

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 13-1.1, 13.05.13

- 1) Entwurfsmuster Fabrikmethode
- 2) Das Graph Framework JGraphT



Obligatorische Literatur

2

- ▶ 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>

Nicht-obligatorische Literatur

3

- ▶ [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

4

- ▶ Eine Graphen-Bibliothek kennenlernen
- ▶ Graphen als spezielle Kollaborationen kennenlernen
- ▶ Fabriken, Iteratoren und Streams in der Anwendung bei Graphen
- ▶ Generische Graphalgorithmen kennenlernen
 - Generatoren
 - Graphanalysen



23.1 Implementierungsmuster Fabrikmethode (FactoryMethod)

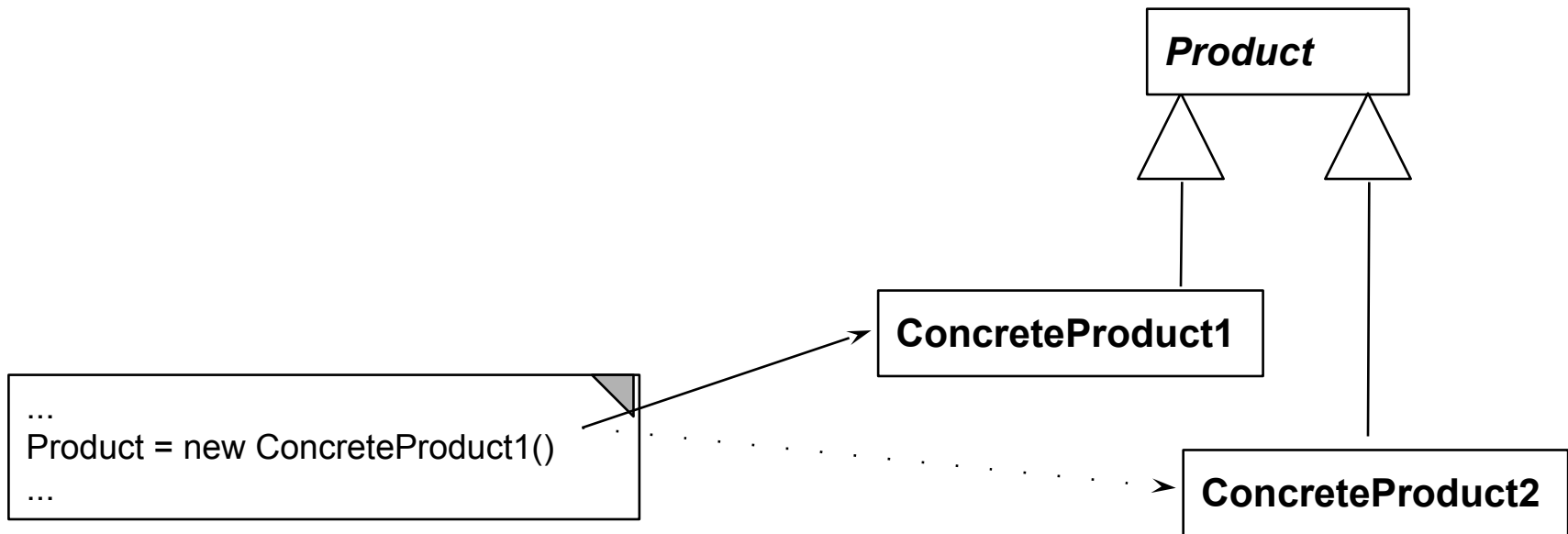
5

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

Problem der Fabrikmethode

6

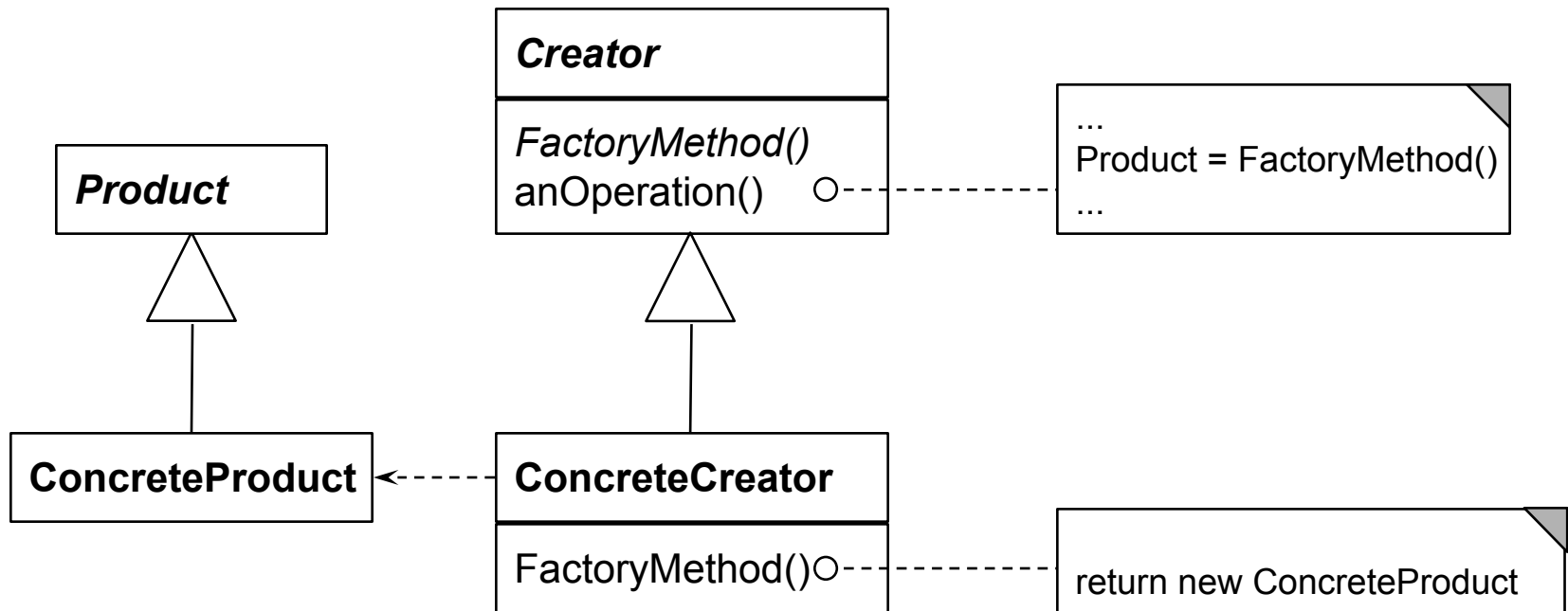
- ▶ Wie variiert man die Erzeugung für eine polymorphe Hierarchie von Produkten?
