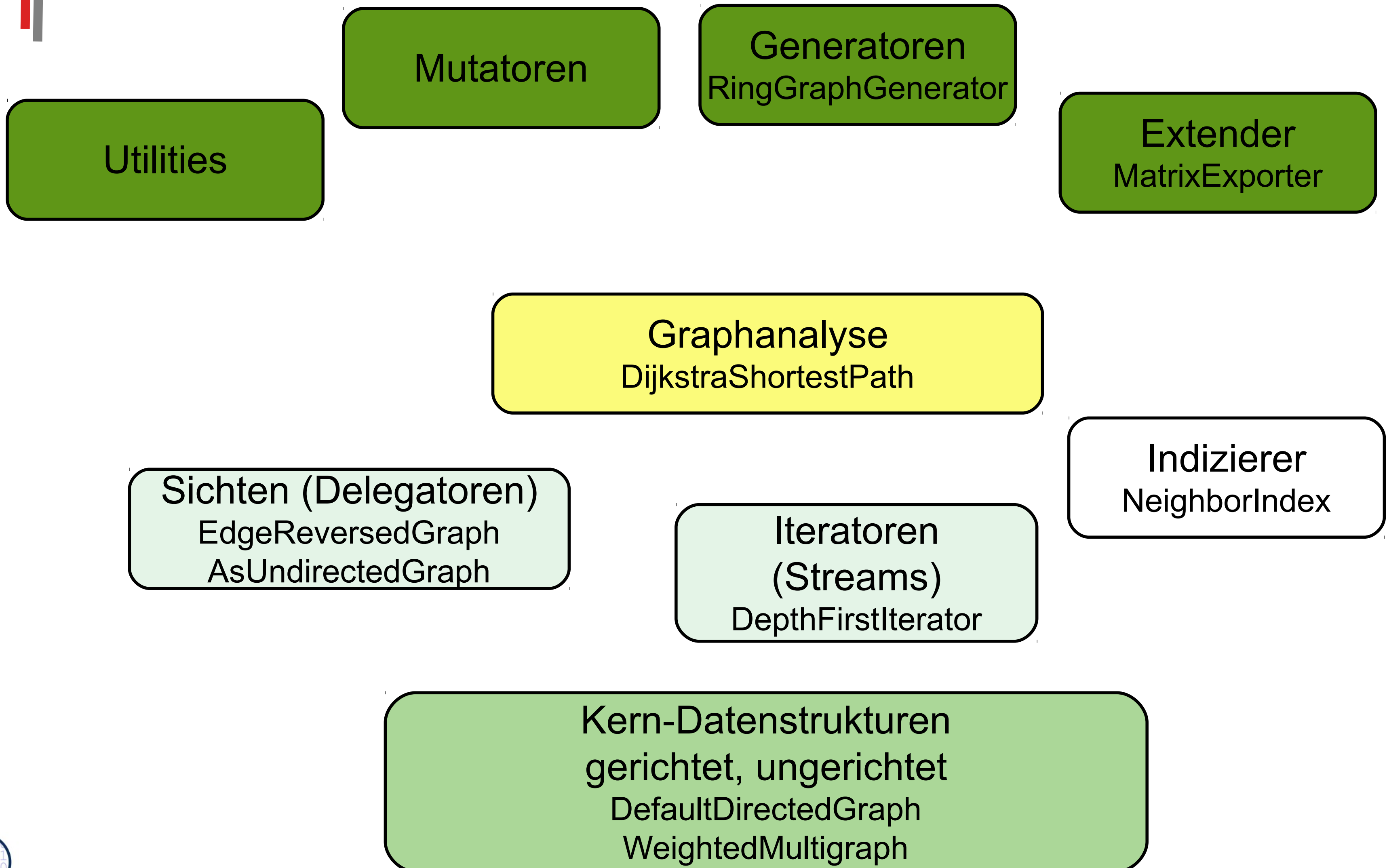
- In Java können Graphen durch ein Framework dargestellt werden
  - [JGraphT] stellt eine Bibliothek mit einer einfachen Abstraktion von Graphen dar
  - Für Graphen auf Objekten, XML Objekten, URLs, Strings, Graphen …
  - Fabrikmethoden, Generics und Iteratoren werden genutzt
- Unterscheidung von speziellen Formen von Graphen
- Sichten auf Graphen
- Generische Algorithmen auf Graphen

Klassifikationsfacetten von Graphen

Multiple Edges	Direction	Cyclicity	Weight
Multiple Edges Unique Edges	Directed Bidirectional	Cyclic Cycle graph (hamiltonian) Acyclic	

Kategorien von Graphalgorithmen in JGraphT

- mit Beispielen



Graphen dienen dem Aufbau von Objektnetzen

Allokation eines Graphen spezifischer Kategorie

Knoten addieren

Kanten addieren

Sichten einziehen (mit Delegatoren)

Graphanalyse (mit Analyse-Kommandoobjekten)

