

# 25. Konstruktion von flexiblen Objektnetzen mit Graphbibliotheken in Java

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und  
Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

Technische Universität Dresden

Version 19-0.2, 17.05.19

1) Implementierungsmuster Fabrikmethode

2) Implementierungsmuster Kommando

3) Das Graph-Framework JGraphT

1) Aufbau

2) Checker

3) Iteratoren

4) Delegatoren für Sichten

5) Analysatoren: Kürzeste Pfade

6) Generatoren



# Obligatorische Literatur

- ▶ JDK Tutorial für J2SE oder J2EE, [www.java.sun.com](http://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](http://ci.apache.org/projects/flink/flink-docs-master/gelly_guide.html)

# Hinweis: Orientierungsplattform Forschung und Praxis auf Bildungsportal Sachsen

3

Softwaretechnologie (ST)

- ▶ <https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/12520161286/CourseNode/94468231277848;jsessionid=D463838C980B367A1738D694E15A67A9.opalN8?0>

- ▶

# Ziele dieses Kapitels: Entwurfsmuster in Aktion

- ▶ Eine komplexe Java-Bibliothek aus dritter Hand, eine Graphen-Bibliothek, kennenlernen
  - Graphen als spezielle, flexible Objektnetze verstehen
  - Einsatz mehrerer Entwurfsmuster in einem Framework sehen
  - Die Bibliothek **vereint viele Entwurfsmuster**, die wir kennengelernt haben
    - Fabrikmethoden, Iteratoren und Streams in Anwendung bei Graphen
- ▶ **Anwendungen**
  - Geoinformatik ([www.openstreetmap.org](http://www.openstreetmap.org))
  - Öffentliche Netzinfrastrukturen (“intelligente Netze”) wie Eisenbahnnetz, Wasserleitungsnetz, Stromnetz
  - Pläne von Gebäuden, Städten, Verkehrswegen
  - Weltmodelle für Roboter
  - Soziale Netze wie Facebook
- ▶ Generische Graphalgorithmen kennenlernen
  - Delegatoren
  - Generatoren
  - Graphanalysen

# Nicht-obligatorische Literatur

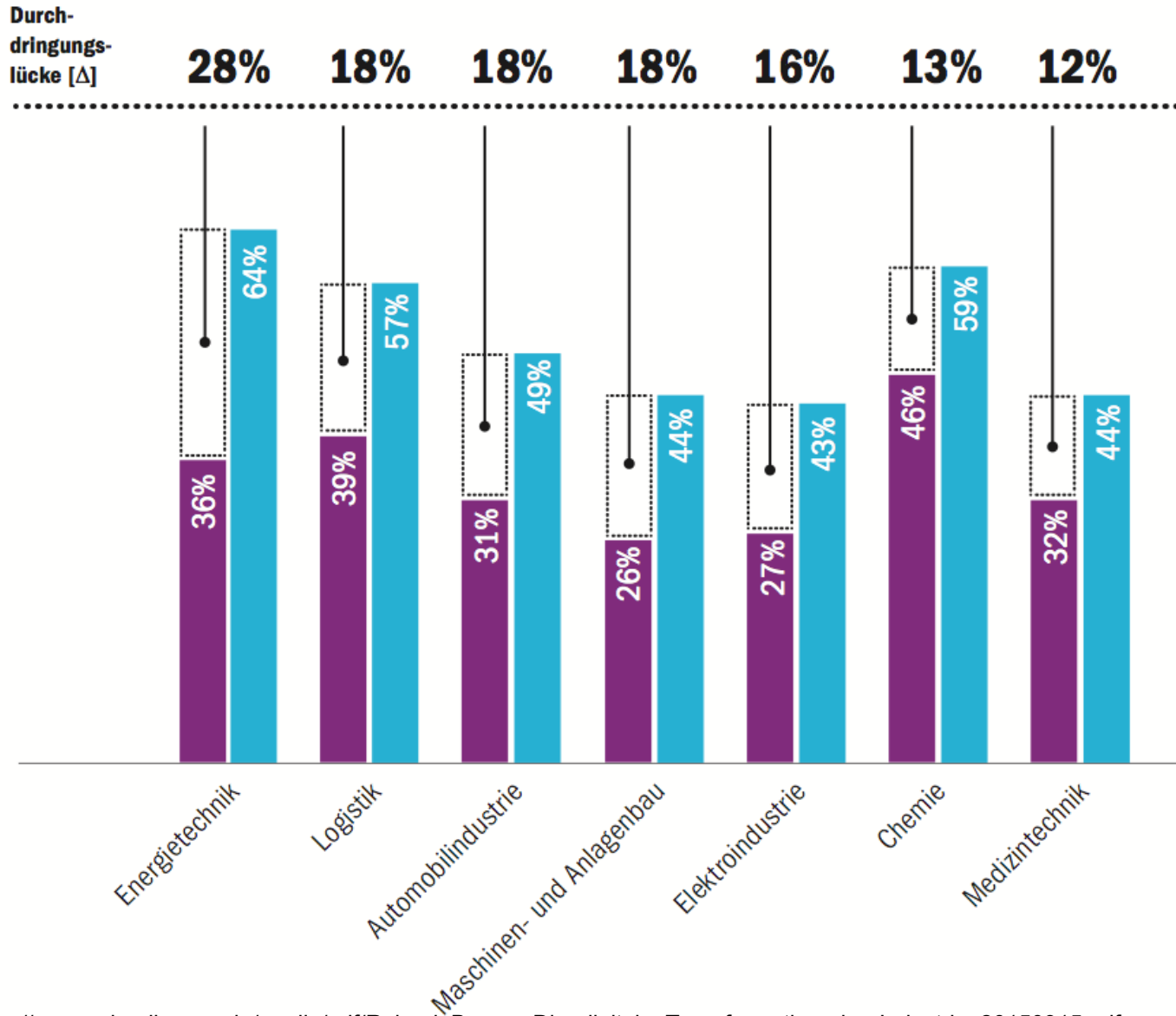
- ▶ [HB01] Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In Jan Bosch, editor, GCSE, volume 2186 of Lecture Notes in Computer Science, pages 10-25. Springer, 2001.
  - Facetten von Graphen und wie man sie systematisch, noch besser in einem Framework anordnet
  - Siehe Vorlesung “Design Patterns and Frameworks”

**Einschätzung der eigenen digitalen Reife**

**Chancen der Branche aus der Digitalisierung**

■ "Sehr hoch" oder "hoch"

■ "Sehr groß" oder "groß"



[http://www.rolandberger.de/media/pdf/Roland\\_Berger\\_Die\\_digitale\\_Transformation\\_der\\_Industrie\\_20150315.pdf](http://www.rolandberger.de/media/pdf/Roland_Berger_Die_digitale_Transformation_der_Industrie_20150315.pdf)

1) Luft- und Raumfahrttechnik aufgrund nicht repräsentativer Zahl von Antworten von Industrievergleichern ausgeschlossen  
 Quelle: Roland Berger, Umfrage unter 300 Top-Managern der deutschen Wirtschaft

# 25.1 Implementierungsmuster

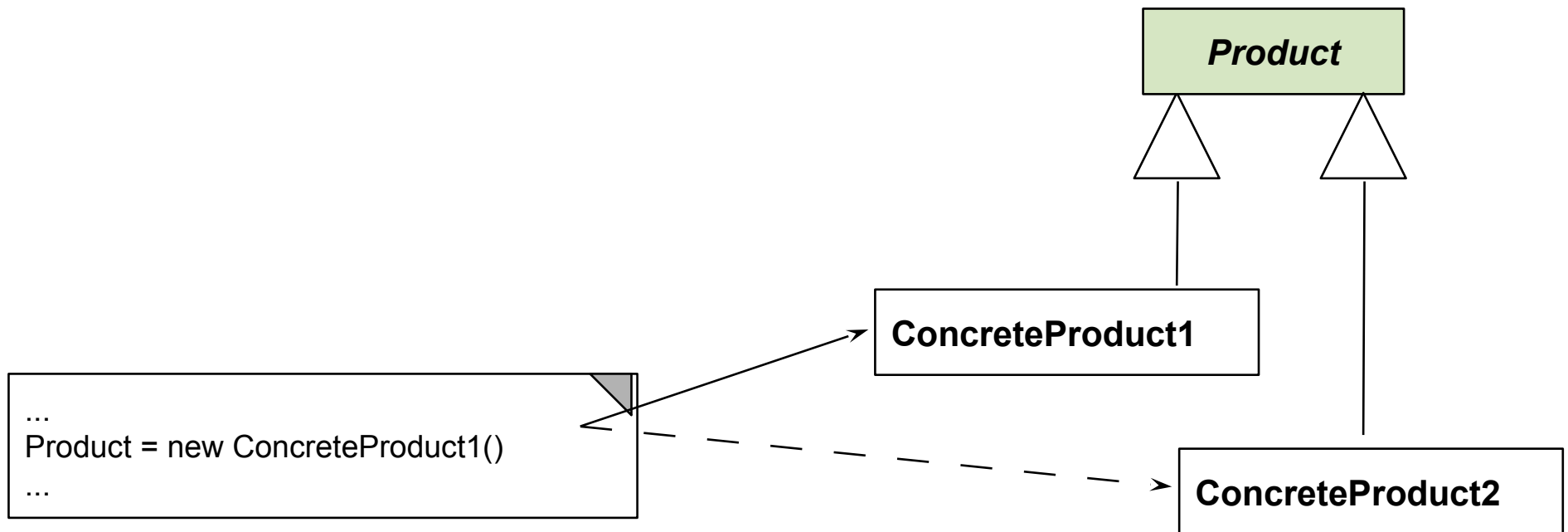
## Fabrikmethode (FactoryMethod)

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



# Problem der Fabrikmethode

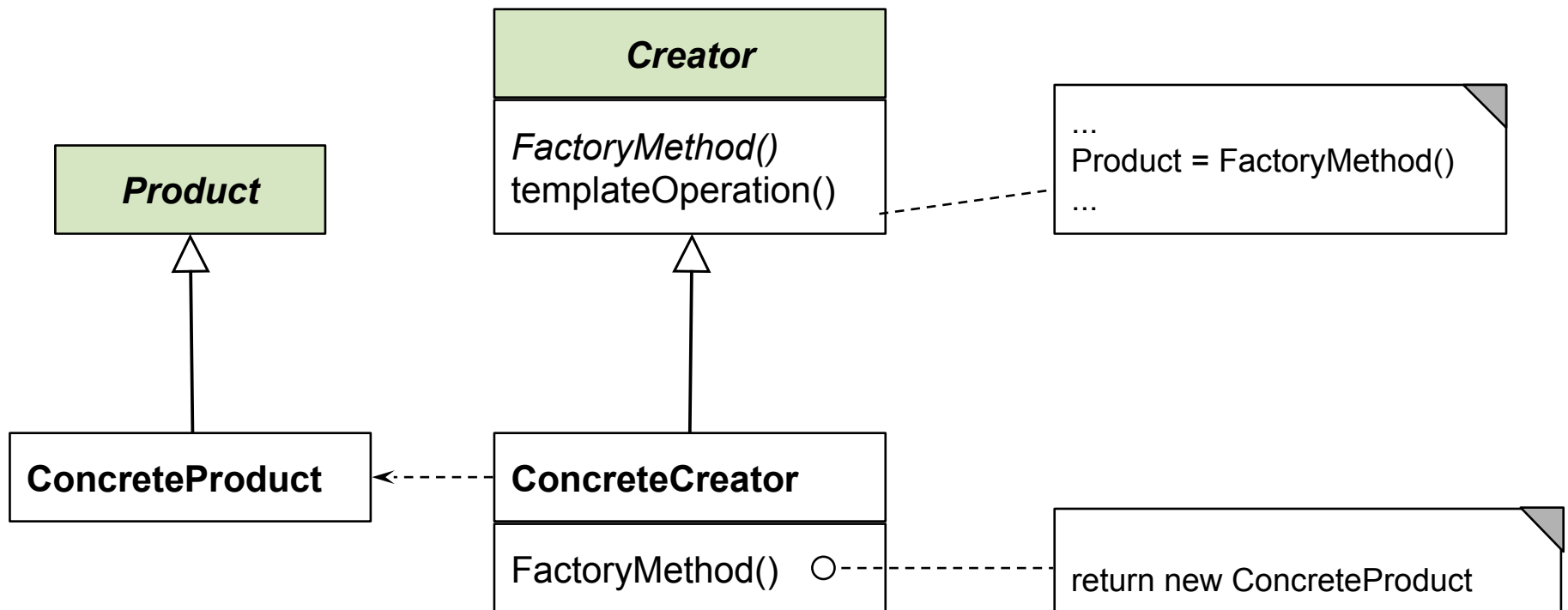
- ▶ Wie variiert man die Erzeugung für eine polymorphe Hierarchie von Produkten?
- ▶ Problem: Konstruktoren sind nicht polymorph!





# Struktur Fabrikmethode

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



# Fabrikmethode (Factory Method)

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

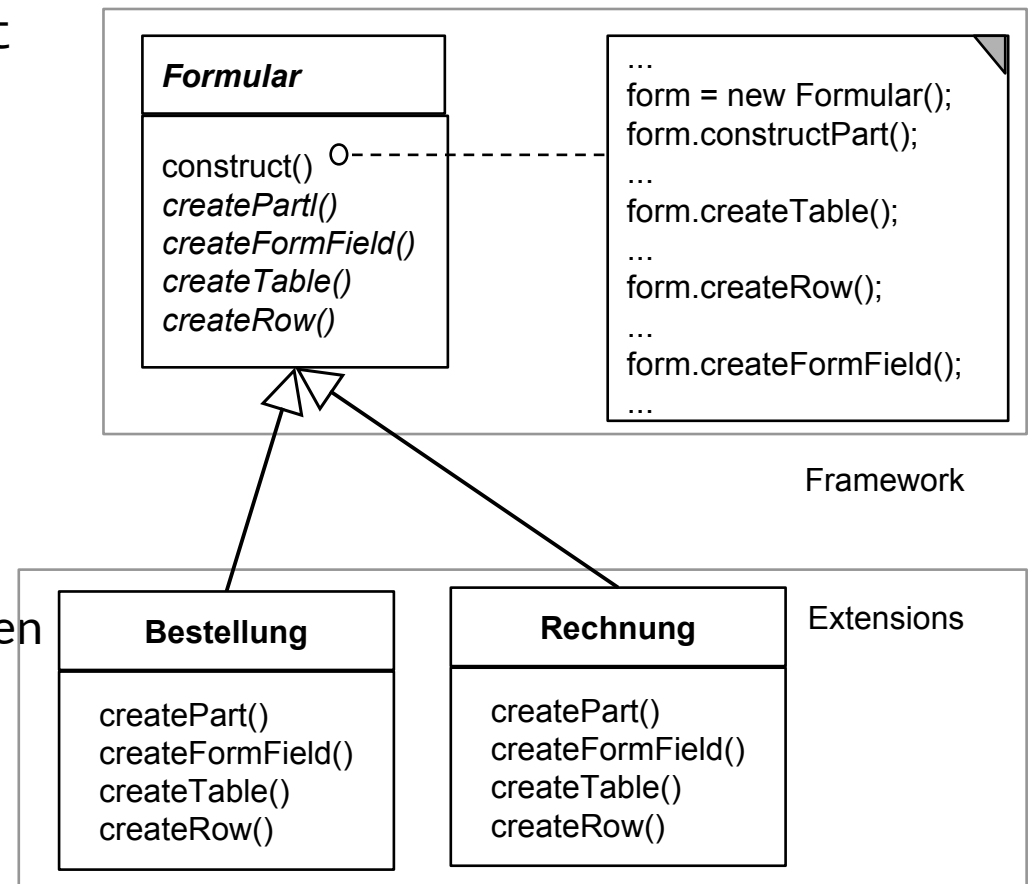
```
public class Client {  
    ...  
    Creator cr = new ConcreteCreator();  
    // call a big factory method  
    cr.collect();  
}
```

```
// 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 Anwendung von FactoryMethod in einem Formular-Framework

- ▶ Framework (Rahmenwerk) für Formulare
  - Klasse Formular hat eine Schablonenmethode `construct` 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

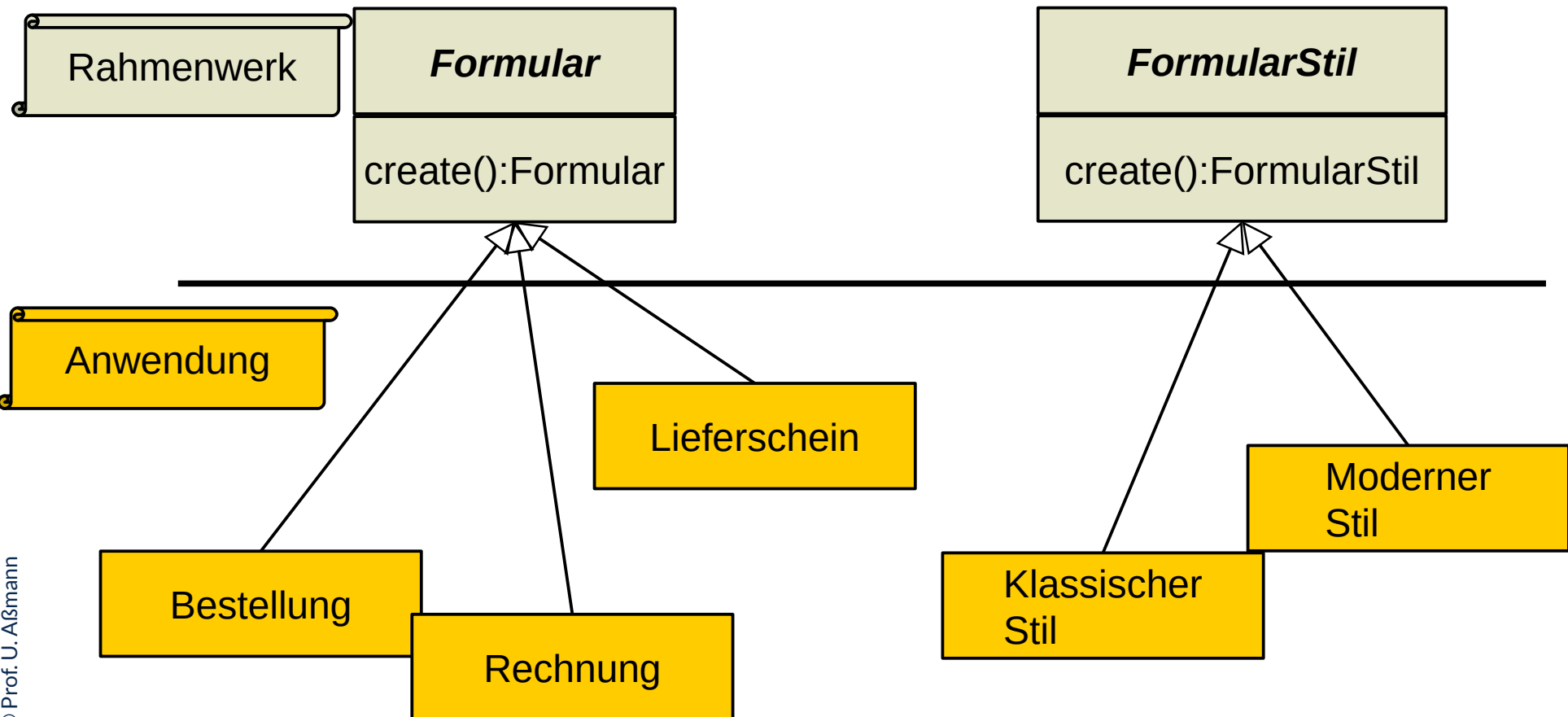
- ▶ Bilde createFormular() als Fabrikmethode aus