- ▶ Problem: Konstruktoren sind nicht polymorph!



Struktur Fabrikmethode

7

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



Fabrikmethode (Factory Method)

8

- ▶ Allokatoren in einer abstrakten Oberklasse nennt man *Fabrikmethoden* (*polymorphe Konstruktoren*)
 - Konkrete Unterklassen spezialisieren den Allokator

```
// Abstract creator class
public abstract class Creator {
    // factory method
    public abstract Set createSet(int n);
}
```

```
public class Client {
... Creator cr = [.. subclass ]..
    public void collect() {
        Set mySet = cr.createSet(10);
        ....
    }
}
```

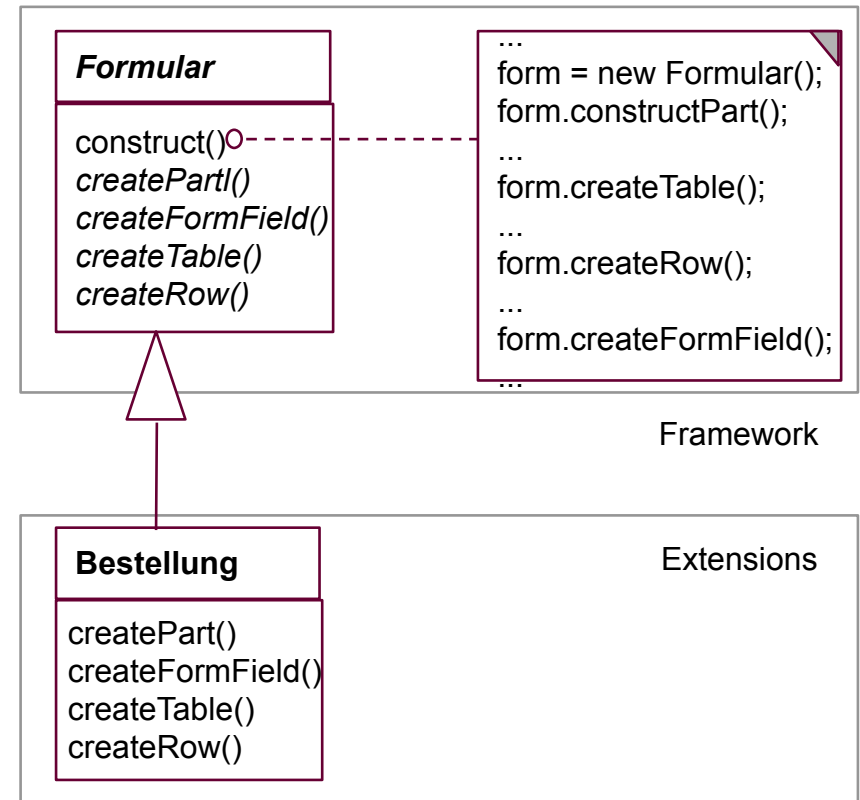
```
// Concrete creator class
public class ConcreteCreator
    extends Creator {
    public Set createSet(int n) {
        return new ListBasedSet(n);
    }
    ...
}
```