<<interface>> DirectedGraph<V,E>

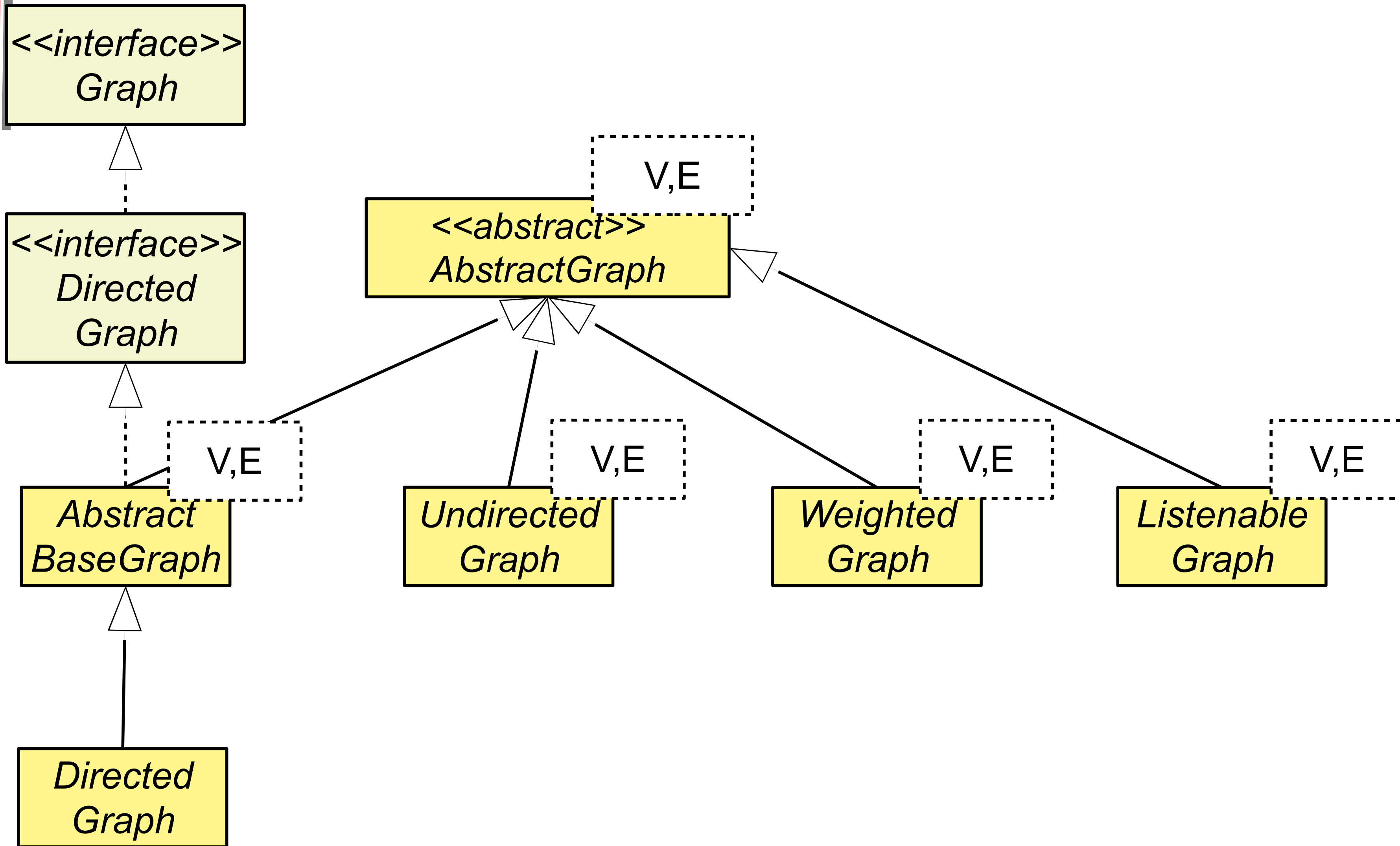
```
// Query-Methoden
 java.util.Set<E>   edgeSet()
 java.util.Set<V>   vertexSet()
 java.util.Set<E>   edgesOf(V vertex)
          // Returns a set of all edges touching the specified vertex.
 java.util.Set<E>   getAllEdges(V sourceVertex, V targetVertex)
 E        getEdge(V sourceVertex, V targetVertex)
   // Returns an edge connecting source vertex to target vertex if such vertices
   // and such edge exist in this graph.
 EdgeFactory<V,E>   getEdgeFactory()
 V        getEdgeSource(E e)
 V        getEdgeTarget(E e)
 double   getEdgeWeight(E e)
// Check-Methoden
 boolean  containsEdge(E e)
 boolean  containsEdge(V sourceVertex, V targetVertex)
 boolean  containsVertex(V v)
// Modifikatoren
 E        addEdge(V sourceVertex, V targetVertex)
 boolean  addVertex(V v)
 boolean  removeAllEdges(java.util.Collection<? extends E> edges)
          // Removes all the edges in this graph that are also contained in the
          // specified edge collection.
 java.util.Set<E>   removeAllEdges(V sourceVertex, V targetVertex)
 boolean  removeAllVertices(java.util.Collection<? extends V> vertices)
          // Removes all the vertices in this graph that are also contained in the
          // specified vertex collection.
 boolean  removeEdge(E e)
 E        removeEdge(V sourceVertex, V targetVertex)
         // Removes an edge going from source vertex to target vertex, if such vertices
         // and such edge exist in this graph.
 boolean  removeVertex(V v)
```

DirectedGraph.java in JGraphT

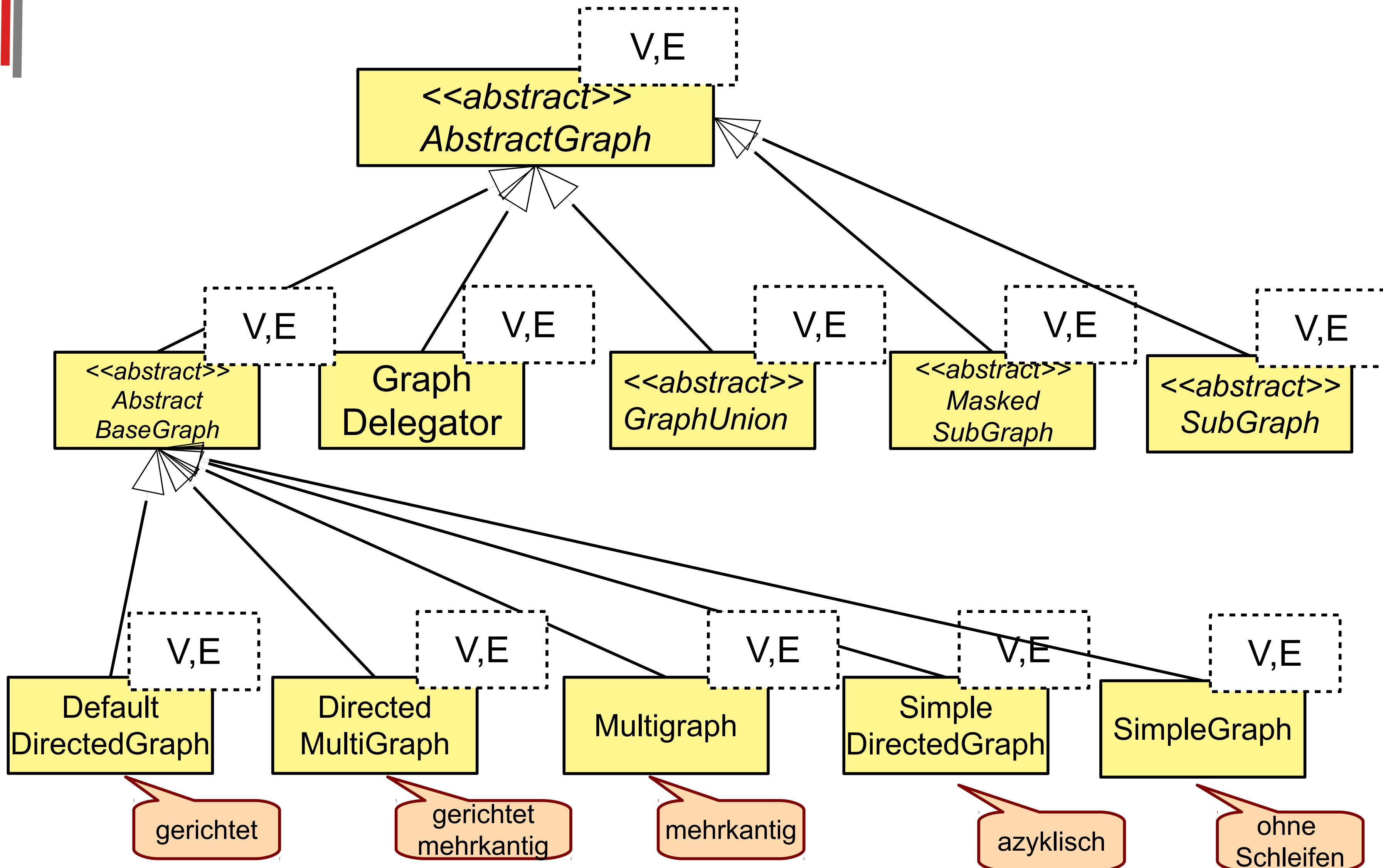
DirectedGraph<V,E>

```
// Constructors (doesnt use a factory)
DefaultDirectedGraph(java.lang.Class<? extends E> edgeClass)
        //  Creates a new directed graph.
DefaultDirectedGraph(EdgeFactory<V,E> ef)
        //  Creates a new directed graph with the specified edge factory.
// Query methods
java.util.Set<E>  incomingEdgesOf(V vertex)
        //  Returns a set of all edges incoming into the specified vertex.
 int     inDegreeOf(V vertex)
        //  Returns the "in degree" of the specified vertex.
 int     outDegreeOf(V vertex)
        //  Returns the "out degree" of the specified vertex.
 java.util.Set<E> outgoingEdgesOf(V vertex)
        //  Returns a set of all edges outgoing from the specified vertex.
```