```
// 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();
    }
    ...
}
```

# Einsatz in Komponentenarchitekturen

- ▶ In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von den Anwendungsschichten aus die Rahmenwerkschicht zu konfigurieren:



## 25.2 Implementierungsmuster Kommandoobjekt (Command)

Zur flexiblen Behandlung von Aktionen



# Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Anwendungsfunktion 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 *dolt()*)

Vorteile:

- Man das Funktionalobjekt mit Parametern versehen, herumreichen, und zum Schluss ausführen (*partielle Applikation von Kommandos*)
- Funktionalität wie *undo()*, *redo()*, *persist()*

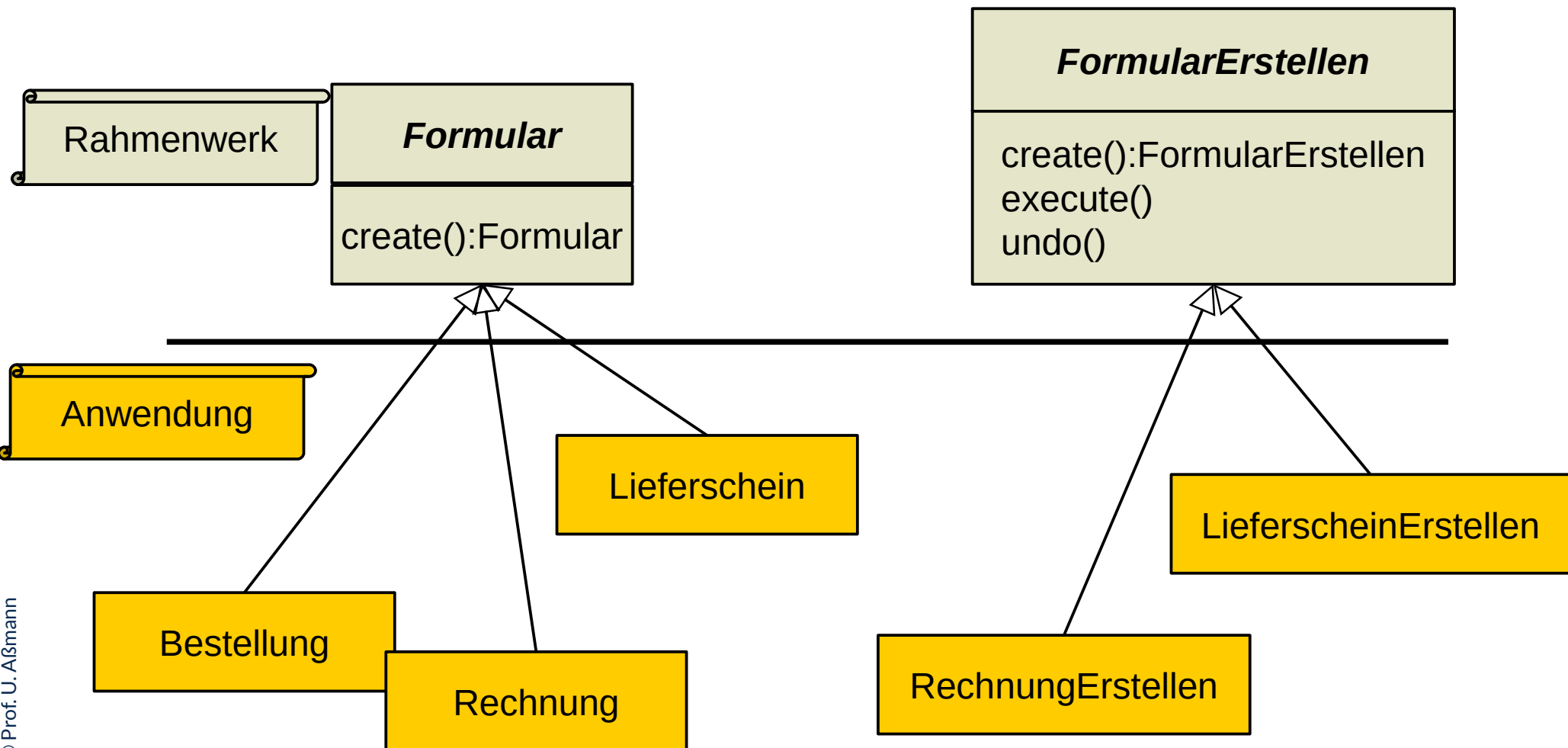


```
// A command object captures a method
public abstract class Command {
    void execute();
    void takeArgument(<T> arg);
}
public class Undoable extend Command {
    void undo();
}
// Repeatable Command
public class Repeatable extend Undoable {
    void redo();
}
```

```
// Repeatable Command
public class PersistentCommand extends
    Repeatable {
    void persist();
}
```

# Einsatz in Komponentenarchitekturen

- ▶ In Rahmenwerk-Architekturen wird das Kommando-Objekt eingesetzt, um ein Default-Verhalten zur Verfügung zu stellen, das verändert werden kann







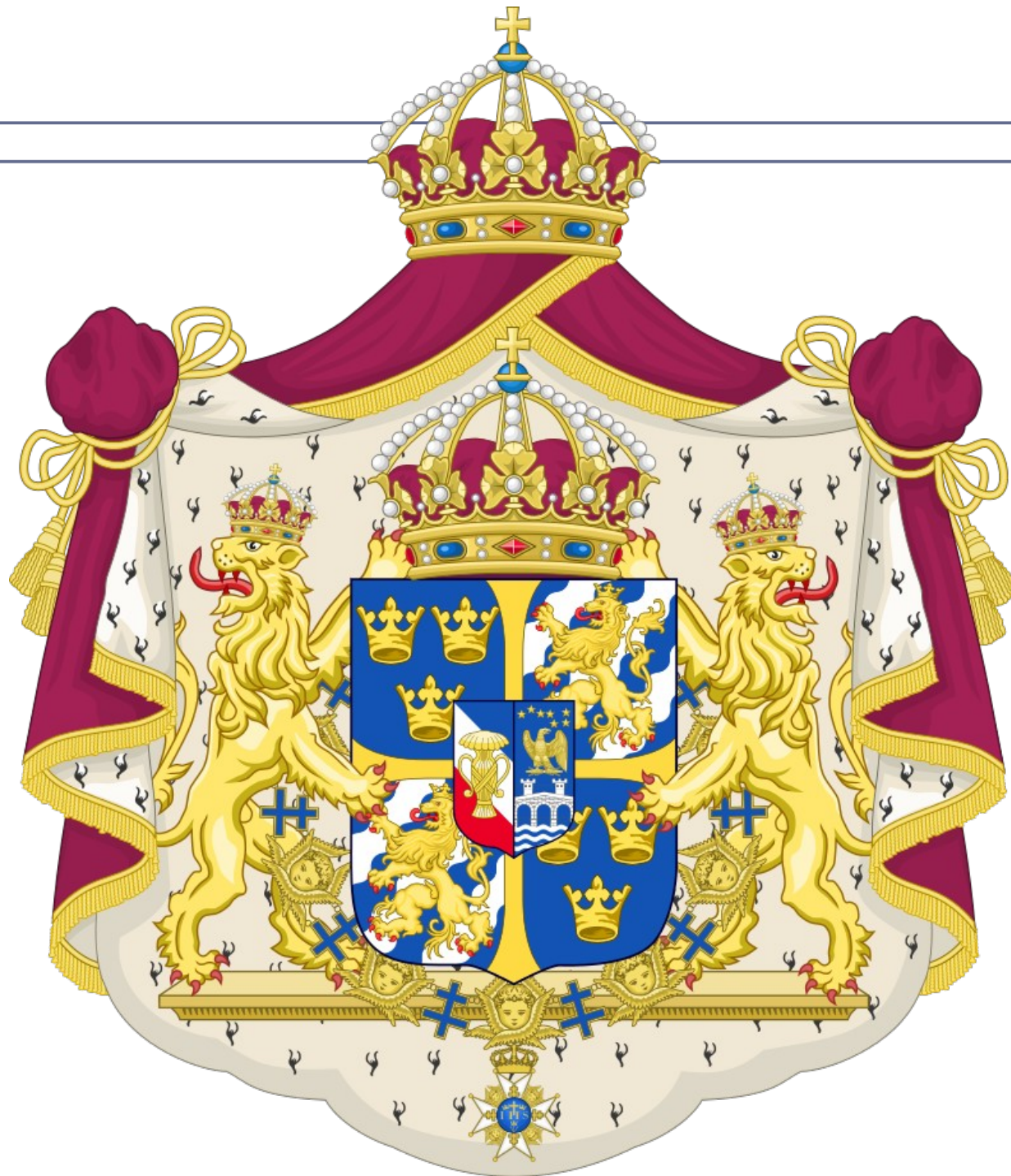
Für jedes  
Command  
benötigt man  
eine Factory!

## 25.3 Einsatz von FactoryMethod und Command im JGraphT Framework

Fabriken, Iteratoren, Kommandoobjekte im Großeinsatz

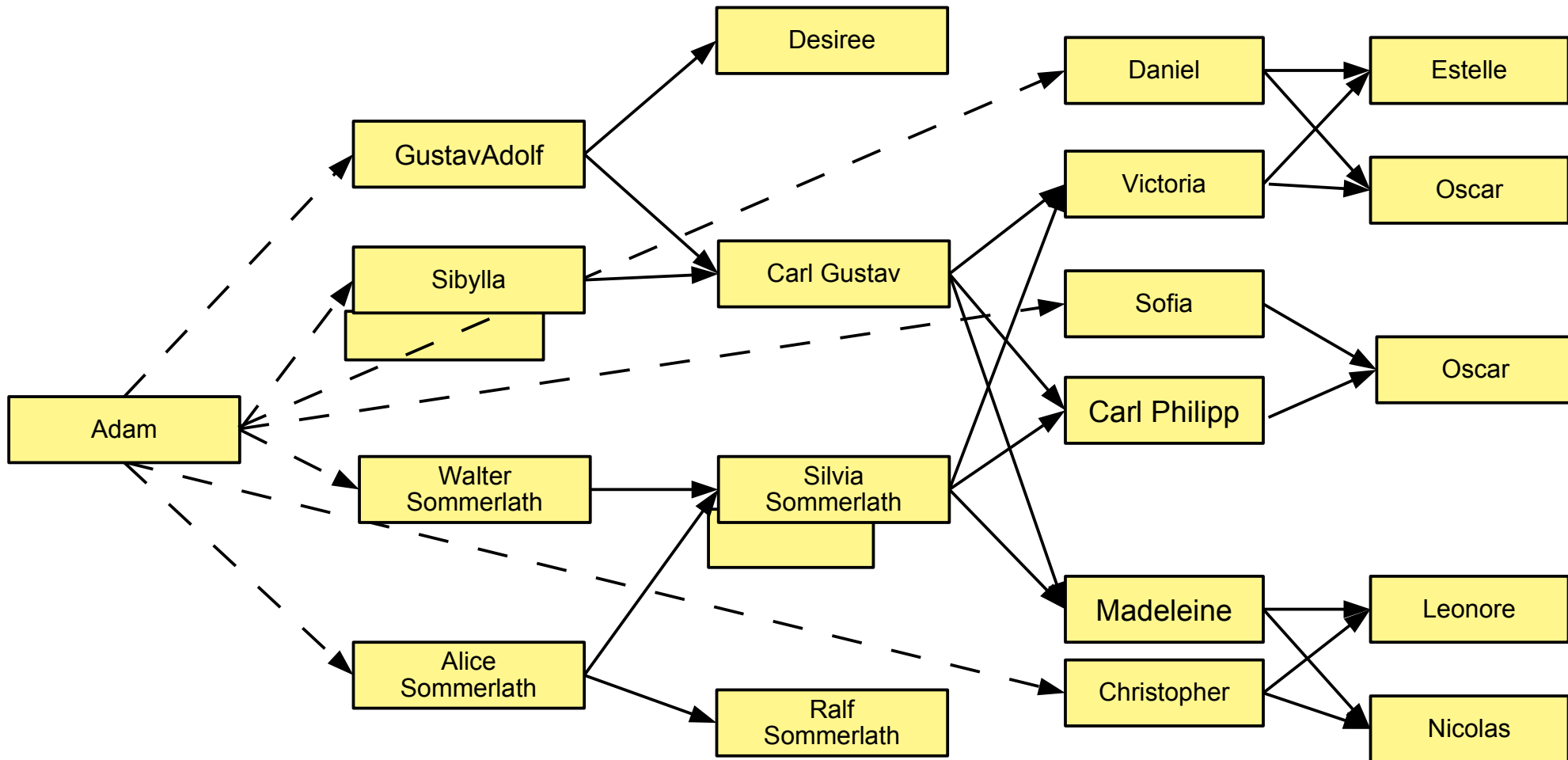


# Beispiel



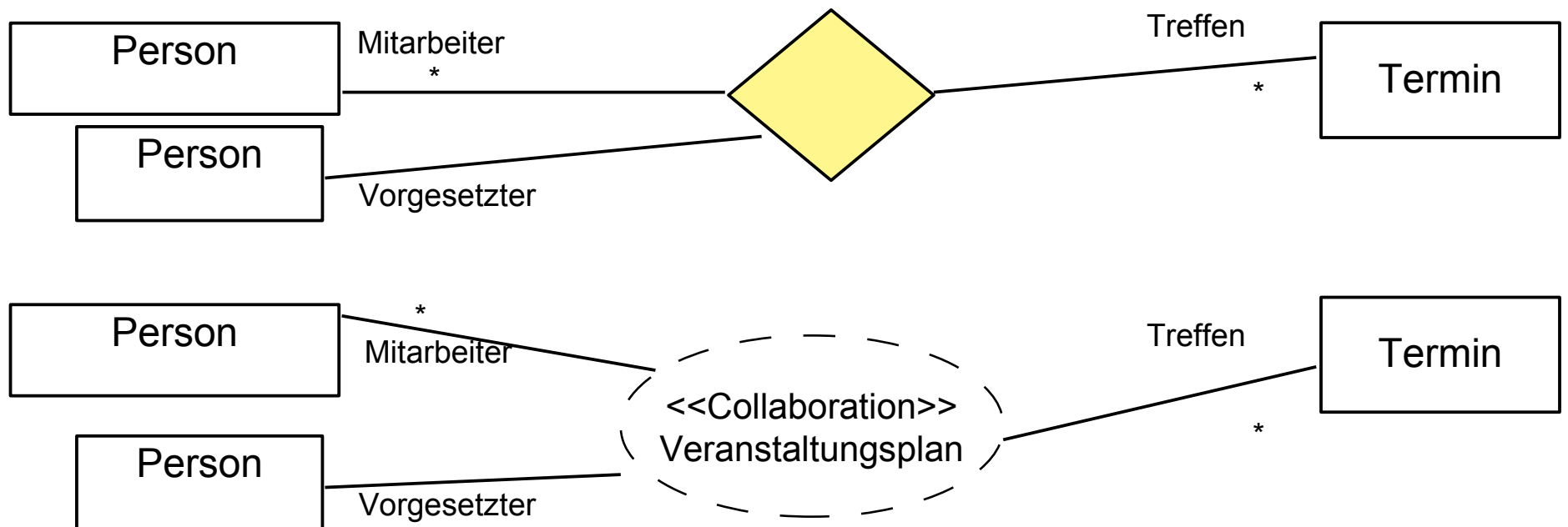
# Beispiel: Verwandtschaftsbeziehungen

- ▶ Familienbeziehungen sind immer azyklisch
- ▶ Die schwedische Königsfamilie als Graph:



# 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
- ▶ Über einem Graphen kann man Kollaborationen (“Ellipsen”) definieren



# Ziele einer Graph-Bibliothek

- ▶ Ziel: Management der Kollaboration von flexiblen Objektnetzen
  - Iteration, Navigation, Algorithmen
- ▶ 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	

## <<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) <
```