Beispiel FactoryMethod für Formulare

9

- ▶ 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

10

- ▶ Bilde createFormular() als Fabrikmethode aus

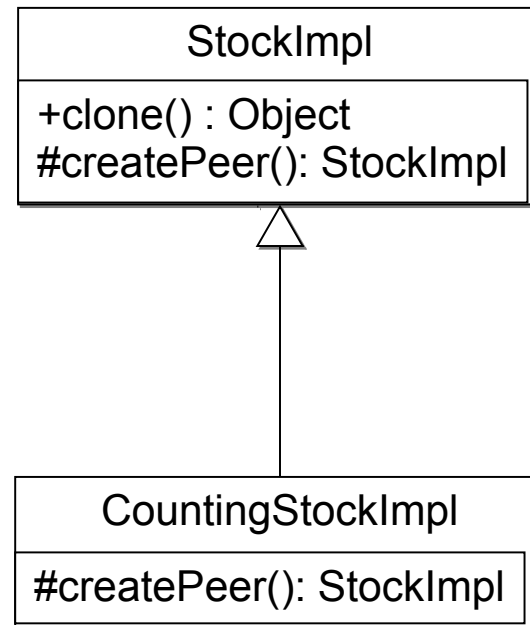
```
// abstract creator class
public abstract class Formular {
    public abstract
        Formular createFormular();
    ...
}
```

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

Factory Method im SalesPoint-Rahmenwerk

11

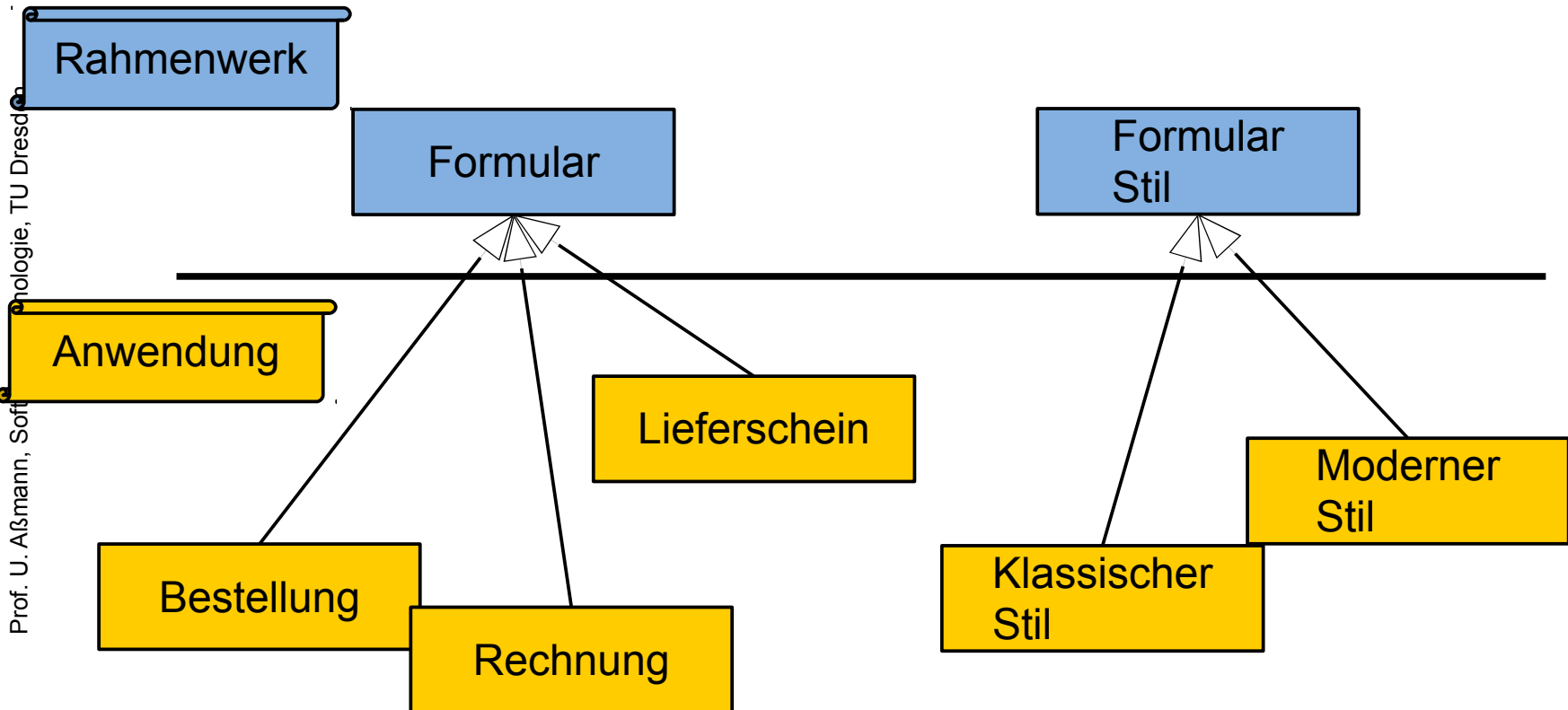
- ▶ Anwender von SalesPoint verfeinern die StockImpl-Klasse, die ein Produkt des Warenhauses im Lager repräsentiert
 - z.B. mit einem CountingStockImpl, der weiß, wieviele Produkte noch da sind