Die Klassenhierarchie Graph

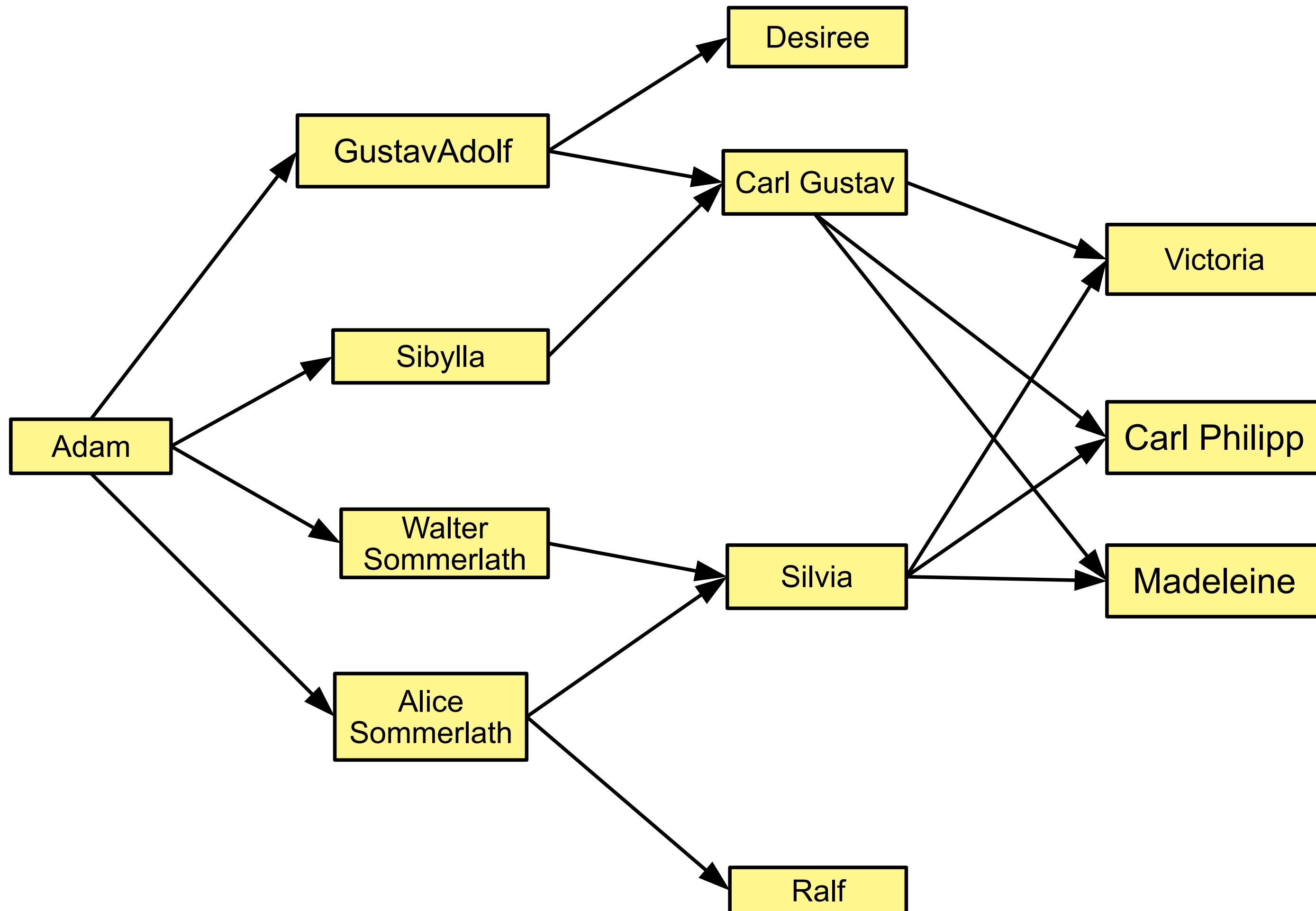


Die Implementierungshierarchie Graph



Beispiel: Verwandtschaftsbeziehungen

- Familienbeziehungen sind immer azyklisch
- Die schwedische Königsfamilie:



Aufbau gerichteter Graphen

SwedishFamilyDemo.java


```
// SwedishKingFamilyDemo.java
//
// constructs a directed graph with
// the specified vertices and edges
DirectedGraph<String, DefaultEdge> parentOf =
  new DefaultDirectedGraph<String, DefaultEdge>
     (DefaultEdge.class);
String adam = "Adam";
String victoria = "Victoria";
String madeleine = "Madeleine";
parentOf.addVertex(adam);
parentOf.addVertex("Eve");
parentOf.addVertex("Sibylla");
parentOf.addVertex("Gustav Adolf");
parentOf.addVertex("Alice Sommerlath");
parentOf.addVertex("Walter Sommerlath");
parentOf.addVertex("Sylvia");
parentOf.addVertex("Ralf");
parentOf.addVertex("Carl Gustav");
parentOf.addVertex("Desiree");
parentOf.addVertex(victoria);
parentOf.addVertex("Carl Philipp");
parentOf.addVertex(madeleine);
```

```
// add edges
parentOf.addEdge("Adam", "Gustav Adolf");
parentOf.addEdge("Adam", "Sibylla");
parentOf.addEdge("Adam", "Walter Sommerlath");
parentOf.addEdge("Adam", "Alice Sommerlath");
parentOf.addEdge("Walter Sommerlath","Sylvia");
parentOf.addEdge("Alice Sommerlath","Sylvia");
parentOf.addEdge("Walter Sommerlath","Ralf");
parentOf.addEdge("Alice Sommerlath","Ralf");
parentOf.addEdge("Gustav Adolf", "Carl Gustav");
parentOf.addEdge("Sibylla", "Carl Gustav");
parentOf.addEdge("Gustav Adolf", "Desiree");
parentOf.addEdge("Sibylla", "Desiree");
parentOf.addEdge("Carl Gustav", "Victoria");
parentOf.addEdge("Carl Gustav", "Carl Philipp");
parentOf.addEdge("Carl Gustav", "Madeleine");
parentOf.addEdge("Sylvia", "Victoria");
parentOf.addEdge("Sylvia", "Carl Philipp");
parentOf.addEdge("Sylvia", "Madeleine");
/* 1 */ // parentOf.addEdge(victoria,adam);
```

Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Funktion darstellt (reifiziert).

- **Funktionalobjekte** kapseln Berechnungen und können sie später ausführen (laziness)
  - Es gibt eine Standard-Funktion in der Klasse des Funktionalobjektes, das die Berechnung ausführt (Standard-Name, z.B. *execute()* oder *doIt()*)
  - Zur Laufzeit kann man das Funktionalobjekt mit Parametern versehen, herumreichen, und zum Schluss ausführen
  - Funktionalität wie `undo(), redo(), persist()` kann vorhanden sein

```
// A functional object captures a method
public abstract class Command {
   void execute();
   void undo();
}
public class Painter extend Command {
   void redo();
}
```

Konsistenzprüfung und Navigation

- Die meisten generischen Algorithmen von jgrapht sind Funktionalobjekte (Entwurfsmuster Command, s. Kap. “Collections”)
- `CycleDetector.findCycles()` findet Zyklen im Graphen, jenseits von Selbstkanten
  - Entspricht `execute()`

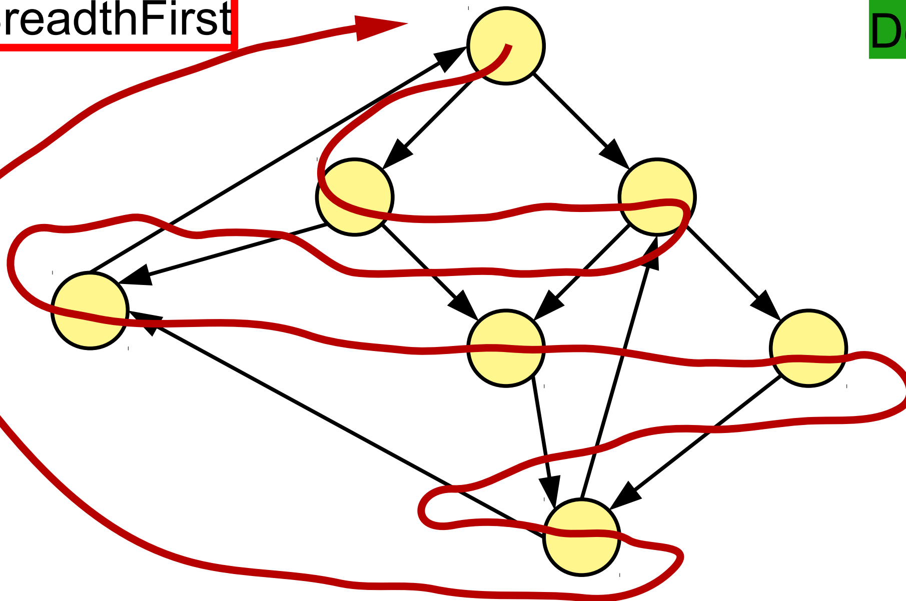
```
// (a) cycle detection in graph parentOf
CycleDetector<String, DefaultEdge> cycleDetector =
    new CycleDetector<String, DefaultEdge>(parentOf);

Set<String> cycleVertices = cycleDetector.findCycles();
System.out.println("Cycle: "+cycleVertices.toString());
```

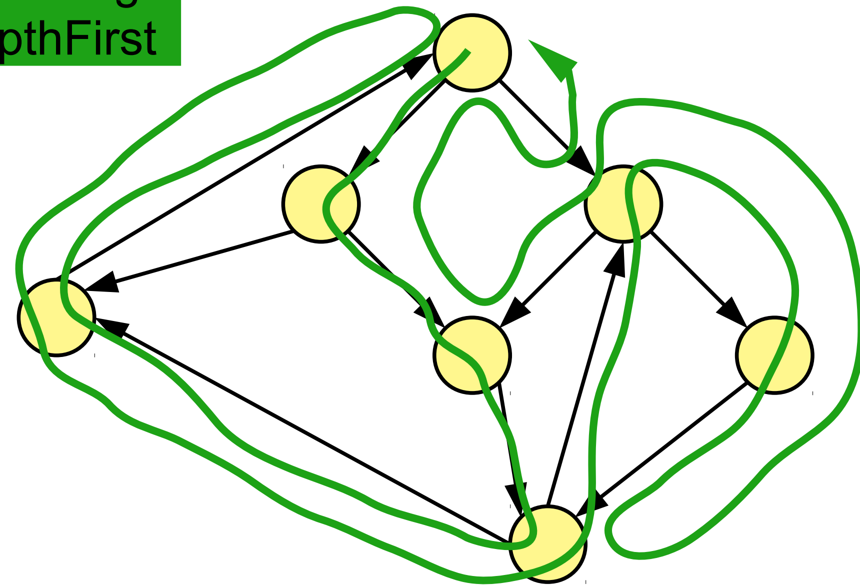
23.2.2 Iteratoren laufen Graphen ab

- Man kann mit einem Graphiterator den Graphen ablaufen und seine Knoten ausgeben, ohne seine Struktur zu kennen