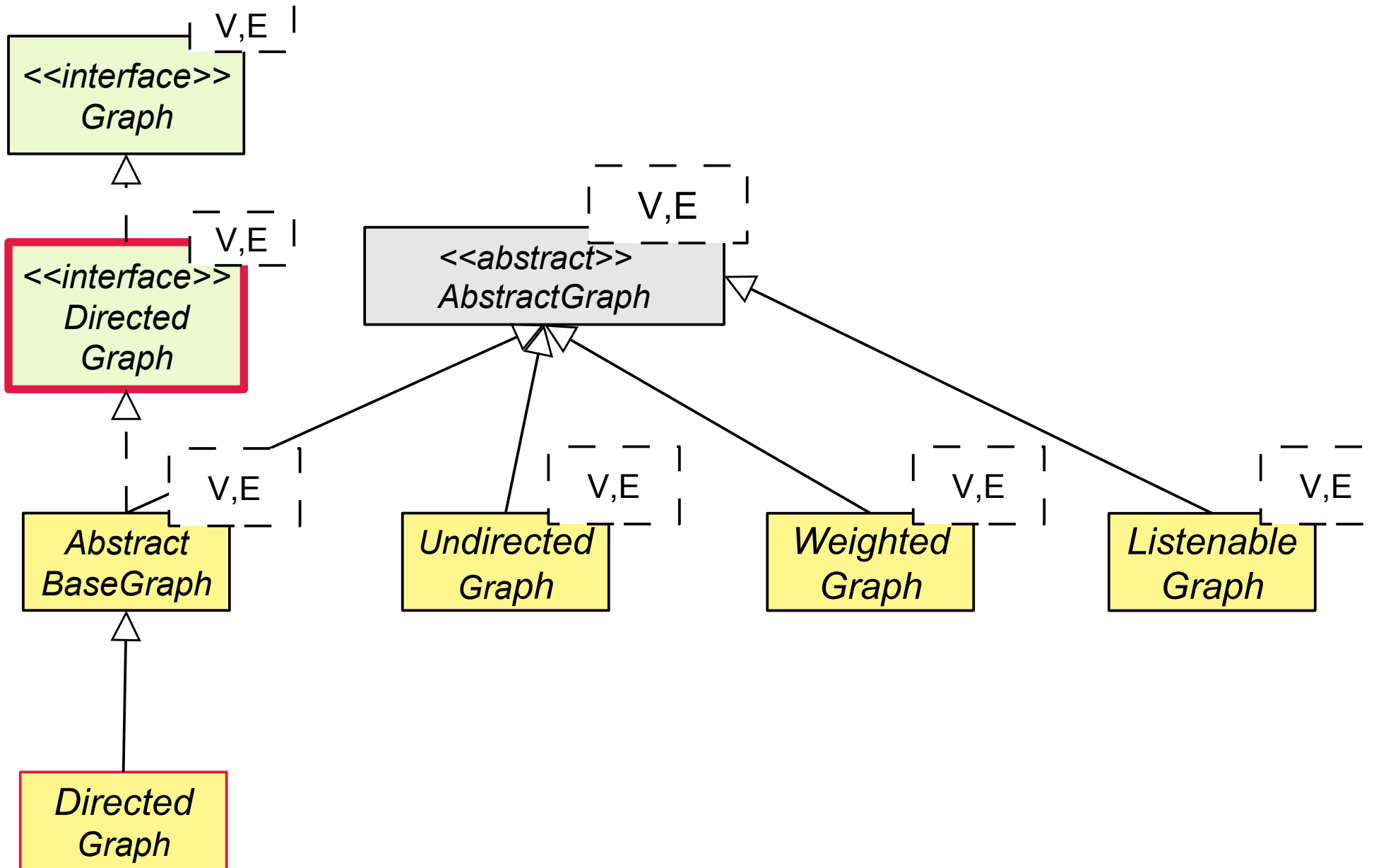


# DirectedGraph.java in JGraphT

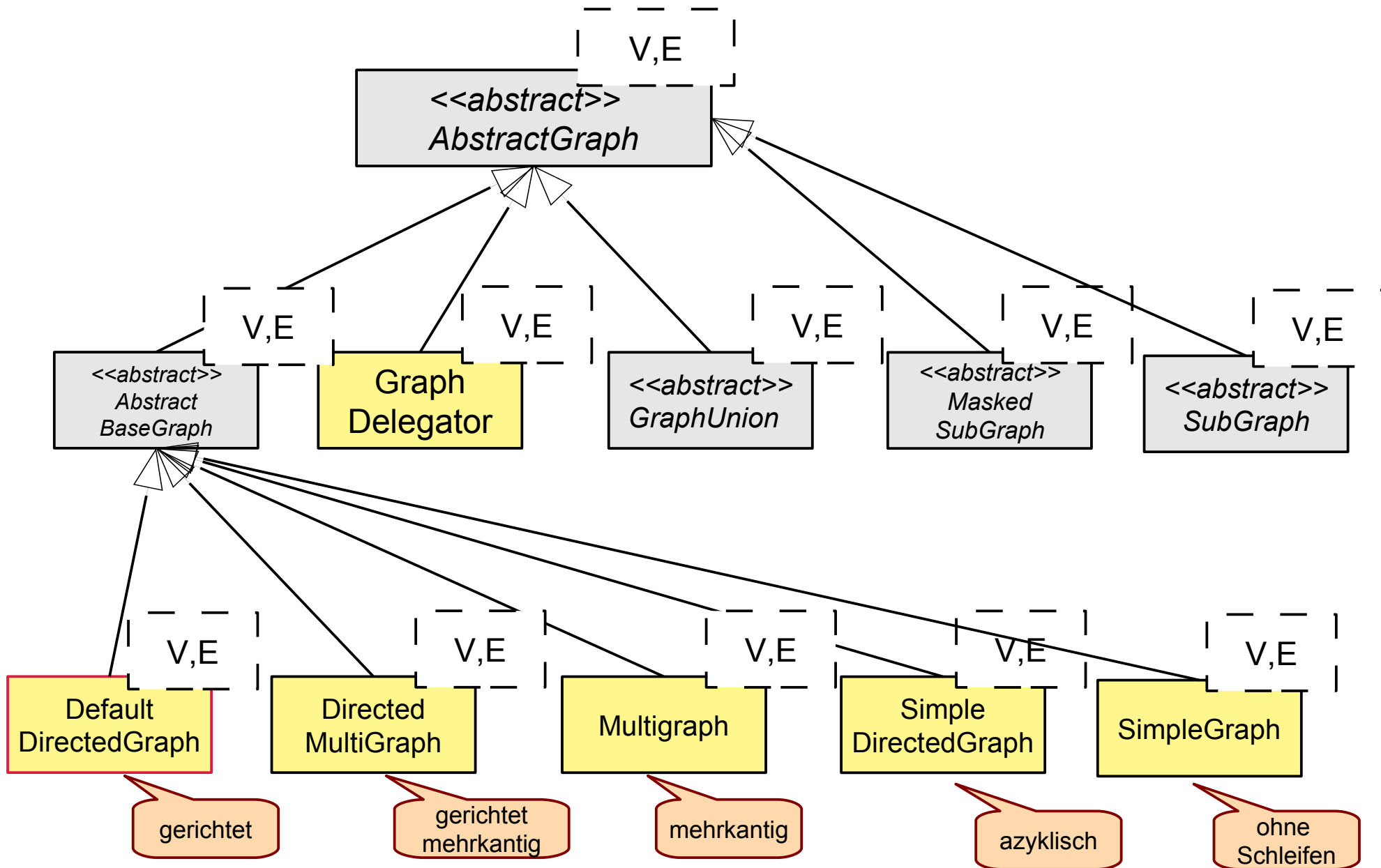
## **<<interface>> DirectedGraph<V,E>**

```
// 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 Schnittstellenhierarchie Graph

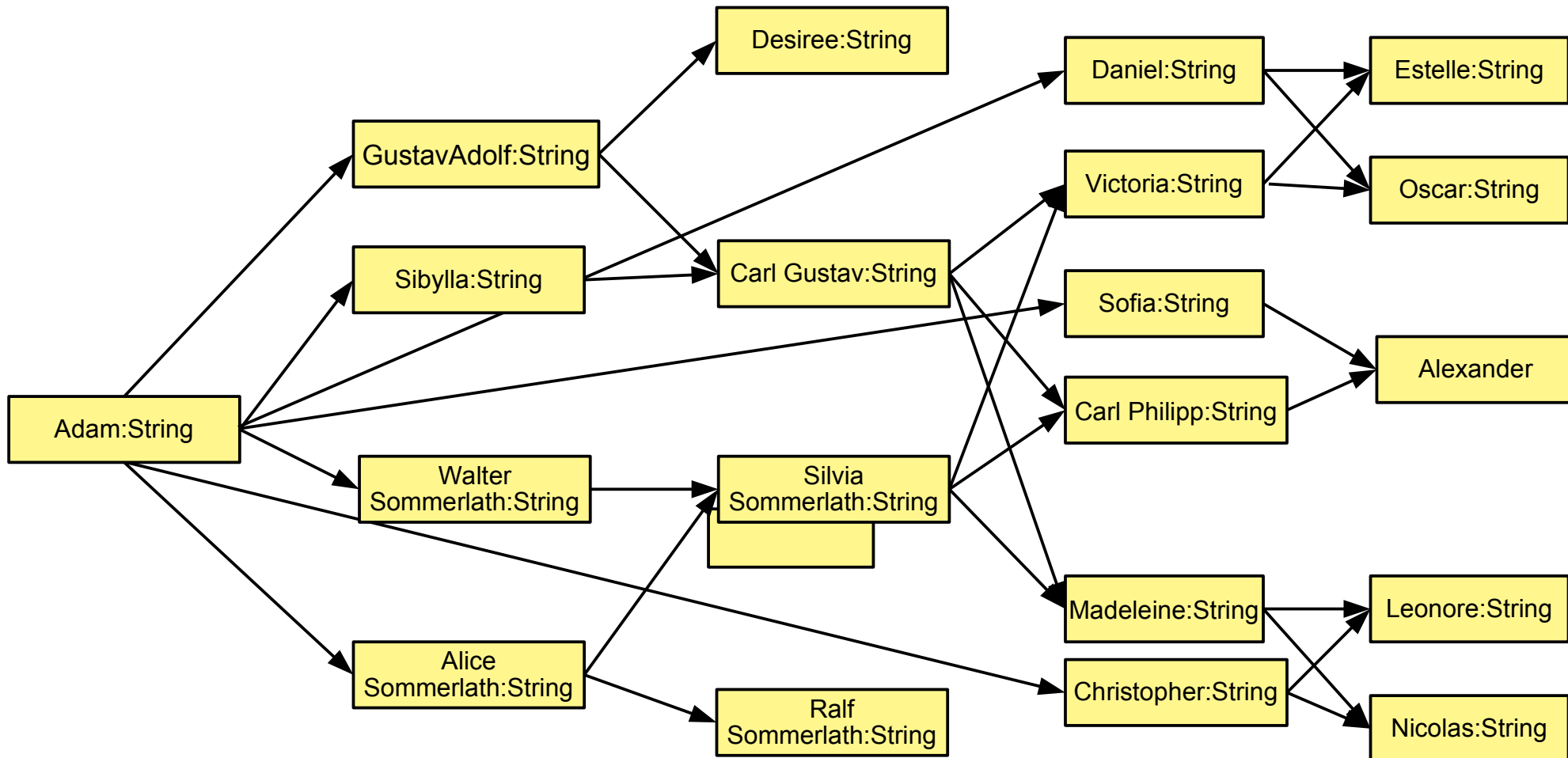


# Die Implementierungshierarchie Graph

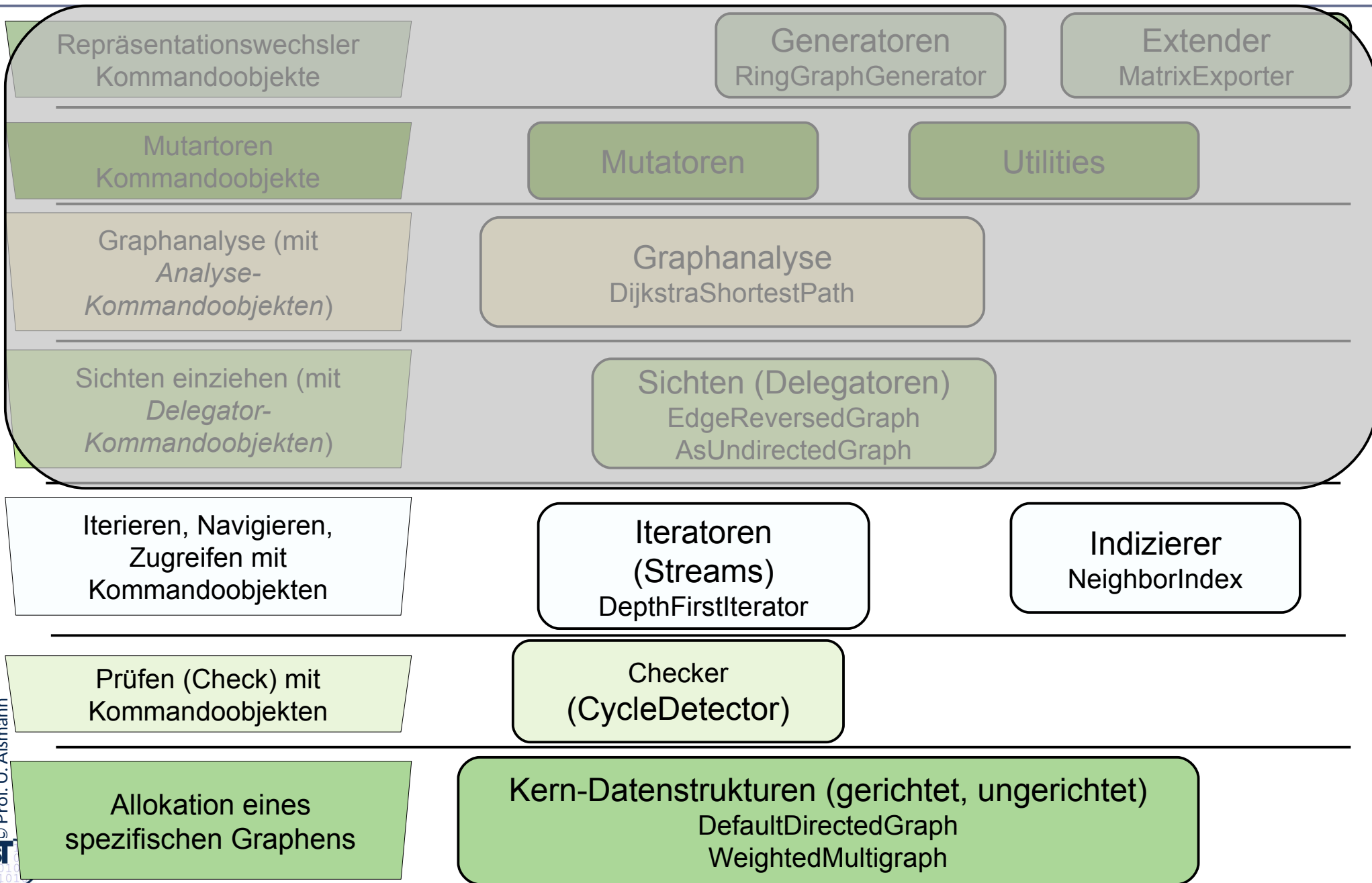


# Beispiel: Verwandtschaftsbeziehungen

- ▶ Familienbeziehungen sind immer azyklisch
- ▶ Die schwedische Königsfamilie als UML-Objektnetz:



# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT



## 25.3.1. Aufbau gerichteter Graphen

SwedishFamilyDemo.java

32 Softwaretechnologie (ST)

```
// 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";
String estelle = "Estelle";
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);
parentOf.addVertex("Daniel");
parentOf.addVertex("Christopher");
parentOf.addVertex("Sofia");
parentOf.addVertex(estelle);
parentOf.addVertex("Oscar");
parentOf.addVertex("Leonore");
parentOf.addVertex("Nicolas");
```

```
// 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");
parentOf.addEdge("Daniel", "Estelle");
parentOf.addEdge("Victoria", "Estelle");
parentOf.addEdge("Daniel", "Oscar");
parentOf.addEdge("Victoria", "Oscar");
parentOf.addEdge("Madeleine", "Leonore");
parentOf.addEdge("Madeleine", "Nicolas");
parentOf.addEdge("Christopher", "Leonore");
parentOf.addEdge("Christopher", "Nicolas");

/* 1 */ // parentOf.addEdge(estelle, adam);
```

## 25.3.2. Konsistenzprüfung und Navigation mit Check-Kommandos

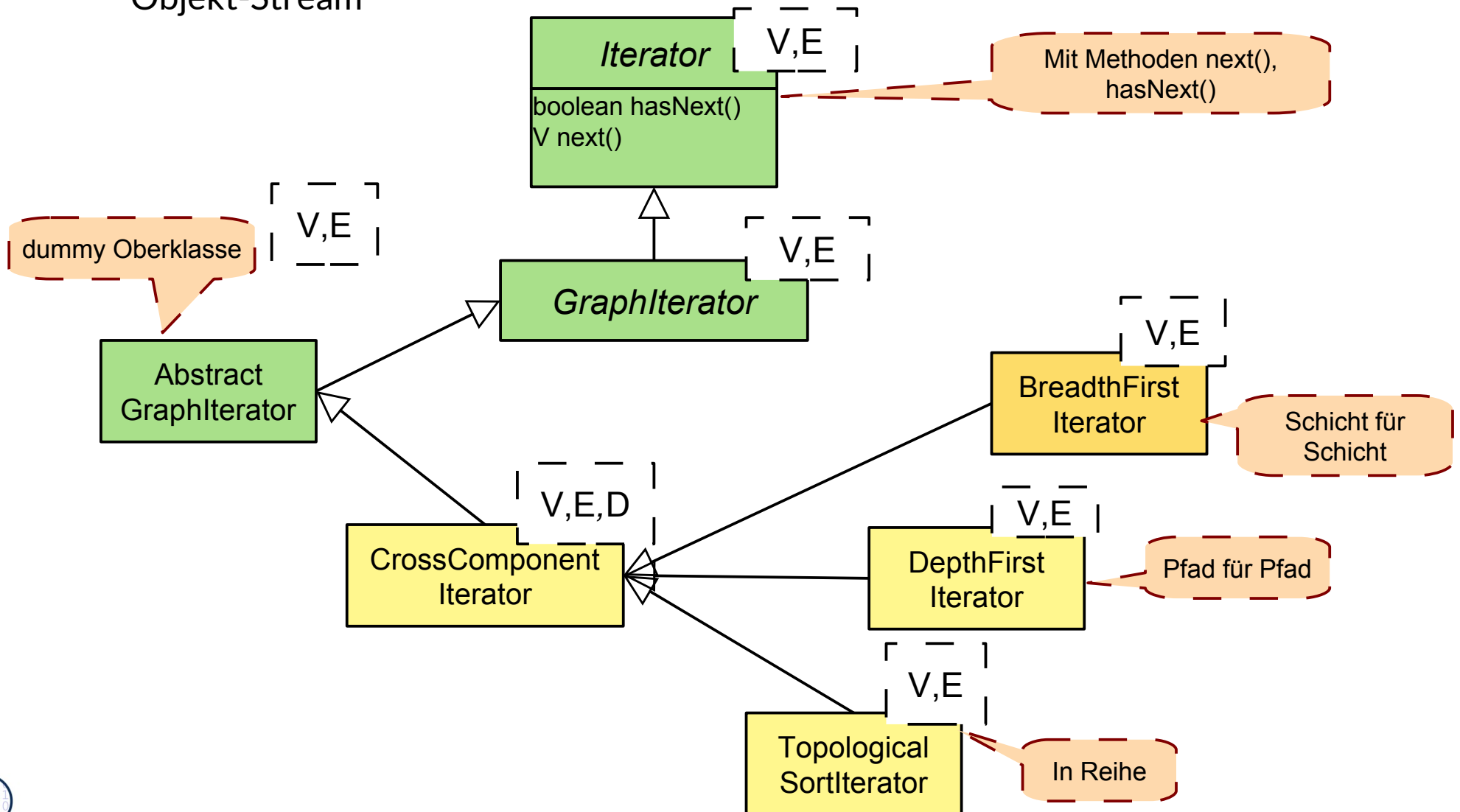
- ▶ Die meisten generischen Algorithmen von jgrapht sind Kommando-Objekte (Entwurfsmuster Command)
- ▶ `CycleDetector.findCycles()` ist ein Check-Kommando und findet Zyklen im Graphen, jenseits von Selbstkanten
  - Entspricht `execute()`

```
// (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());
```

## 25.3.3 Iteratoren laufen Graphen ab

- Man kann mit einem Graphiterator den Graphen ablaufen und seine Knoten ausgeben, ohne seine Struktur zu kennen. Der Iterator wandelt also den Graph in einen Objekt-Stream

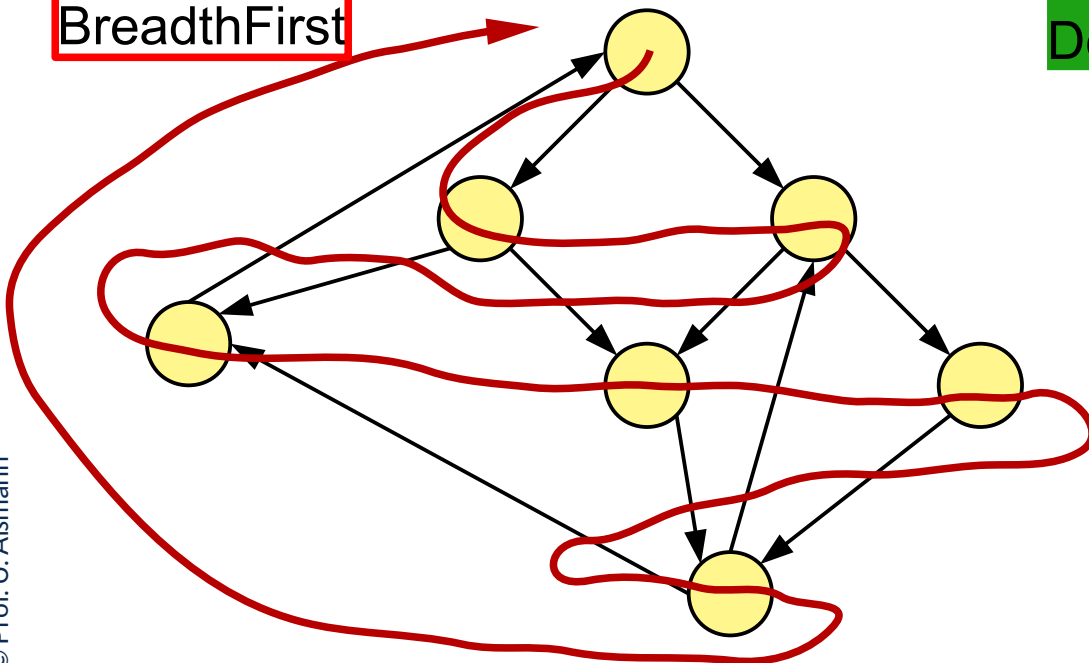




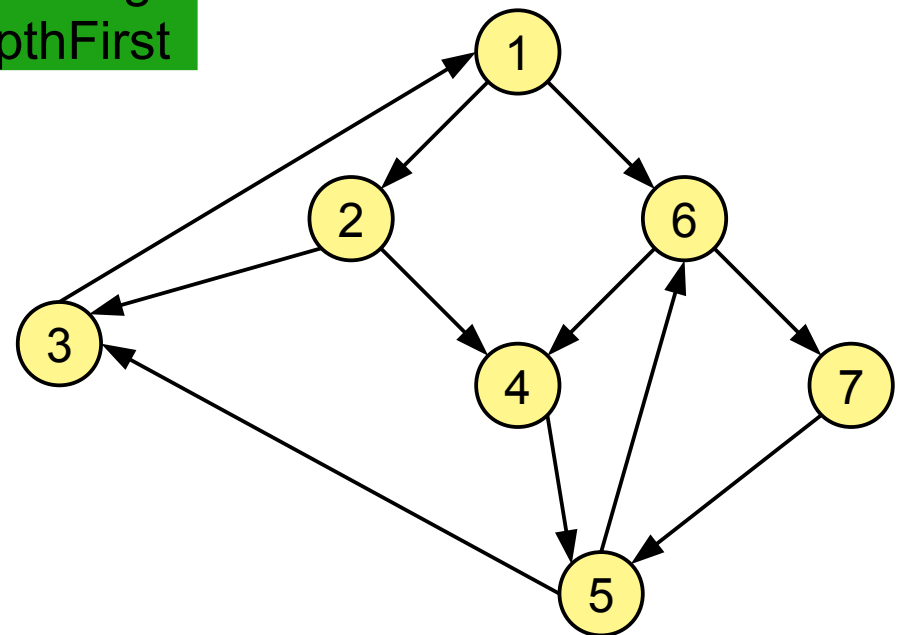
# Arten von Durchläufen mit Iteratoren

- ▶ `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”

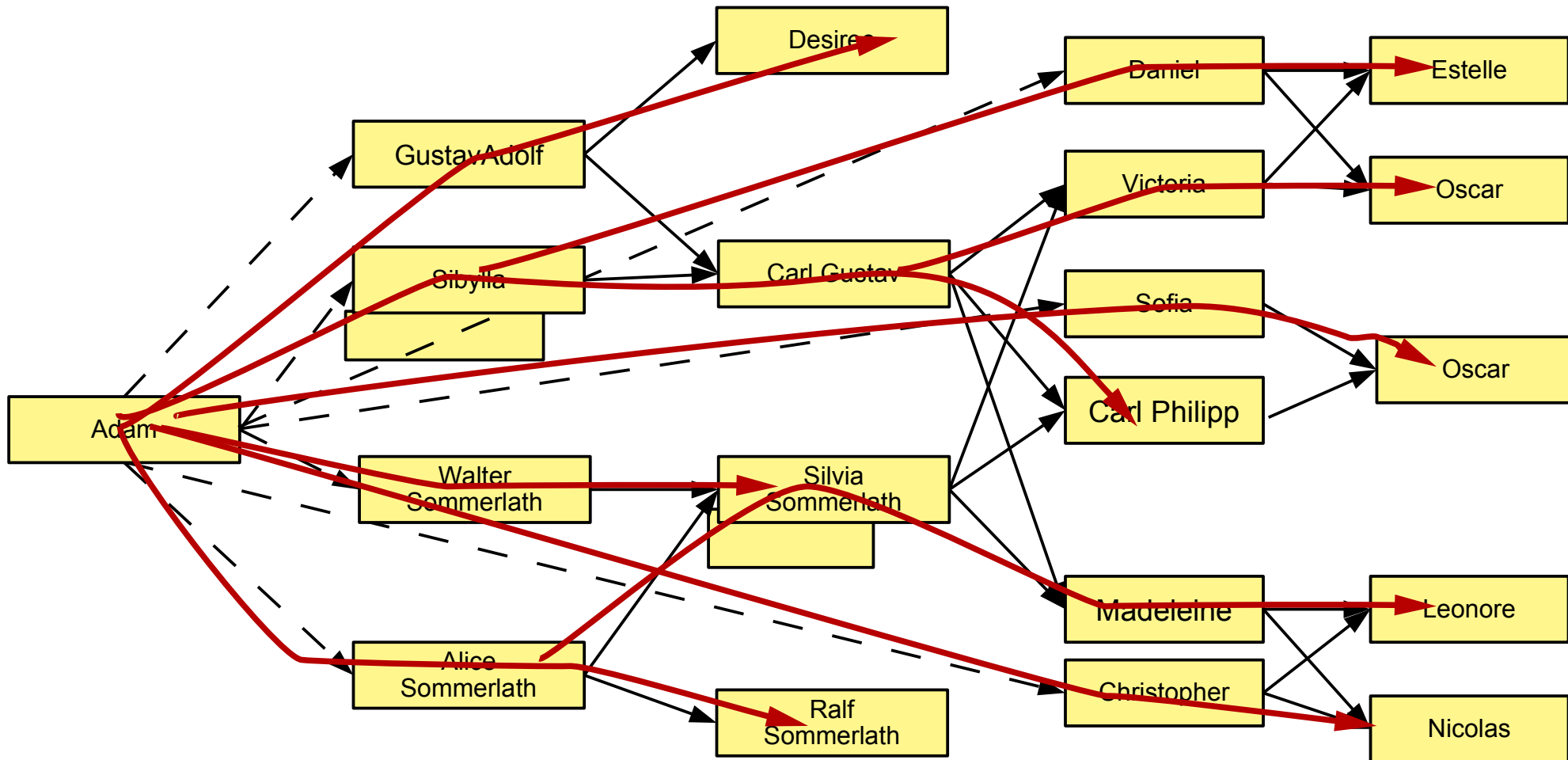
Left-to-right  
BreadthFirst



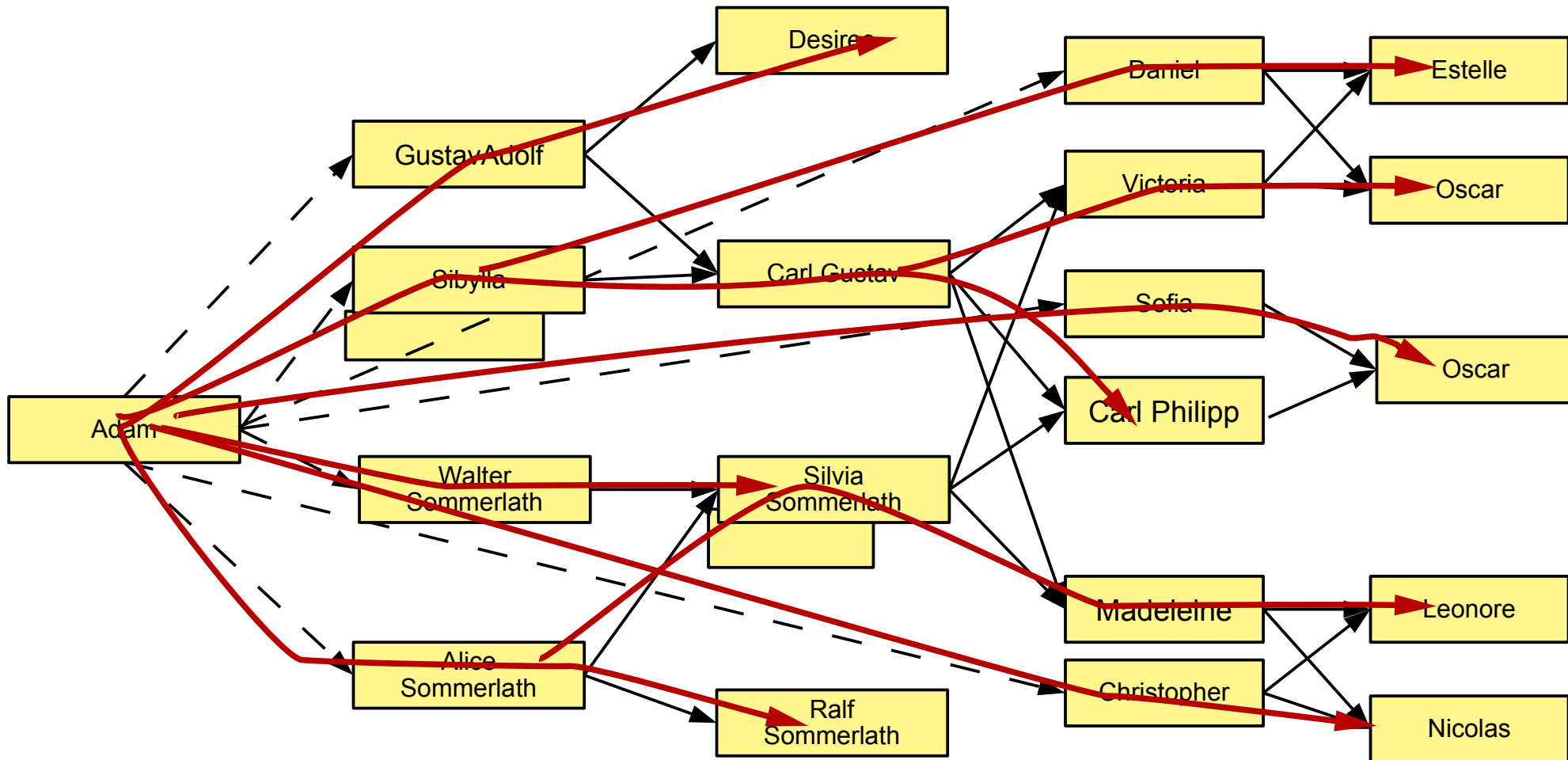
Left-to-right  
DepthFirst



# Tiefensuche auf dem azykl. Graphen der Königsfamilie



# Tiefensuche auf dem azykl. Graphen der Königsfamilie



# Examples for 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);
}
```



## 25.4 Weitere Schichten im JGraphT Framework

Die Königsklasse bei den Frameworks: Schichtenbildung



# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT

40 Softwaretechnologie (ST)

Repräsentationswechsler  
Kommandoobjekte

Generatoren  
RingGraphGenerator

Extender  
MatrixExporter

Mutatoren  
Kommandoobjekte

Mutatoren

Utilities

Graphanalyse (mit  
*Analyse-*  
*Kommandoobjekten*)

Graphanalyse  
DijkstraShortestPath

Sichten einziehen (mit  
*Delegator-*  
*Kommandoobjekten*)