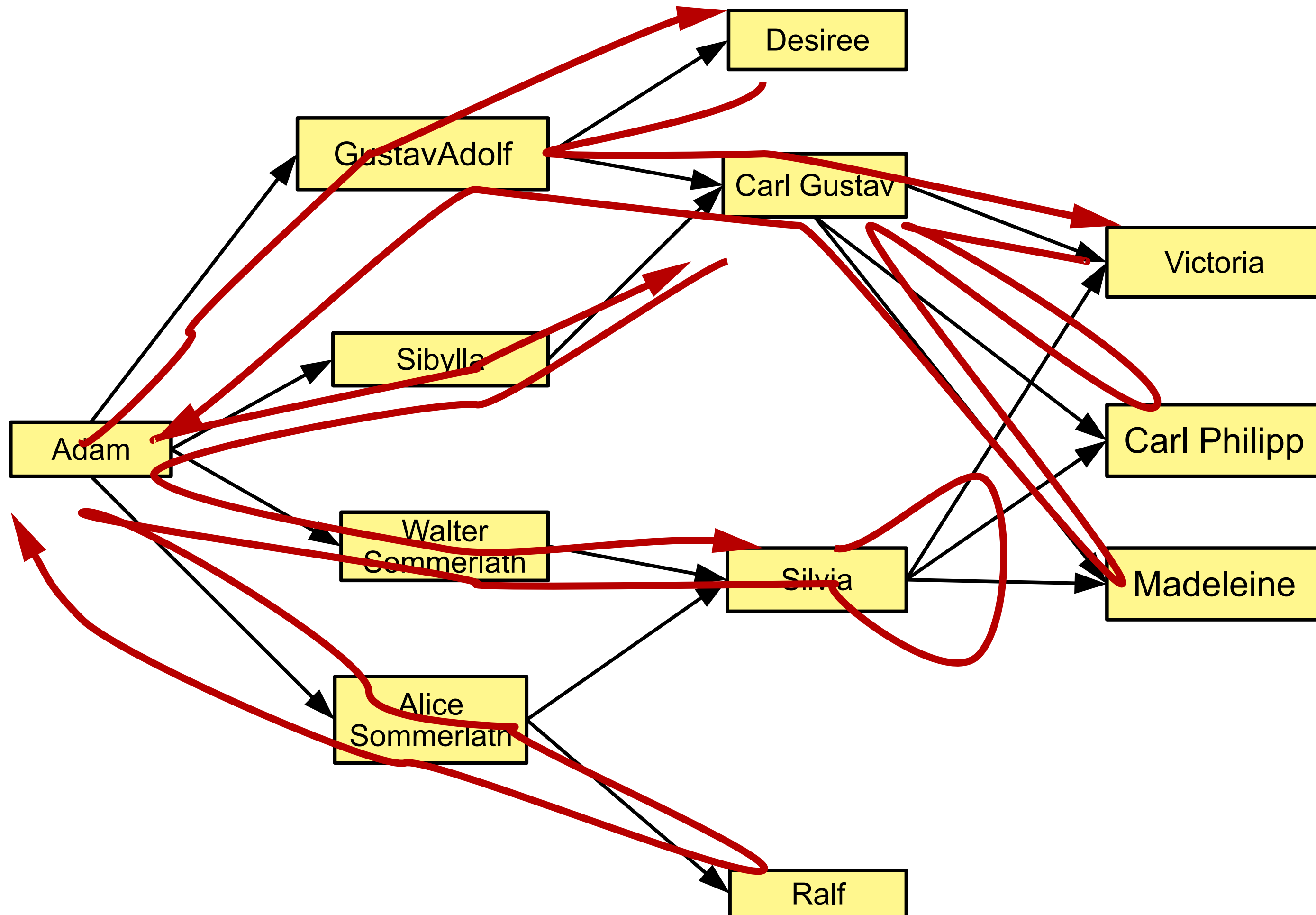
Arten von Durchläufen mit Iteratoren

- `BreadthFirstIterator` läuft über den Graphen in Breitensuche, sozusagen “Schicht für Schicht”, und gibt die Knoten aus
- `DepthFirstIterator` läuft über den Graphen in Tiefensuche, sozusagen “Pfad für Pfad”





Tiefensuche auf dem azykl. Graphen der Königsfamilie



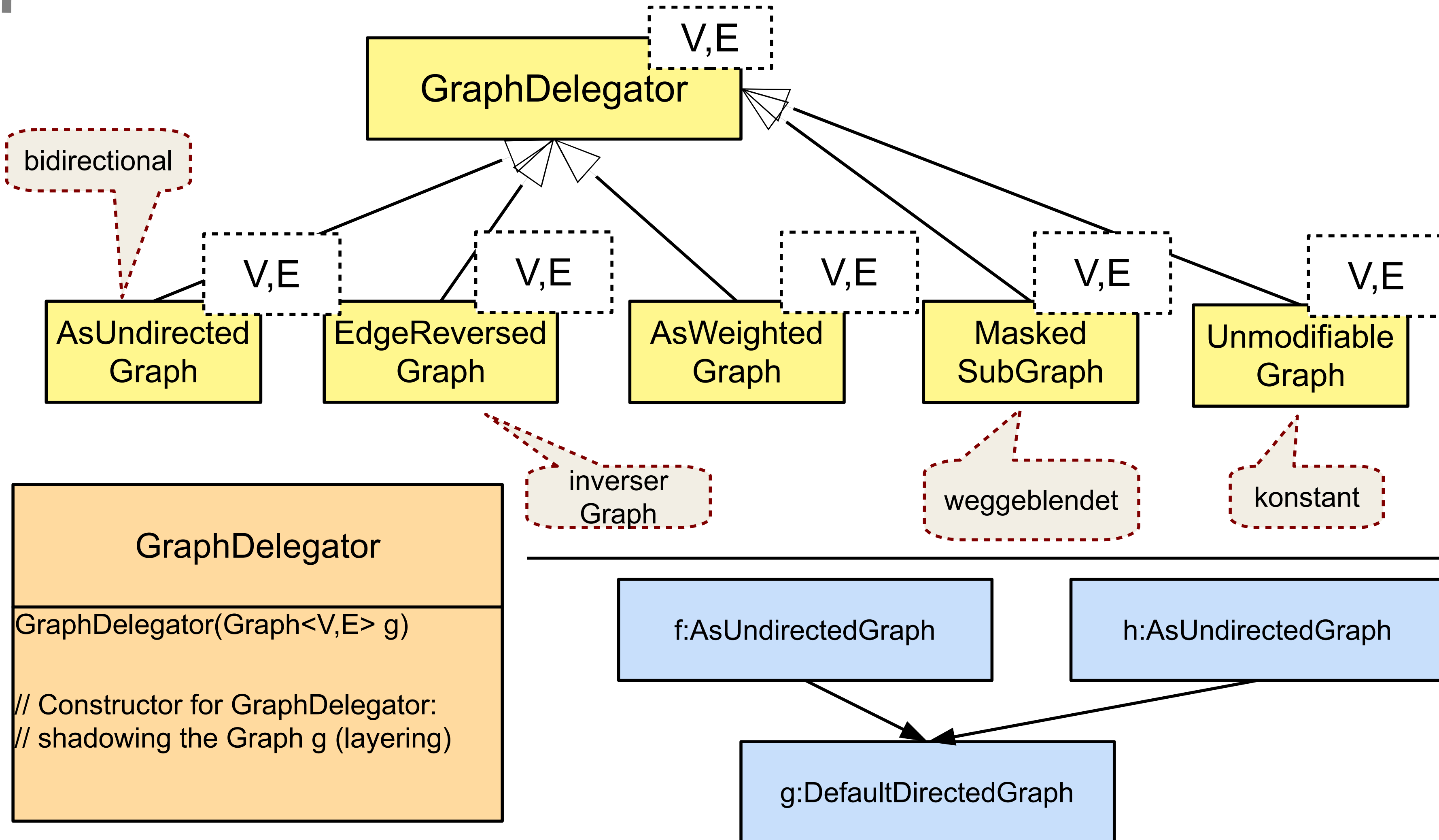
Pulling Iterators


```
// (b) depth-first iteration in graph parentOf

System.out.println("breadth first enumeration: ");
DepthFirstIterator<String,DefaultEdge> dfi =
     new DepthFirstIterator<String, DefaultEdge>(parentOf);

for (String node = dfi.next(); dfi.hasNext(); node = dfi.next()) {
    System.out.println("node: "+node);
}
```

```
// (bc) breadth-first iteration in graph parentOf

System.out.println("breadth first enumeration: ");
BreadthFirstIterator<String,DefaultEdge> bfi =
     new BreadthFirstIterator<String, DefaultEdge>(parentOf);

for (String node = bfi.next(); bfi.hasNext(); node = bfi.next()) {
    System.out.println("node: "+node);
}
```

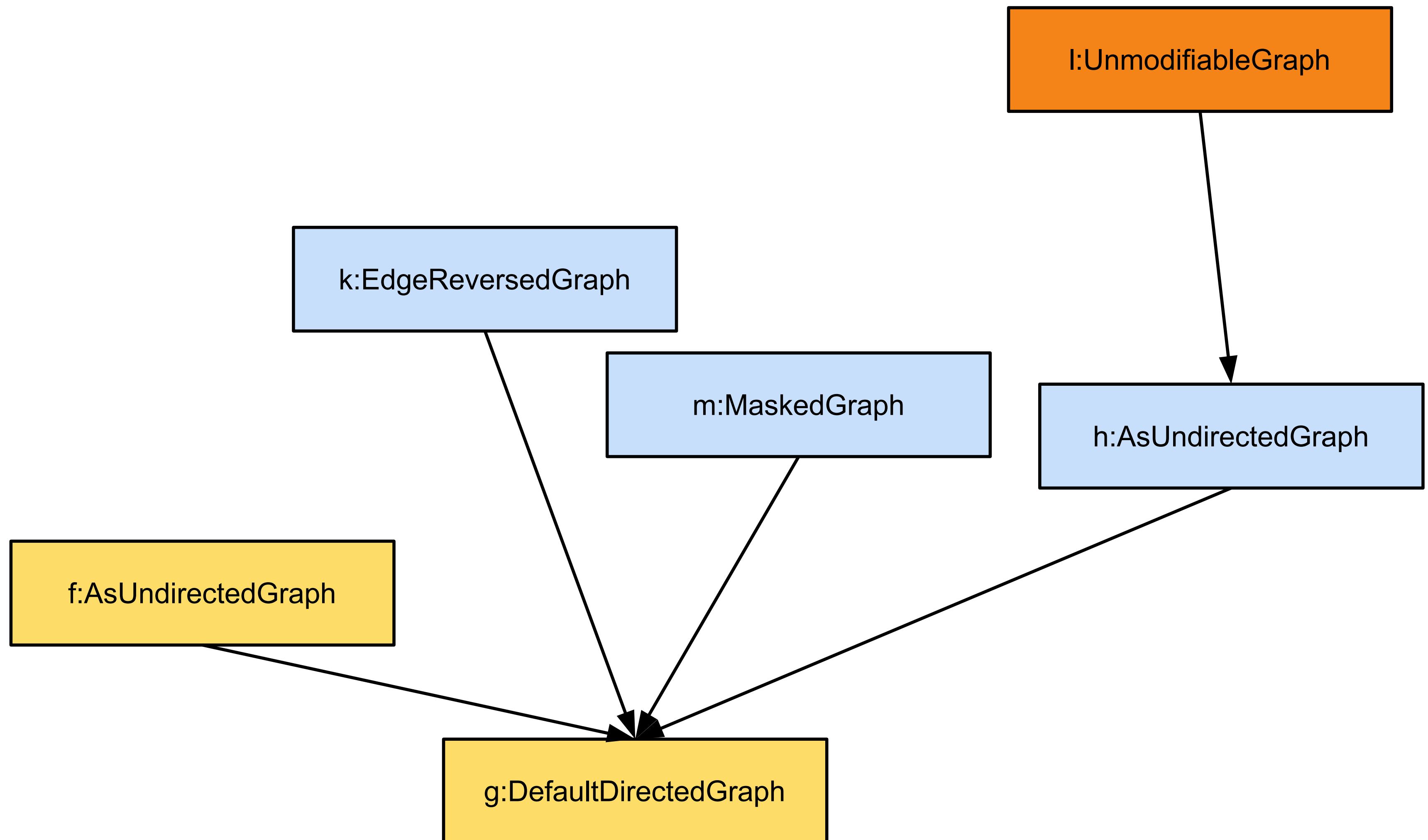
Delegatoren erzeugen Sichten

- Ein **Delegator** ist ein Objekt, das einen Graphen “vorspiegelt” und auf einen Basisgraphen anderen Typs zurückführt.



Schichtung von Graphen mit Delegatoren (Layering of Graphs)

- Was sieht ein Aufrufer (client) eines spezifischen Graphs?