Sichten (Delegatoren)  
EdgeReversedGraph  
AsUndirectedGraph

Iterieren, Navigieren,  
Zugreifen mit  
Kommandoobjekten

Iteratoren  
(Streams)  
DepthFirstIterator

Indizierer  
NeighborIndex

Prüfen (Check) mit  
Kommandoobjekten

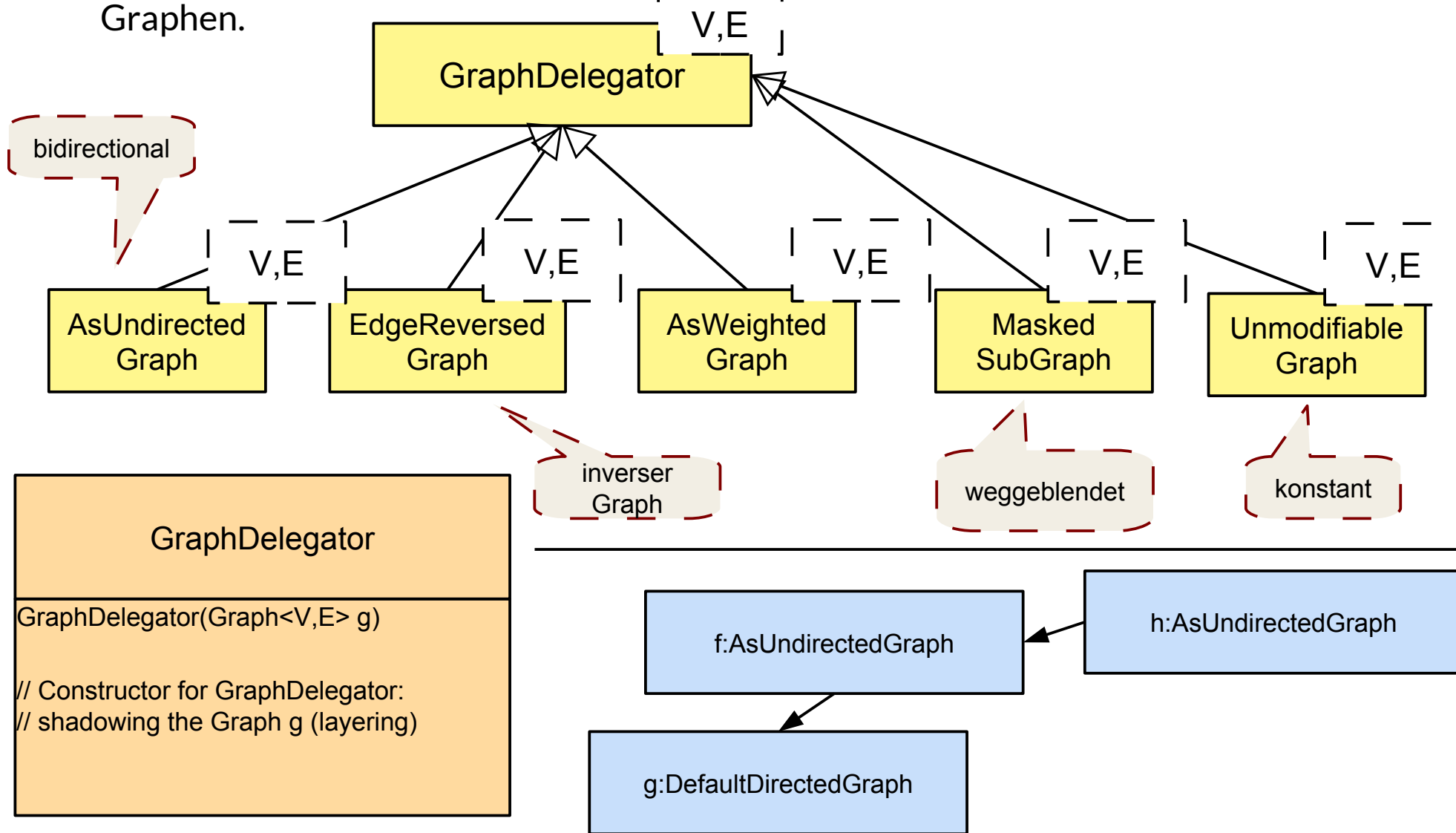
Checker  
(CycleDetector)

Allokation eines  
spezifischen Graphens

Kern-Datenstrukturen (gerichtet, ungerichtet)  
DefaultDirectedGraph  
WeightedMultigraph

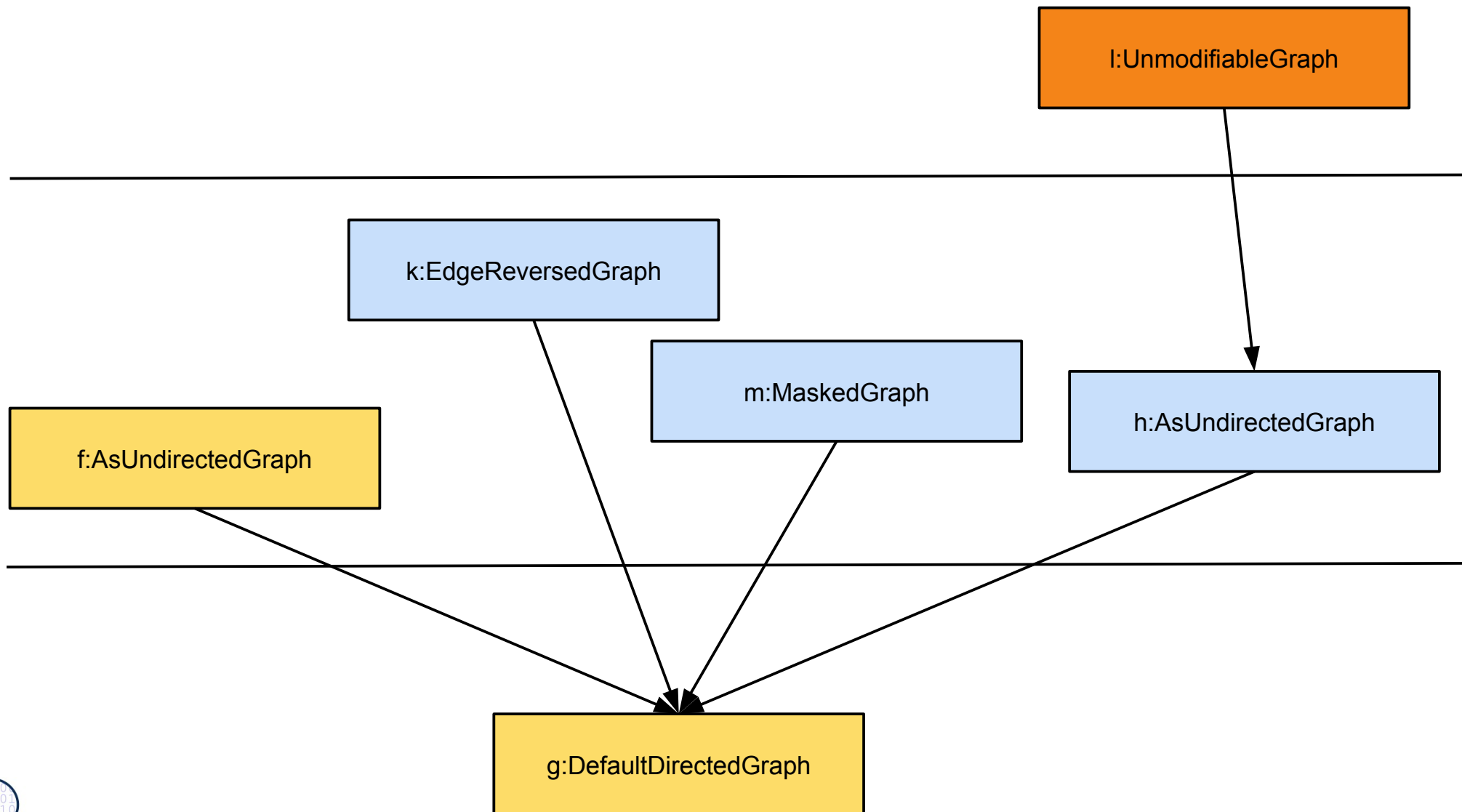
# 25.4.1 Delegatoren erzeugen Sichten

- Def.: Ein **Delegator** ist ein Objekt, das einen Graphen "vorspiegelt" und auf einen Basisgraphen anderen Typs zurückführt. Ein Delegator liefert eine Sicht auf einen Graphen.



# Schichtung von Graphen mit Delegatoren (Layering of Graphs)

- ▶ Was sieht ein Aufrufer (client) eines spezifischen Graphen?
- ▶ Snapshot hier:





## 25.4.2 Analysen in 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 **gerichteten** Graphen 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 ungerichteter Graph kann als Delegator auf einen anderen Graphen erstellt werden (Sicht).
- ▶ Dann kann mit **DijkstraShortestPath** auch auf der Sicht gesucht werden, d.h. der kürzeste Pfad in einem ungerichteten Graphen gesucht werden.

```
// 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()+"):");

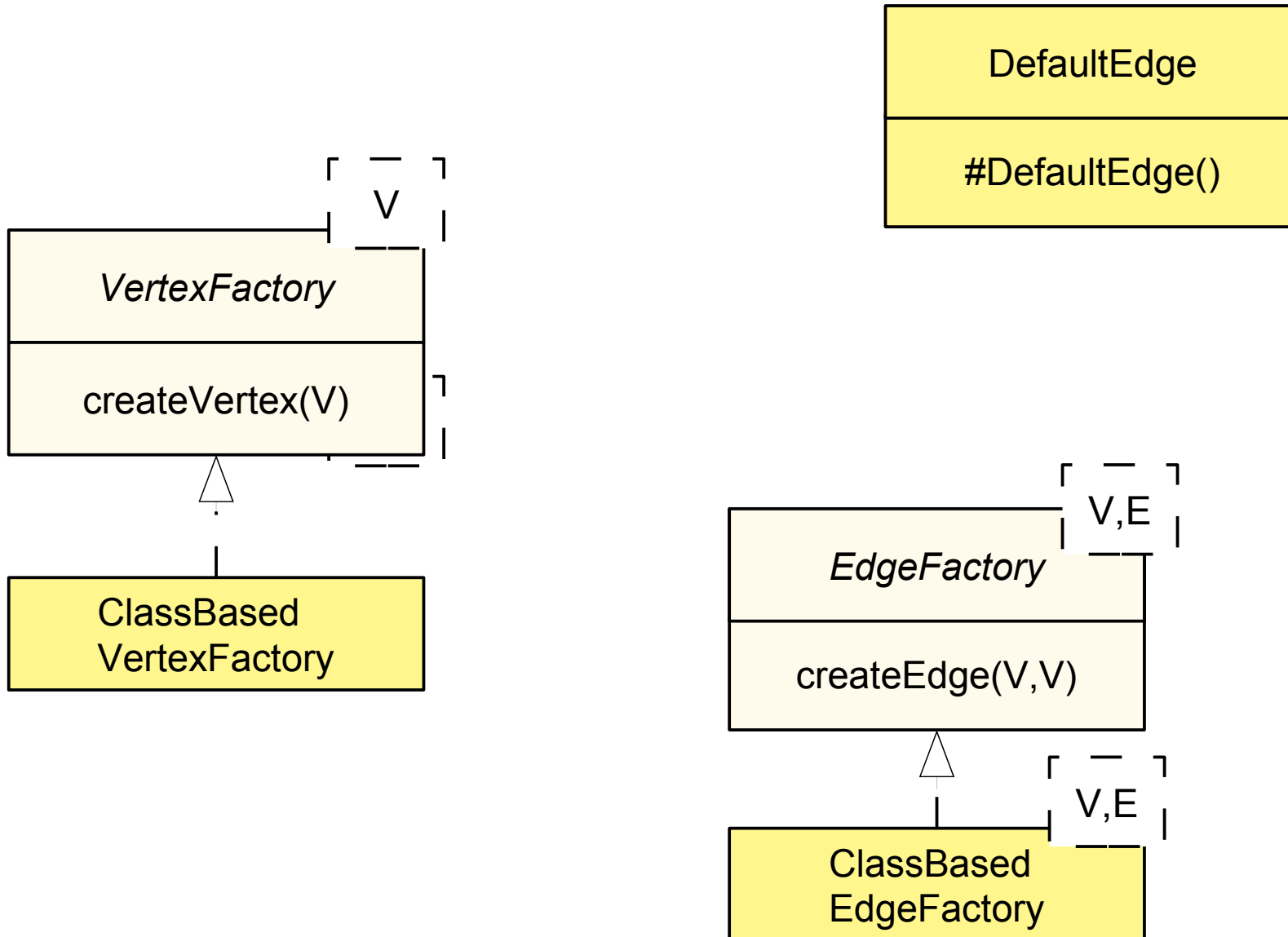
GraphPath<String,DefaultEdge> path = ancestorPath.getPath();

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



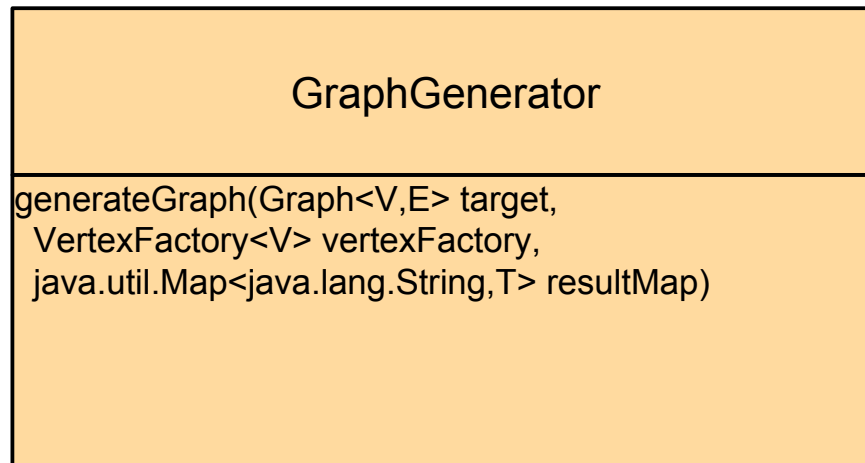
- ▶ `BellmanFordShortestPath` findet kürzeste Wege in gewichteten Graphen
  - Berühmter Algorithmus zum Berechnen von Wegen in Netzen
  - [www.bahn.de](http://www.bahn.de)
  - Logistik, Handlungsreisende, etc.
  - Optimierung von Problemen mit Gewichten
- ▶ `StrongConnectivityInspector` liefert “Zusammenhangsbereiche”,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

# Fabrikmethoden für Knoten und Kanten



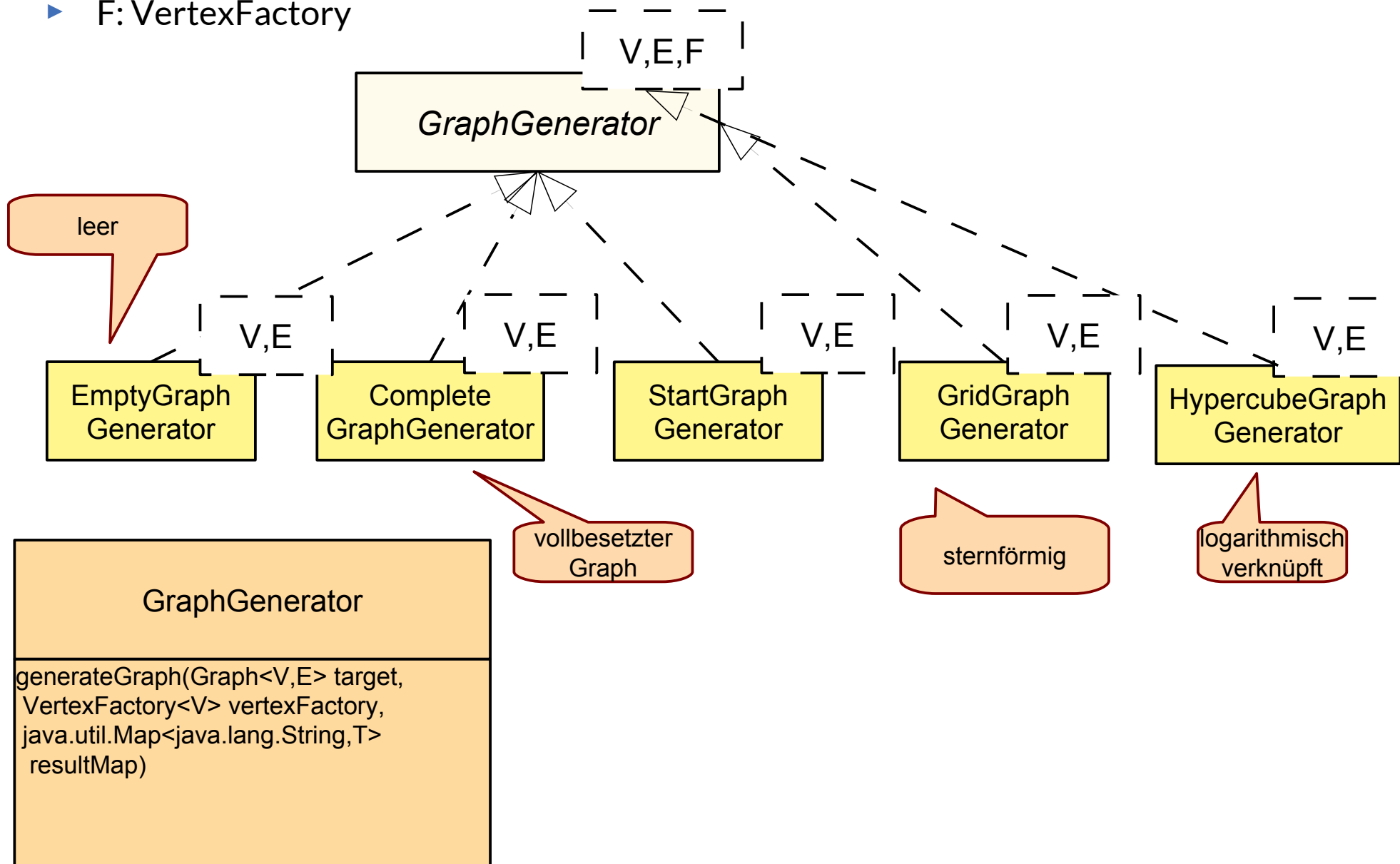
## 25.4.3. Generatoren

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



# Generatoren erzeugen verschiedene Arten von Graphen

► F: VertexFactory



# 25.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?

## 25.E.2 Lern-Exkurs: Die Library GELLY

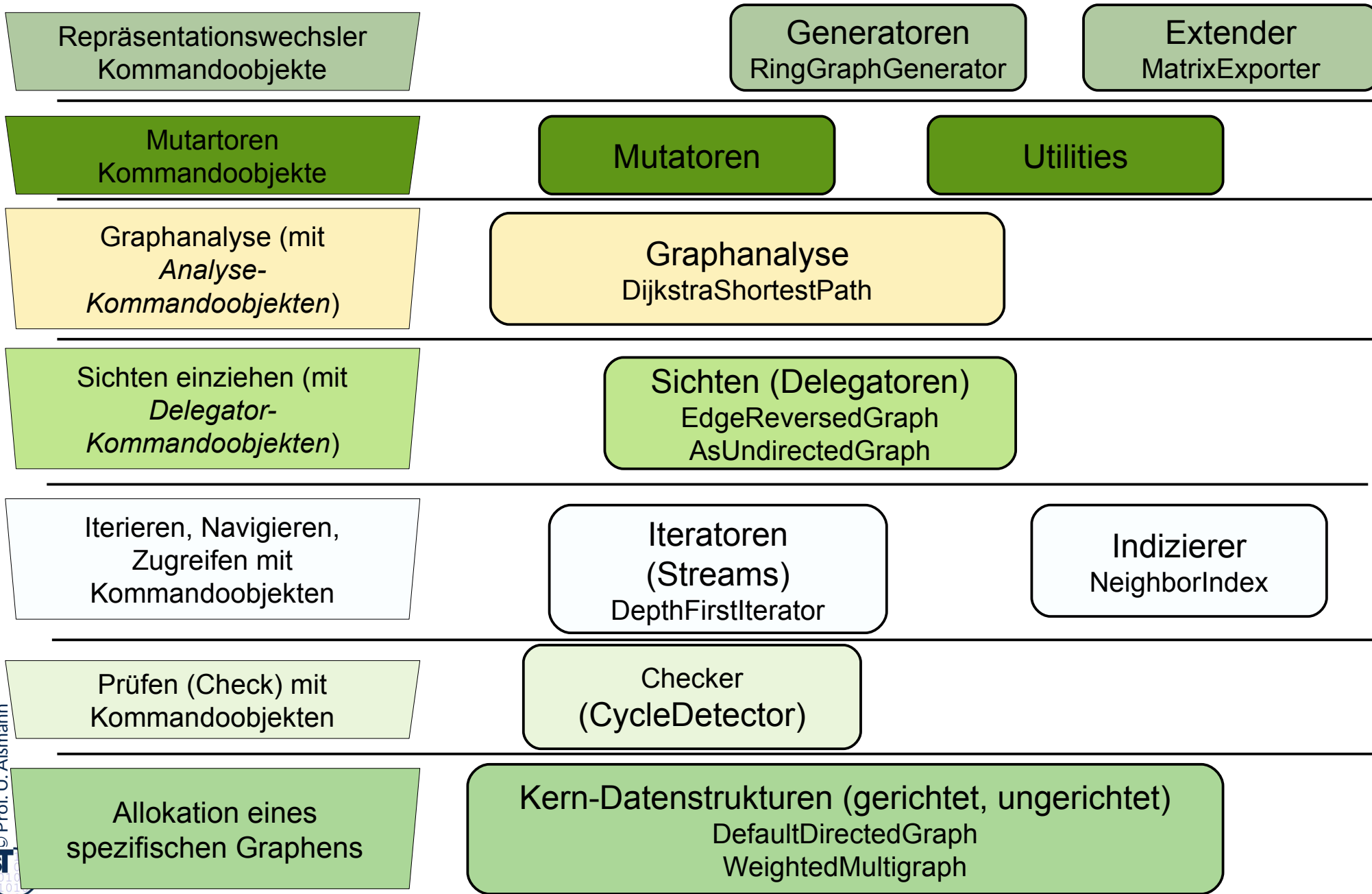
- ▶ Analysieren Sie die Webseite von GELLY
  - <http://gellyschool.com/>
  - [http://ci.apache.org/projects/flink/flink-docs-master/gelly\\_guide.html](http://ci.apache.org/projects/flink/flink-docs-master/gelly_guide.html)
- ▶ Welche Unterschiede gibt es zu JGraphT beim Allozieren von Graphen, Knoten 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 und Assoziationsklassen (Kantenklassen) 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
- ▶ Analysen durch Funktionalobjekte
- ▶ Analysen sind weitreichend nutzbar (s. Vorlesung Softwaretechnologie-II)

# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT



- ▶ Warum benötigt man überhaupt Fabrikmethoden, wenn Java doch schon eine so mächtige Sprache ist?
- ▶ Warum benötigt man überhaupt Kommandoobjekte, wenn Java doch schon eine so mächtige Sprache ist?
- ▶ Wieso lohnt es sich, Iteratoren für Graphenbibliotheken zu nutzen?
- ▶ Wie entwirft man einen Kanal zwischen einem Graphen und einem konsumierenden Aktor, der die Elemente des Graphen eins nach dem anderen “konsumiert”?
- ▶ Wie kann man mit einem Output-Stream einen Graphen persistieren und mit einem Input-Stream ihn wieder lesen?
- ▶ Wieso ist es für den Aufbau von Graphen gut, Generizität zu haben?
- ▶ Entwerfen Sie einen Algorithmus RandomSearch, der durch einen Iterator zufällig die Elemente eines Graphen aufzählt.



## 25. Konstruktion von flexiblen Objektnetzen mit Graphbibliotheken in Java

Prof. Dr. rer. nat. Uwe Aßmann  
Institut für Software- und  
Multimediatechnik  
Lehrstuhl Softwaretechnologie  
Fakultät für Informatik  
Technische Universität Dresden  
Version 19-0.2, 17.05.19

- 1) Implementierungsmuster Fabrikmethode
- 2) Implementierungsmuster Kommando
- 3) Das Graph-Framework JGraphT
  - 1) Aufbau
  - 2) Checker
  - 3) Iteratoren
  - 4) Delegatoren für Sichten
  - 5) Analytoren: Kürzeste Pfade
  - 6) Generatoren



# Obligatorische Literatur

- ▶ JDK Tutorial für J2SE oder J2EE, [www.java.sun.com](http://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](http://ci.apache.org/projects/flink/flink-docs-master/gelly_guide.html)



## Hinweis: Orientierungsplattform Forschung und Praxis auf Bildungsportal Sachsen

3 Softwaretechnologie (ST)

---

- ▶ <https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/12520161286/CourseNode/94468231277848;jsessionId=D463838C980B367A1738D694E15A67A9.opalN8?0>
- ▶

## Ziele dieses Kapitels: Entwurfsmuster in Aktion

- ▶ Eine komplexe Java-Bibliothek aus dritter Hand, eine Graphen-Bibliothek, kennenlernen
  - Graphen als spezielle, flexible Objektnetze verstehen
  - Einsatz mehrerer Entwurfsmuster in einem Framework sehen
  - Die Bibliothek **vereint viele Entwurfsmuster**, die wir kennengelernt haben
    - Fabrikmethoden, Iteratoren und Streams in Anwendung bei Graphen
- ▶ **Anwendungen**
  - Geoinformatik ([www.openstreetmap.org](http://www.openstreetmap.org))
  - Öffentliche Netzinfrastrukturen ("intelligente Netze") wie Eisenbahnnetz, Wasserleitungsnetz, Stromnetz
  - Pläne von Gebäuden, Städten, Verkehrswegen
  - Weltmodelle für Roboter
  - Soziale Netze wie Facebook
- ▶ Generische Graphalgorithmen kennenlernen
  - Delegatoren
  - Generatoren
  - Graphanalysen



- ▶ [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 10-25. Springer, 2001.
  - Facetten von Graphen und wie man sie systematisch, noch besser in einem Framework anordnet
  - Siehe Vorlesung "Design Patterns and Frameworks"



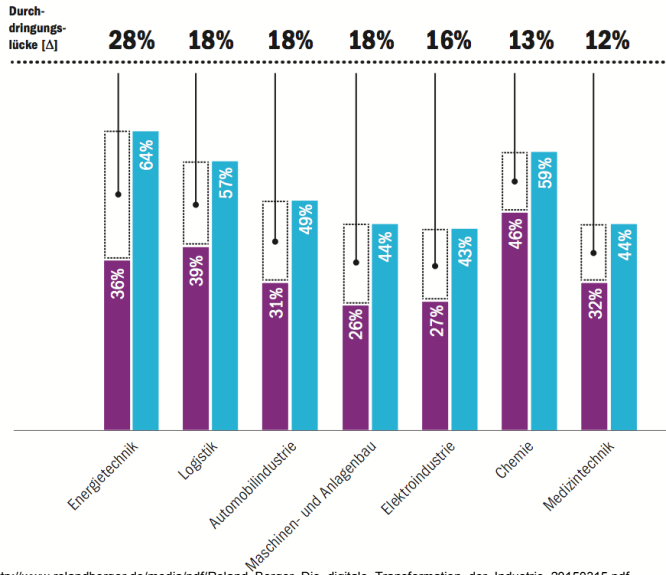


Einschätzung der eigenen digitalen Reife

"Sehr hoch" oder "hoch"

Chancen der Branche aus der Digitalisierung

"Sehr groß" oder "groß"



[http://www.rolandberger.de/media/pdf/Roland\\_Berger\\_Die\\_digitale\\_Transformation\\_der\\_Industrie\\_20150315.pdf](http://www.rolandberger.de/media/pdf/Roland_Berger_Die_digitale_Transformation_der_Industrie_20150315.pdf)

1) Luft- und Raumfahrttechnik aufgrund nicht repräsentativer Zahl von Antworten von Industrievergleich ausgeschlossen  
Quelle: Roland Berger, Umfrage unter 300 Top-Managern der deutschen Wirtschaft



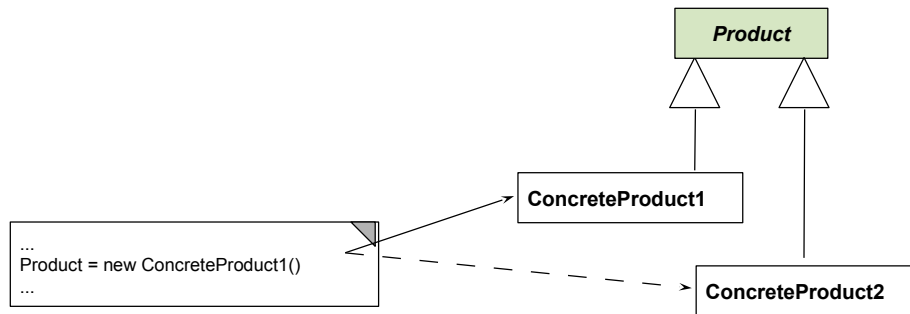
## 25.1 Implementierungsmuster Fabrikmethode (FactoryMethod)

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



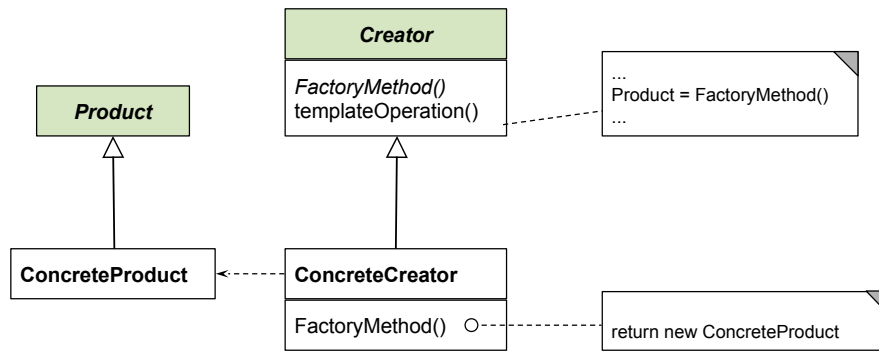
## Problem der Fabrikmethode

- ▶ 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] in Frameworks



# 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();  
    // call a big factory method  
    cr.collect();  
}
```

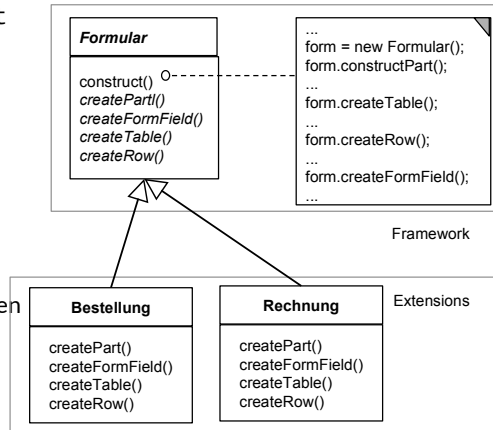
```
// Abstract creator class  
public abstract class Creator {  
    public void collect() {  
        Set mySet = createSet(10);  
        Set secondSet = createSet(20);  
        ...  
    }  
    // factory method  
    public abstract Set createSet(int n);  
}
```

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



# Beispiel Anwendung von FactoryMethod in einem Formular-Framework

- ▶ Framework (Rahmenwerk) für Formulare
  - Klasse Formular hat eine Schablonenmethode `construct` 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

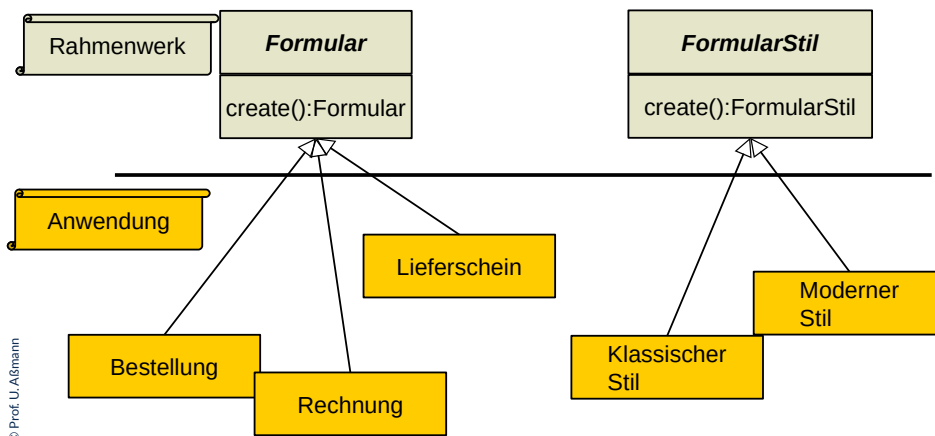
```
// abstract creator class
public abstract class Formular {
    public abstract
        Formular createFormular();
    ...
}
```

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



# Einsatz in Komponentenarchitekturen

- ▶ In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von den Anwendungsschichten aus die Rahmenwerkschicht zu konfigurieren:







## 25.2 Implementierungsmuster Kommandoobjekt (Command)

Zur flexiblen Behandlung von Aktionen



# Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

17 Softwaretechnologie (ST)

Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Anwendungsfunktion 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()`)

Vorteile:

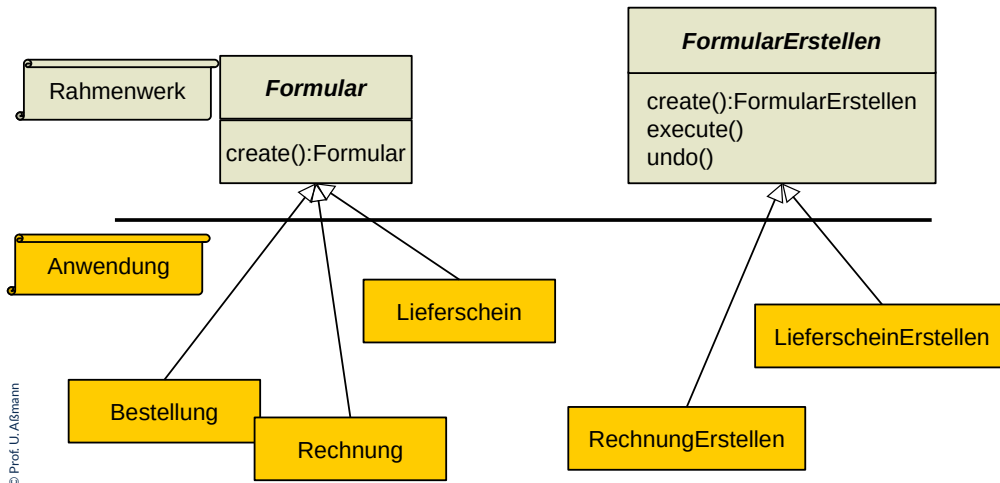
- Man das Funktionalobjekt mit Parametern versehen, herumreichen, und zum Schluss ausführen (*partielle Applikation von Kommandos*)
- Funktionalität wie `undo()`, `redo()`, `persist()`

```
// A command object captures a method
public abstract class Command {
    void execute();
    void takeArgument(<T> arg);
}
public class Undoable extend Command {
    void undo();
}
// Repeatable Command
public class Repeatable extend Undoable {
    void redo();
}
```

```
// Repeatable Command
public class PersistentCommand extends
    Repeatable {
    void persist();
}
```

# Einsatz in Komponentenarchitekturen

- ▶ In Rahmenwerk-Architekturen wird das Kommando-Objekt eingesetzt, um ein Default-Verhalten zur Verfügung zu stellen, das verändert werden kann





Für jedes  
Command  
benötigt man  
eine Factory!