23.2.3 Gewichtete Graphen Finden kürzester Pfade

- Dijkstra's Algorithmus findet zwischen 2 Knoten den kürzesten Pfad
- Ein **Pfadobjekt** stellt einen Pfad in einem Graphen dar. Ein Pfadobjekt ist ein Delegator auf einen anderen Graphen (Sicht).
- **DijkstraShortestPath** bildet den kürzesten Pfad in einem ungerichteten Grpahen ab.


```
// (c) Shortest path with Dijkstra's method
DijkstraShortestPath<String,DefaultEdge> descendantPath
    = new DijkstraShortestPath(parentOf,adam,victoria);
System.out.println("shortest path between Adam and Victoria ("
    +descendantPath.getPathLength()+"):");

GraphPath<String,DefaultEdge> path = descendantPath.getPath();

// Hint: Graphs is an algorithm class (helper class)
List<String> nodeList = Graphs.getPathVertexList(path);
for (String node : nodeList) {
    System.out.println("node: "+node);
}
```

Finden kürzester Pfade im ungerichteten Graphen (Sicht)

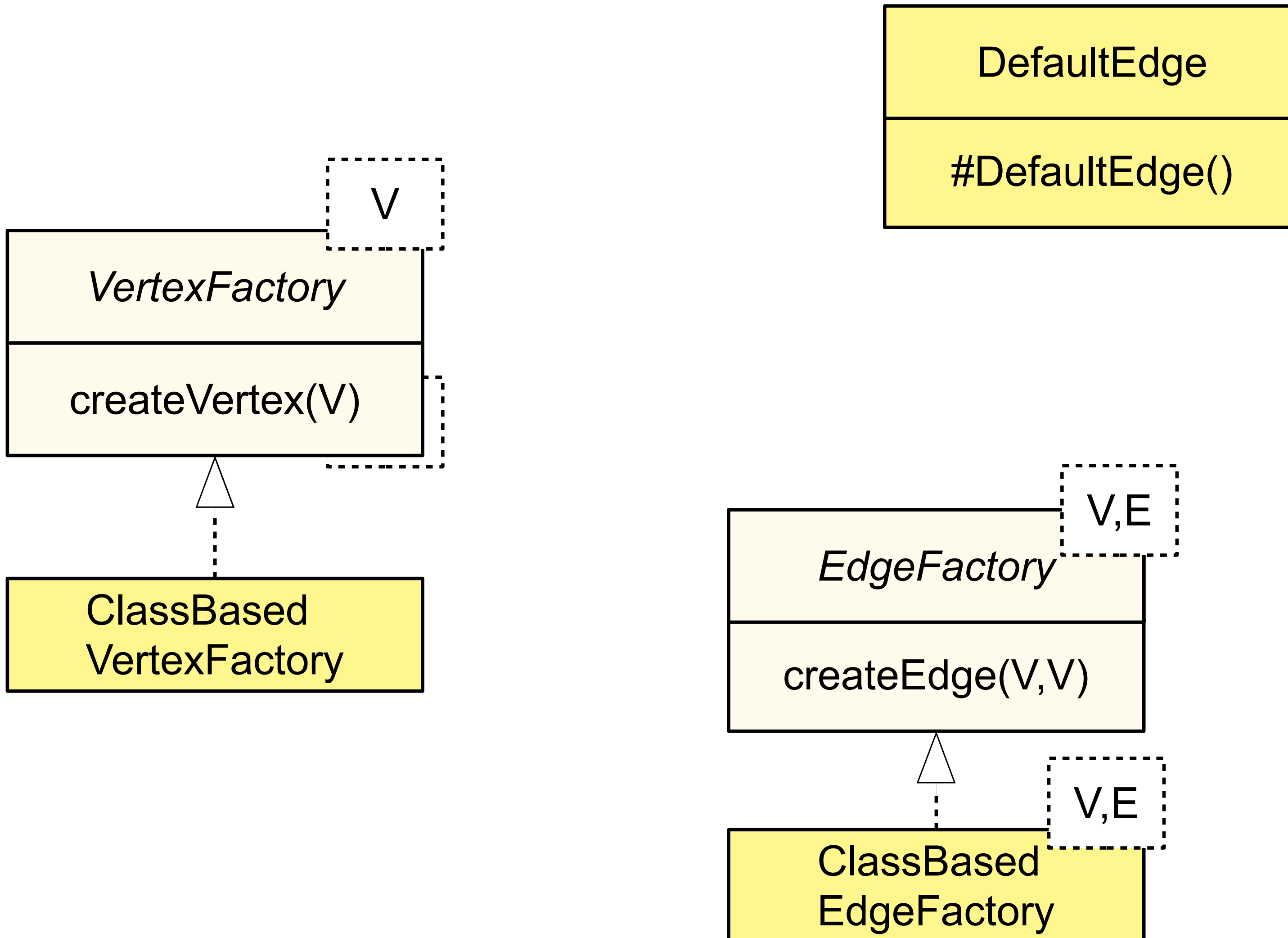
- Ein **Pfadobjekt** stellt einen Pfad in einem Graphen dar. Ein Pfadobjekt ist ein Delegator auf einen anderen Graphen (Sicht).
- **DijkstraShortestPath** bildet den kürzesten Pfad in einem ungerichteten Grpahen ab.

```
// Now interpret the directed graph as undirected
AsUndirectedGraph<String,DefaultEdge> descendantOrAscendant = new AsUndirectedGraph(parentOf);
System.out.println("related graph: "+descendantOrAscendant.toString());

// Shortest path with Dijkstra's method in the undirected graph
DijkstraShortestPath<String,DefaultEdge> ancestorPath
    = new DijkstraShortestPath(descendantOrAscendant,madeleine,adam);

System.out.println("shortest path between Madeleine and Adam ("+ancestorPath.getPathLength()+"):");

GraphPath<String,DefaultEdge> path = ancestorPath.getPath();

nodeList = Graphs.getPathVertexList(path);
for (String node : nodeList) {
    System.out.println("node: "+node);
}
```

Weitere Analyseklassen

- `BellmanFordShortestPath` findet kürzeste Wege in gewichteten Graphen
  - Berühmter Algorithmus zum Berechnen von Wegen in Netzen
  - www.bahn.de
  - Logistik, Handlungsreisende, etc.
  - Optimierung von Problemen mit Gewichten
- `StrongConnectivityInspector` liefert "Zusammenhangsbereiche", starke Zusammenhangskomponenten, des Graphen
  - In einem Zusammenhangsbereich sind alle Knoten gegenseitig erreichbar
- u.v.m.

Fabrikmethoden für Knoten und Kanten



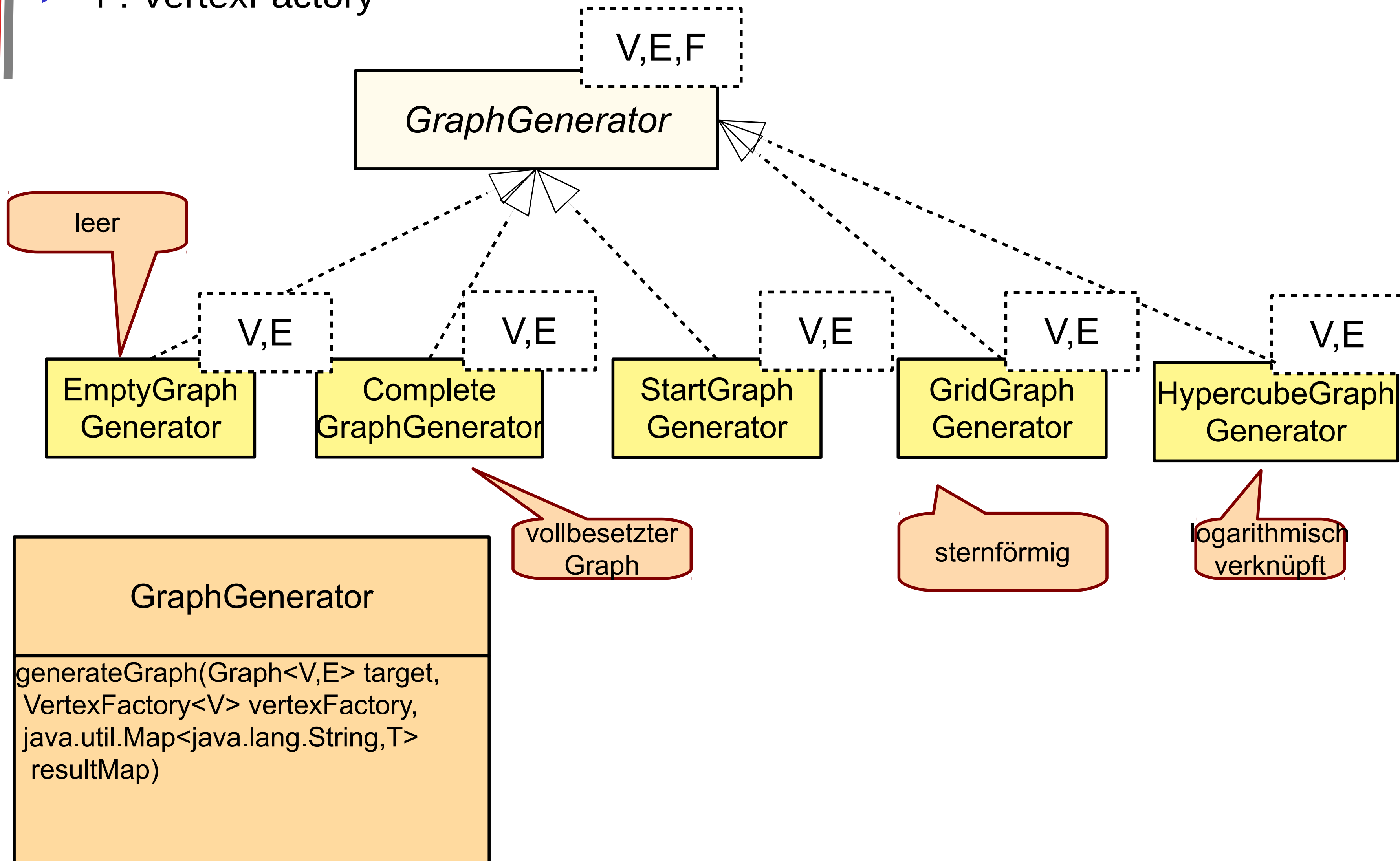
23.2.4 Generatoren

- Neue Graphen mit anderen Strukturen können aus einem bestehenden Graphen heraus erzeugt werden

GraphGenerator
generateGraph(Graph<V,E> target, VertexFactory<V> vertexFactory, java.util.Map<java.lang.String,T> resultMap)

Generatoren erzeugen verschiedene Arten von Graphen

- F: VertexFactory



23.E.1 Lern-Exkurs: Die Test-Suite von JGraphT

- Auf der Webseite finden Sie unter

`JGraphT-Examples/JGraphT-JUnit-3-8-Tests`

- die Test-Suite von JGraphT (freie Lizenz GPL), die auf JUnit-3.8 basiert.
- Welche Datei enthält eine Zusammenstellung aller Tests in eine Suite?
- Inspizieren Sie die Datei `SimpleDirectedGraphTest.java`:
  - Welche Testfälle können Sie identifizieren?
  - Welche Teile der Funktionalität von SimpleDirectedGraph sind gut, welche nicht gut abgedeckt, d.h. mit Testfällen versehen worden?
- Würden Sie JGraphT als *Software* oder nur als *Programm* bezeichnen?

23.E.2 Lern-Exkurs: Die Library GELLY

- Analysieren Sie die Webseite von GELLY
  - http://gellyschool.com/
  - http://ci.apache.org/projects/flink/flink-docs-master/gelly_guide.html
- Welche Unterschiede gibt es zu JGraphT beim Allozieren von Graphen, Knoten und Kanten von Graphen?
- Welche Informationen kann man aus einem Graphknoten herausholen?
- Welche Nachteile hat die Graph-Klasse von GELLY, die nicht in eine Vererbungshierarchie eingebettet ist?
- Würden Sie GELLY als *Software* oder nur als *Programm* bezeichnen?

Was haben wir gelernt?

- Objektnetze, die in einem UML-Modell mit Assoziationen spezifiziert worden sind, können direkt mit JGraphT realisiert werden
  - Es gibt viele Varianten von Graphen
  - Fabrikmethoden für verschiedene Implementierungen von Knoten, Kanten, Graphen
- Sichten auf Graphen möglich
- Analyen durch Funktionalobjekte
- Analysen sind weitreichend nutzbar (s. Vorlesung Softwaretechnologie-II)

Was haben wir gelernt?

- UML Assoziationen können mit JGraphT direkt realisiert werden
  - Es gibt viele Varianten von Graphen
  - Fabrikmethoden für verschiedene Implementierungen von Knoten, Kanten, Graphen
- Sichten auf Graphen möglich
- Analyen durch Funktionalobjekte
- Analysen sind weitreichend nutzbar (s. Vorlesung Softwaretechnologie-II)