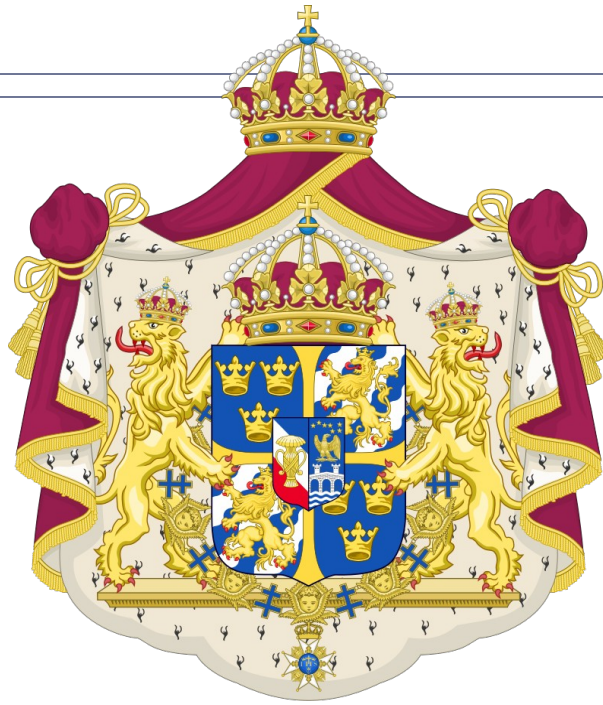


## 25.3 Einsatz von FactoryMethod und Command im JGraphT Framework

Fabriken, Iteratoren, Kommandoobjekte im Großeinsatz

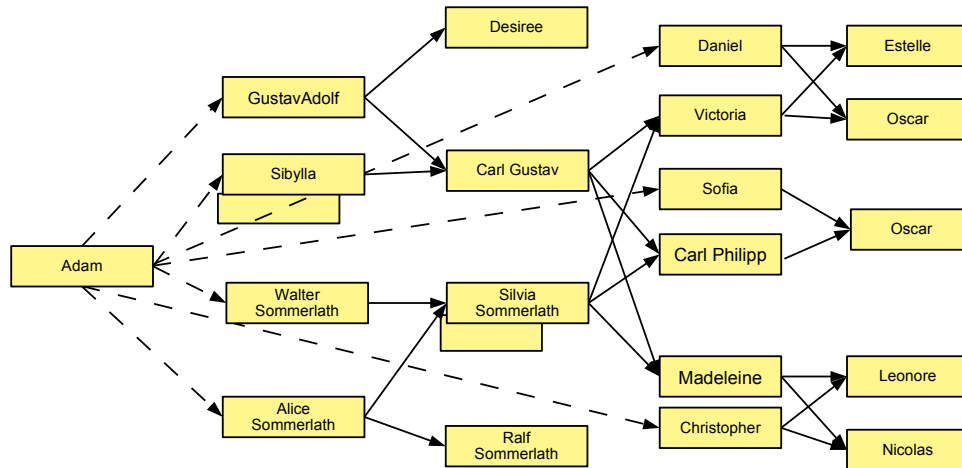


# Beispiel

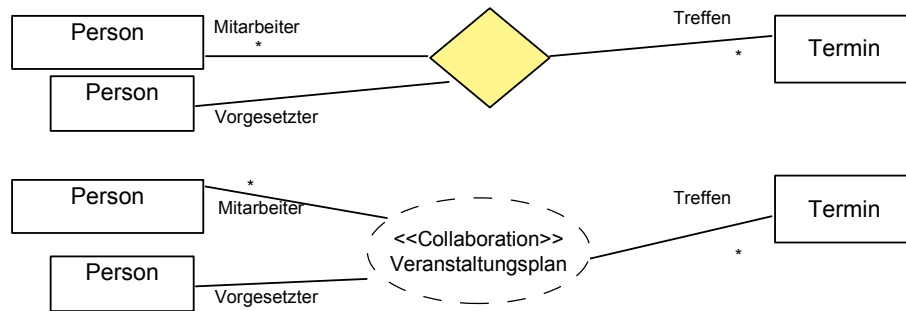


## Beispiel: Verwandtschaftsbeziehungen

- ▶ Familienbeziehungen sind immer azyklisch
- ▶ Die schwedische Königsfamilie als Graph:



- ▶ 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
- ▶ Über einem Graphen kann man Kollaborationen ("Ellipsen") definieren



- Java bietet keine Sprachkonstrukte für Assoziationen und Graphen; Graphen müssen durch ein Framework dargestellt werden
- Es gibt sehr viele Varianten von Graphen; ähnlich zu Collections haben sie viele Facetten
  - [JGraphT] stellt eine Bibliothek mit einer einfachen Abstraktion von Graphen dar
  - Fabrikmethoden, Generics und Iteratoren werden genutzt
- Unterscheidung von speziellen Formen von Graphen
  - Gerichtete azyklische Graphen (directed acyclic graphs, DAG)
  - Multi-Graphen (mit mehreren gleichen Kanten zwischen 2 Knoten)
  - Typisierte Graphen (mit Typen und Attributen)
  - Konstante Graphen, nicht-modifizierbar
  - Kantenobjekte mit Attributen, z.B. gewichtete Graphen
  - Beobachtbare Graphen (mit Observer-Entwurfsmuster)
- Sichten auf Graphen: Inverse Graphen, Untergraphen, Teilgraphen
- Für Graphen auf Objekten, XML Objekten, URLs, Strings, Graphen ...
- Generische Algorithmen auf Graphen
  - Navigation: Pfadsuche, Iteration, Navigation, Abstände
  - Andere: Netzwerkflüsse...



## Ziele einer Graph-Bibliothek

- ▶ Ziel: Management der Kollaboration von flexiblen Objektnetzen
  - Iteration, Navigation, Algorithmen
- ▶ 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	

## <<interface>> *DirectedGraph*<V,E>

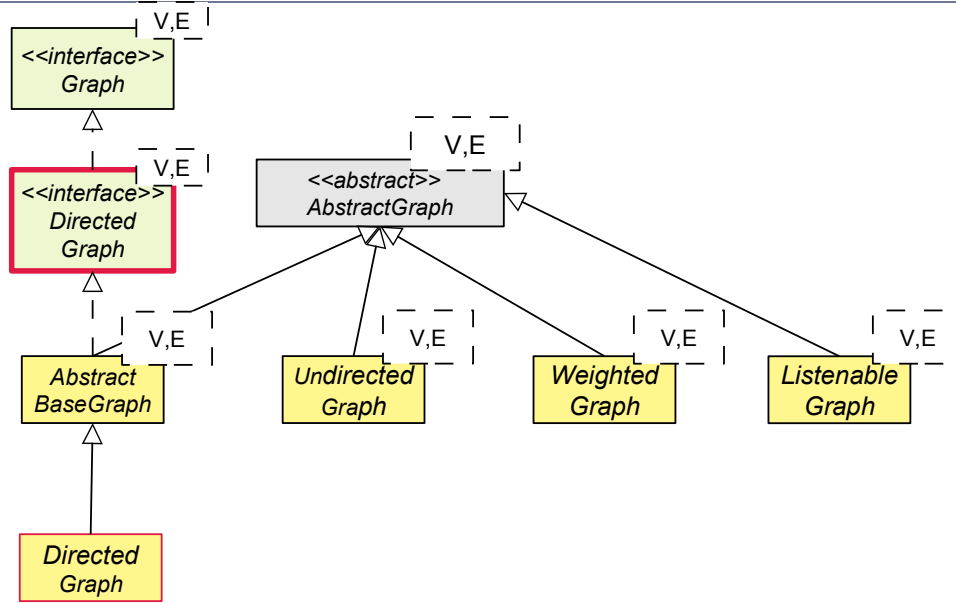
```
// Query-Methoden
java.util.Set<E> edgeSet()
java.util.Set<V> vertexSet()
java.util.Set<E> edgesOf(V vertex)
// Returns a set of all edges touching the specified vertex.
java.util.Set<E> getAllEdges(V sourceVertex, V targetVertex)
E getEdge(V sourceVertex, V targetVertex)
// Returns an edge connecting 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) <
```

## <<interface>> *DirectedGraph*<V,E>

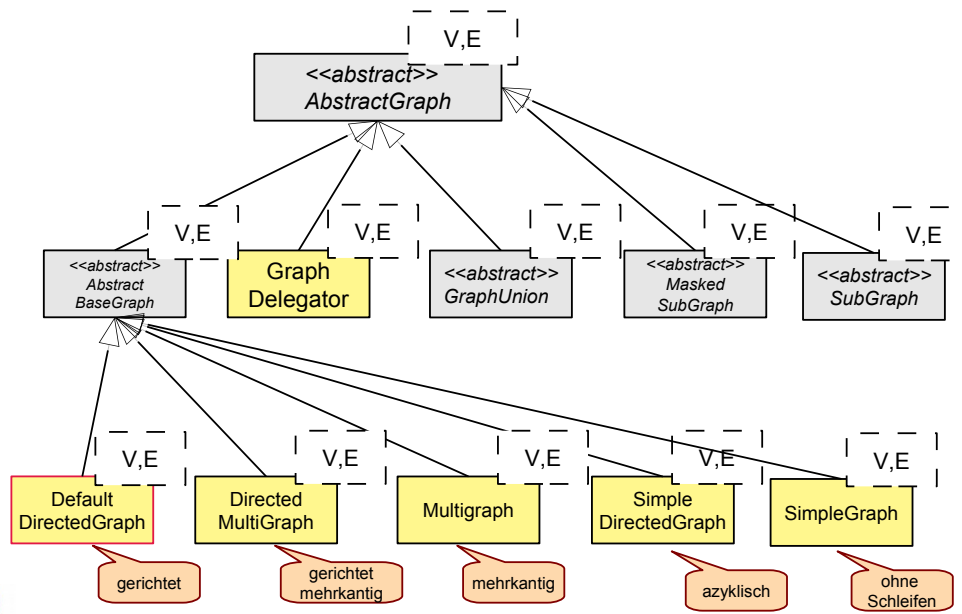
```
// Constructors (doesnt use a factory)
DefaultDirectedGraph(java.lang.Class<? extends E> edgeClass)
    // Creates a new directed graph.
DefaultDirectedGraph(EdgeFactory<V,E> ef)
    // Creates a new directed graph with the specified edge factory.
// Query methods
java.util.Set<E> incomingEdgesOf(V 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 Schnittstellenhierarchie Graph

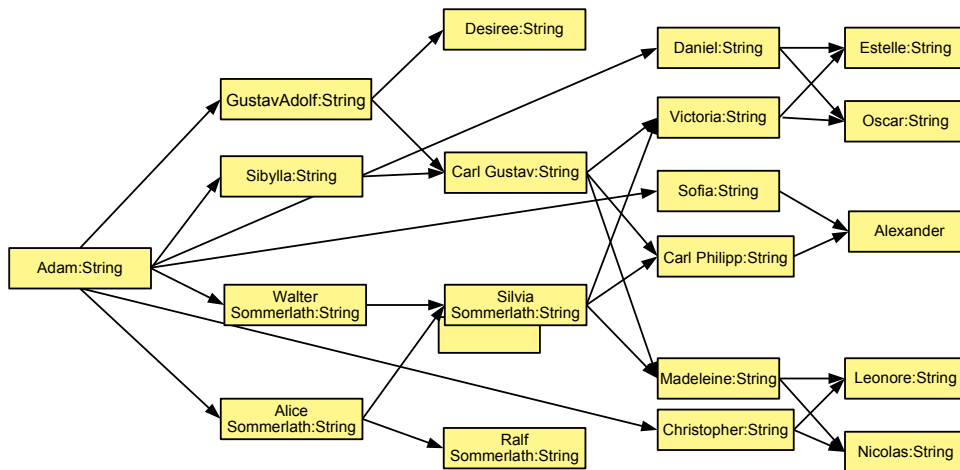


# Die Implementierungshierarchie Graph

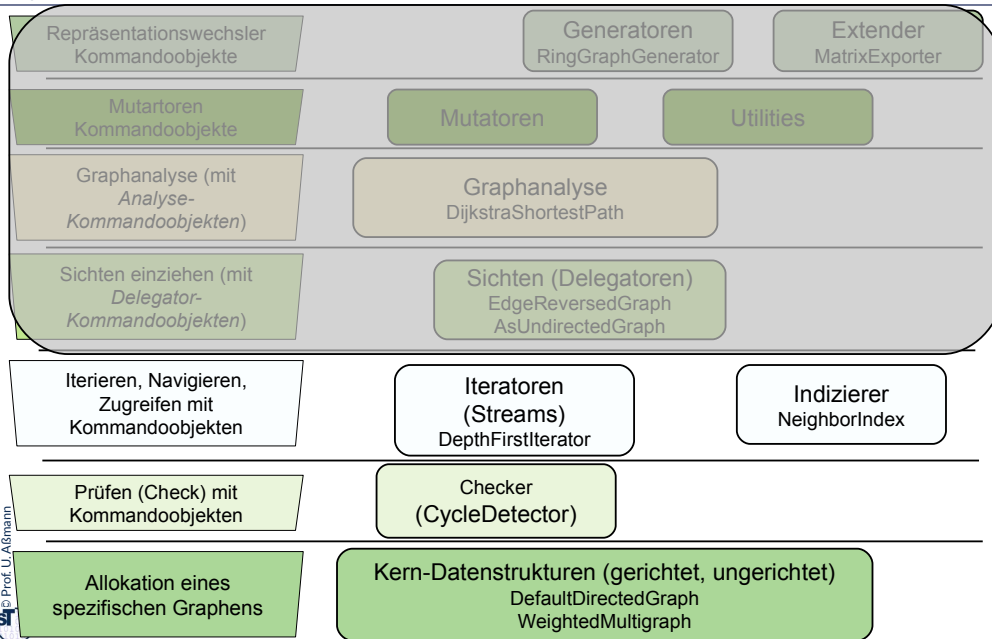


## Beispiel: Verwandtschaftsbeziehungen

- ▶ Familienbeziehungen sind immer azyklisch
- ▶ Die schwedische Königsfamilie als UML-Objektnetz:



# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT





```

// 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";
String estelle = "Estelle";
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);
parentOf.addVertex("Daniel");
parentOf.addVertex("Christopher");
parentOf.addVertex("Sofia");
parentOf.addVertex(estelle);
parentOf.addVertex("Oscar");
parentOf.addVertex("Leonore");
parentOf.addVertex("Nicolas");

```

```

// 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");
parentOf.addEdge("Daniel", "Estelle");
parentOf.addEdge("Victoria", "Estelle");
parentOf.addEdge("Daniel", "Oscar");
parentOf.addEdge("Victoria", "Oscar");
parentOf.addEdge("Madeleine", "Leonore");
parentOf.addEdge("Madeleine", "Nicolas");
parentOf.addEdge("Christopher", "Leonore");
parentOf.addEdge("Christopher", "Nicolas");

/* 1 */ // parentOf.addEdge(estelle, adam);

```

## 25.3.2. Konsistenzprüfung und Navigation mit Check-Kommandos

- ▶ Die meisten generischen Algorithmen von jgraphT sind Kommando-Objekte (Entwurfsmuster Command)
- ▶ `CycleDetector.findCycles()` ist ein Check-Kommando und findet Zyklen im Graphen, jenseits von Selbstkanten
  - Entspricht `execute()`

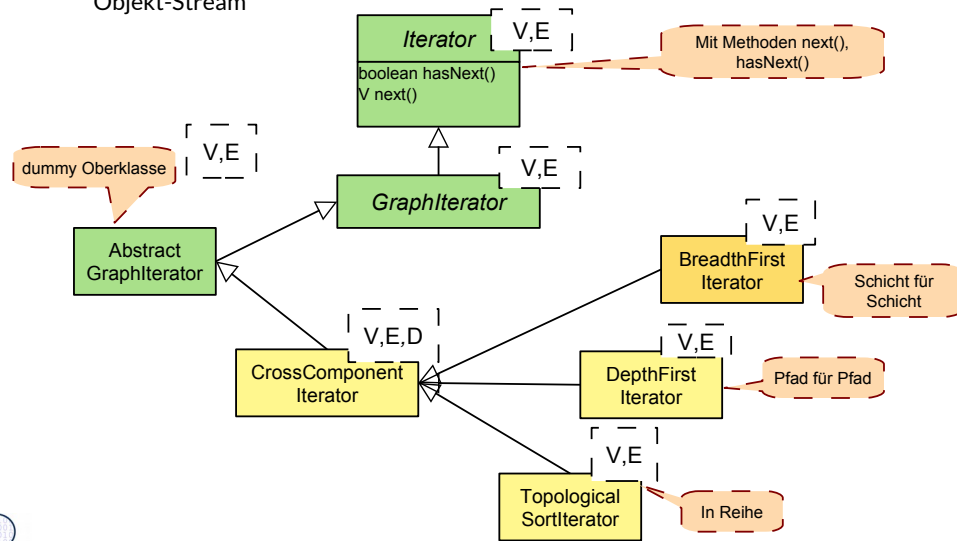
```
// (a) cycle detection in graph parentOf
CycleDetector<String, DefaultEdge> cycleDetector =
    new CycleDetector<String, DefaultEdge>(parentOf);

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



### 25.3.3 Iteratoren laufen Graphen ab

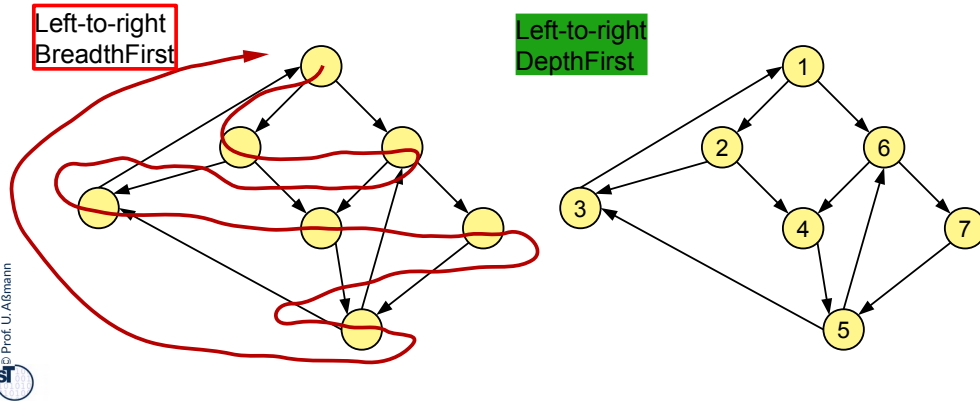
- Man kann mit einem Graphiterador den Graphen ablaufen und seine Knoten ausgeben, ohne seine Struktur zu kennen. Der Iterator verwandelt also den Graph in einen Objekt-Stream



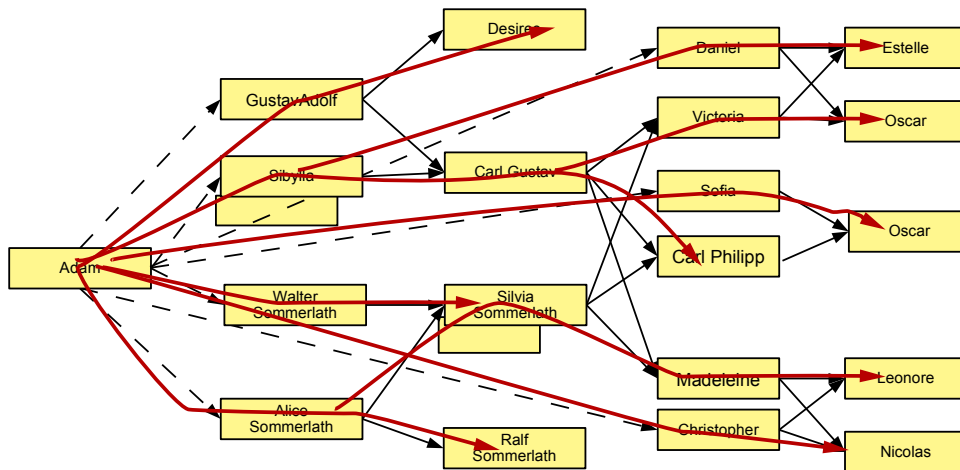
Graphen bilden ein sehr schönes Einsatzfeld für Streams. Die interne Struktur eines Graphen kann völlig verborgen werden, während der Iterator die Knoten (und Kanten) ausgibt.

## Arten von Durchläufen mit Iteratoren

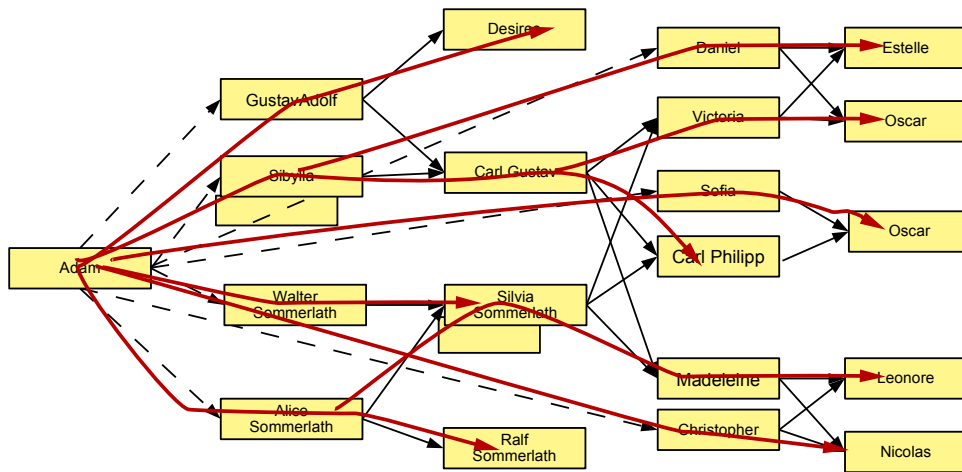
- ▶ `BreadthFirstIterator` läuft über den Graphen in Breitensuche, sozusagen "Schicht für Schicht", und gibt die Knoten aus
- ▶ `DepthFirstIterator` läuft über den Graphen in Tiefensuche, sozusagen "Pfad für Pfad"



# Tiefensuche auf dem azykl. Graphen der Königsfamilie



# Tiefensuche auf dem azykl. Graphen der Königsfamilie



## Examples for 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);
}
```



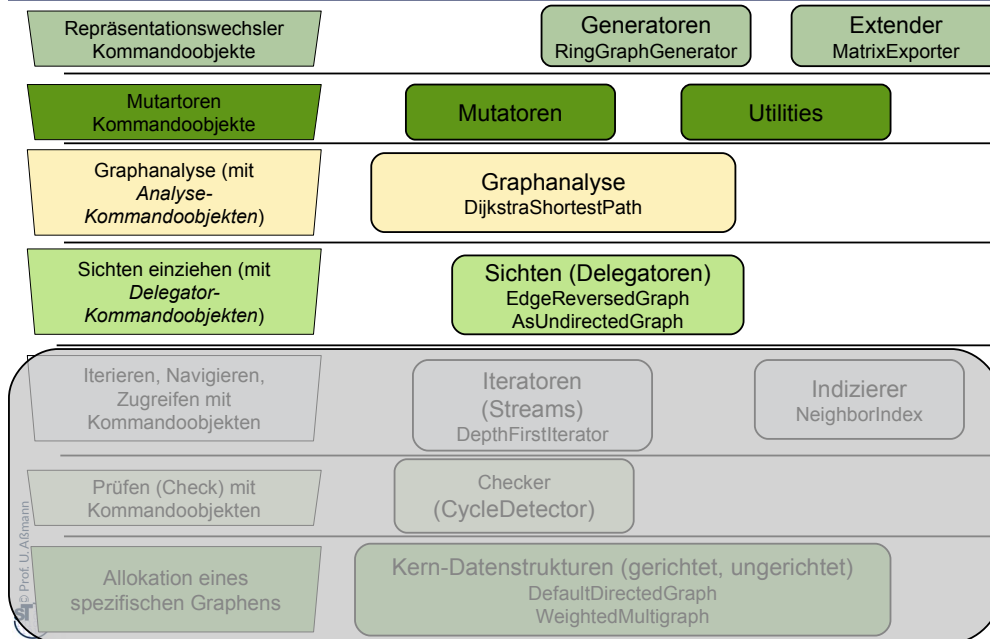
## 25.4 Weitere Schichten im JGraphT Framework

Die Königsklasse bei den Frameworks: Schichtenbildung



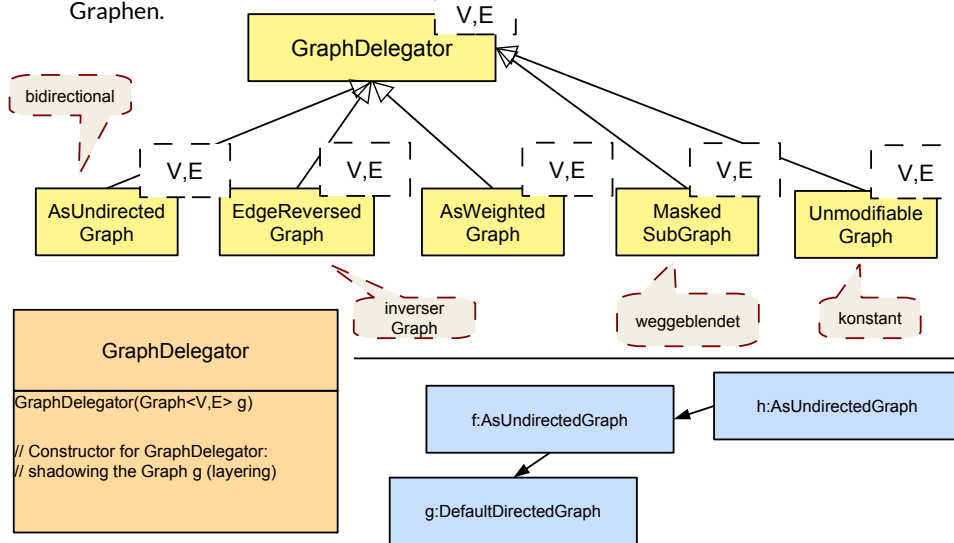


# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT



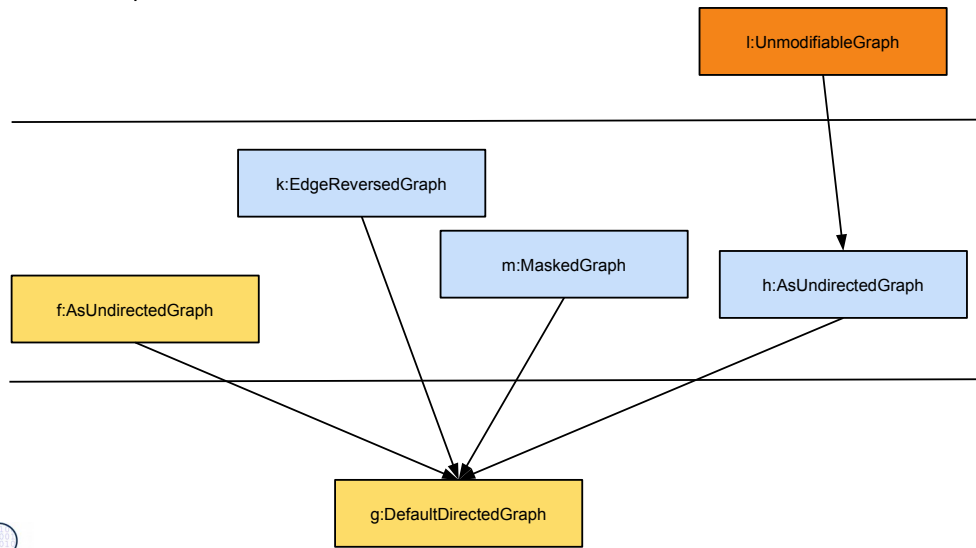
## 25.4.1 Delegatoren erzeugen Sichten

- Def.: Ein **Delegator** ist ein Objekt, das einen Graphen "vorspiegelt" und auf einen Basisgraphen anderen Typs zurückführt. Ein Delegator liefert eine Sicht auf einen Graphen.



## Schichtung von Graphen mit Delegatoren (Layering of Graphs)

- ▶ Was sieht ein Aufrufer (client) eines spezifischen Graphen?
- ▶ Snapshot hier:



## 25.4.2 Analysen in 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 **gerichteten** Graphen 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 ungerichteter Graph kann als Delegator auf einen anderen Graphen erstellt werden (Sicht).
- ▶ Dann kann mit **DijkstraShortestPath** auch auf der Sicht gesucht werden, d.h. der kürzeste Pfad in einem ungerichteten Graphen gesucht werden.

```
// 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();

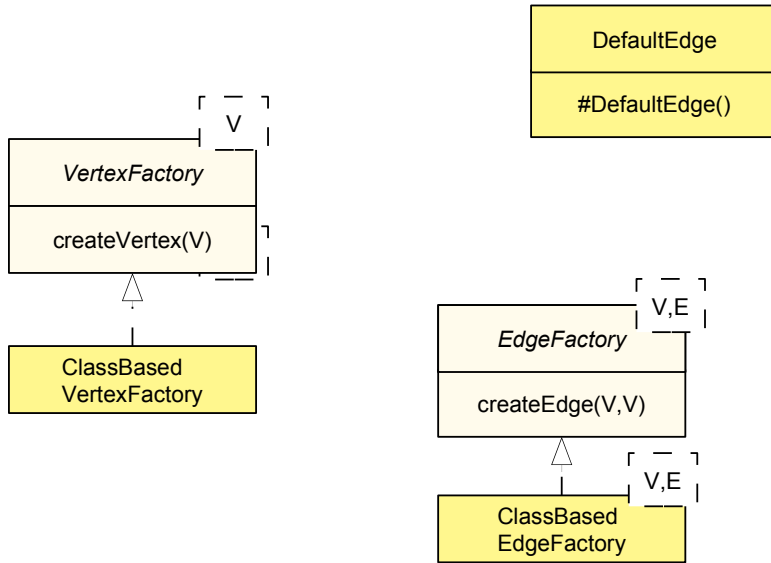
nodeList = Graphs.getPathVertexList(path);
for (String node : nodeList) {
    System.out.println("node: "+node);
}
```



- ▶ `BellmanFordShortestPath` findet kürzeste Wege in gewichteten Graphen
  - Berühmter Algorithmus zum Berechnen von Wegen in Netzen
  - [www.bahn.de](http://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.

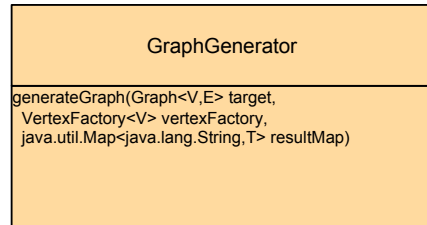


# Fabrikmethoden für Knoten und Kanten



## 25.4.3. Generatoren

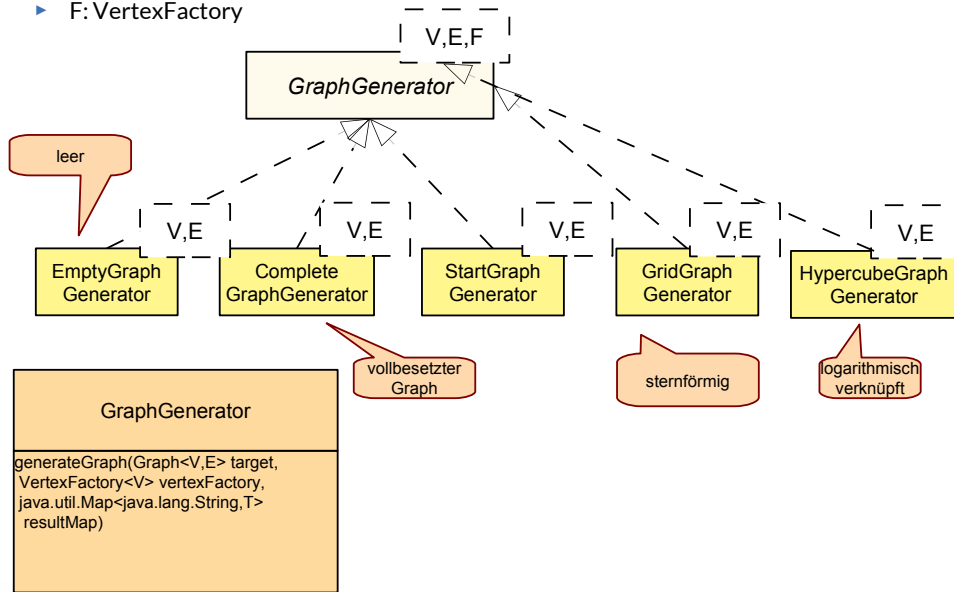
- ▶ Neue Graphen mit anderen Strukturen können aus einem bestehenden Graphen heraus erzeugt werden





# Generatoren erzeugen verschiedene Arten von Graphen

► F: VertexFactory



## 25.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?



## 25.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](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?

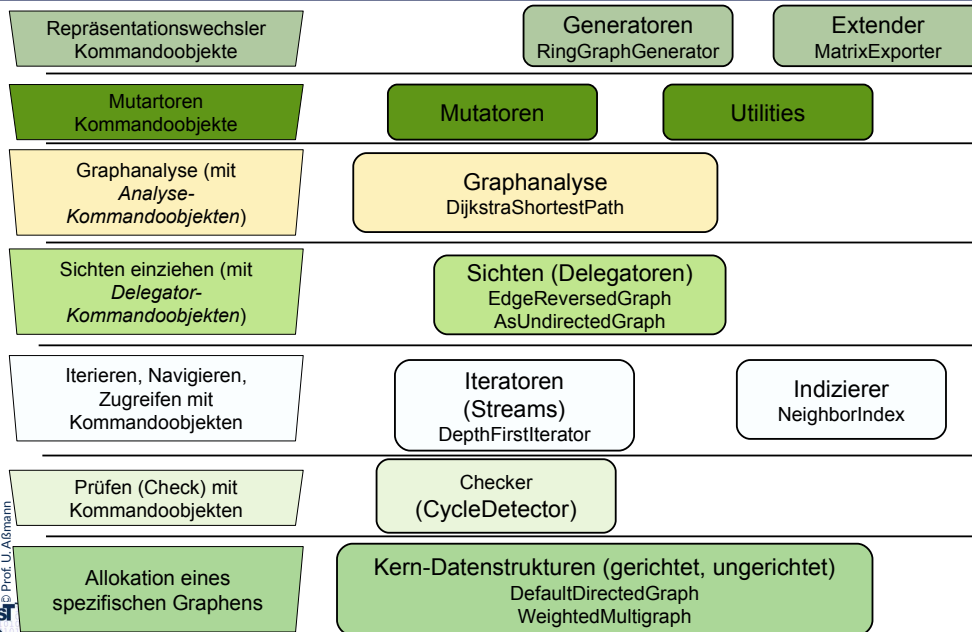


## Was haben wir gelernt?

- ▶ Objektnetze, die in einem UML-Modell mit Assoziationen und Assoziationsklassen (Kantenklassen) 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
- ▶ Analysen durch Funktionalobjekte
- ▶ Analysen sind weitreichend nutzbar (s. Vorlesung Softwaretechnologie-II)



# Kategorien von Graphalgorithmen und Kommandoobjekten in JGraphT



## The End

- ▶ Warum benötigt man überhaupt Fabrikmethoden, wenn Java doch schon eine so mächtige Sprache ist?
- ▶ Warum benötigt man überhaupt Kommandoobjekte, wenn Java doch schon eine so mächtige Sprache ist?
- ▶ Wieso lohnt es sich, Iteratoren für Graphenbibliotheken zu nutzen?
- ▶ Wie entwirft man einen Kanal zwischen einem Graphen und einem konsumierenden Aktor, der die Elemente des Graphen eins nach dem anderen “konsumiert”?
- ▶ Wie kann man mit einem Output-Stream einen Graphen persistieren und mit einem Input-Stream ihn wieder lesen?
- ▶ Wieso ist es für den Aufbau von Graphen gut, Generizität zu haben?
- ▶ Entwerfen Sie einen Algorithmus RandomSearch, der durch einen Iterator zufällig die Elemente eines Graphen aufzählt.