Einsatz in Komponentenarchitekturen

12

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



23.2 Das JGraphT Framework

13

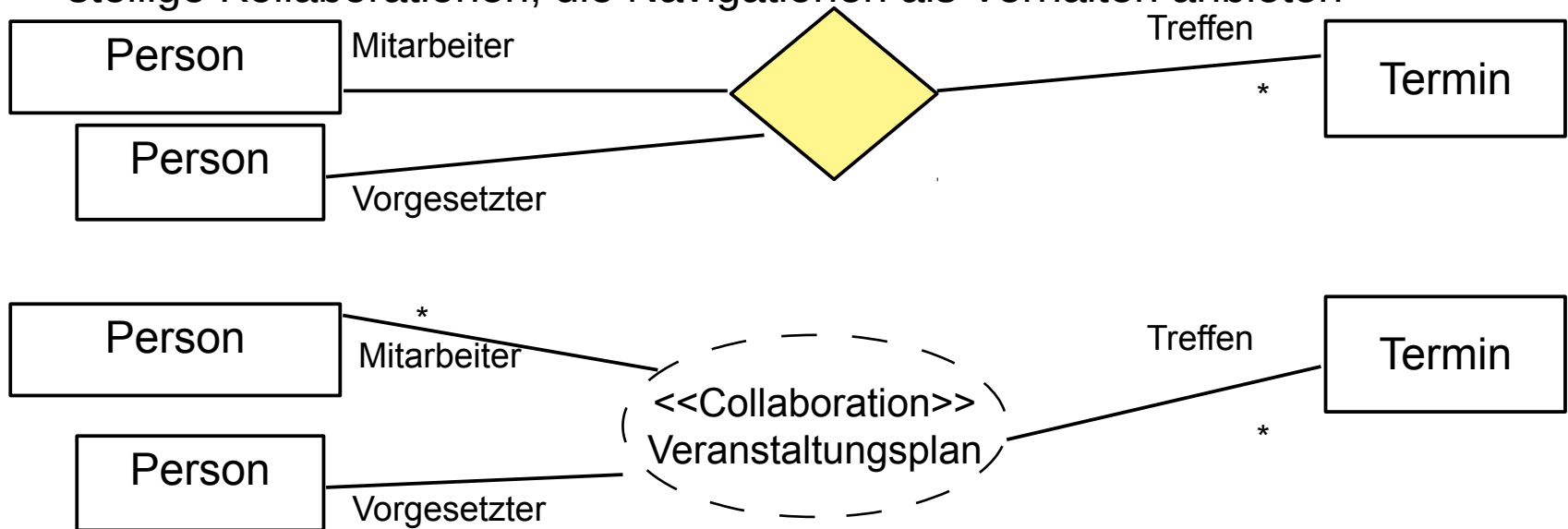
Fabriken überall



Graphen als spezielle Kollaborationen

14

- ▶ UML bietet das Sprachkonstrukt “Assoziation”. 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

15

- ▶ 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...

Klassifikationsfacetten von Graphen

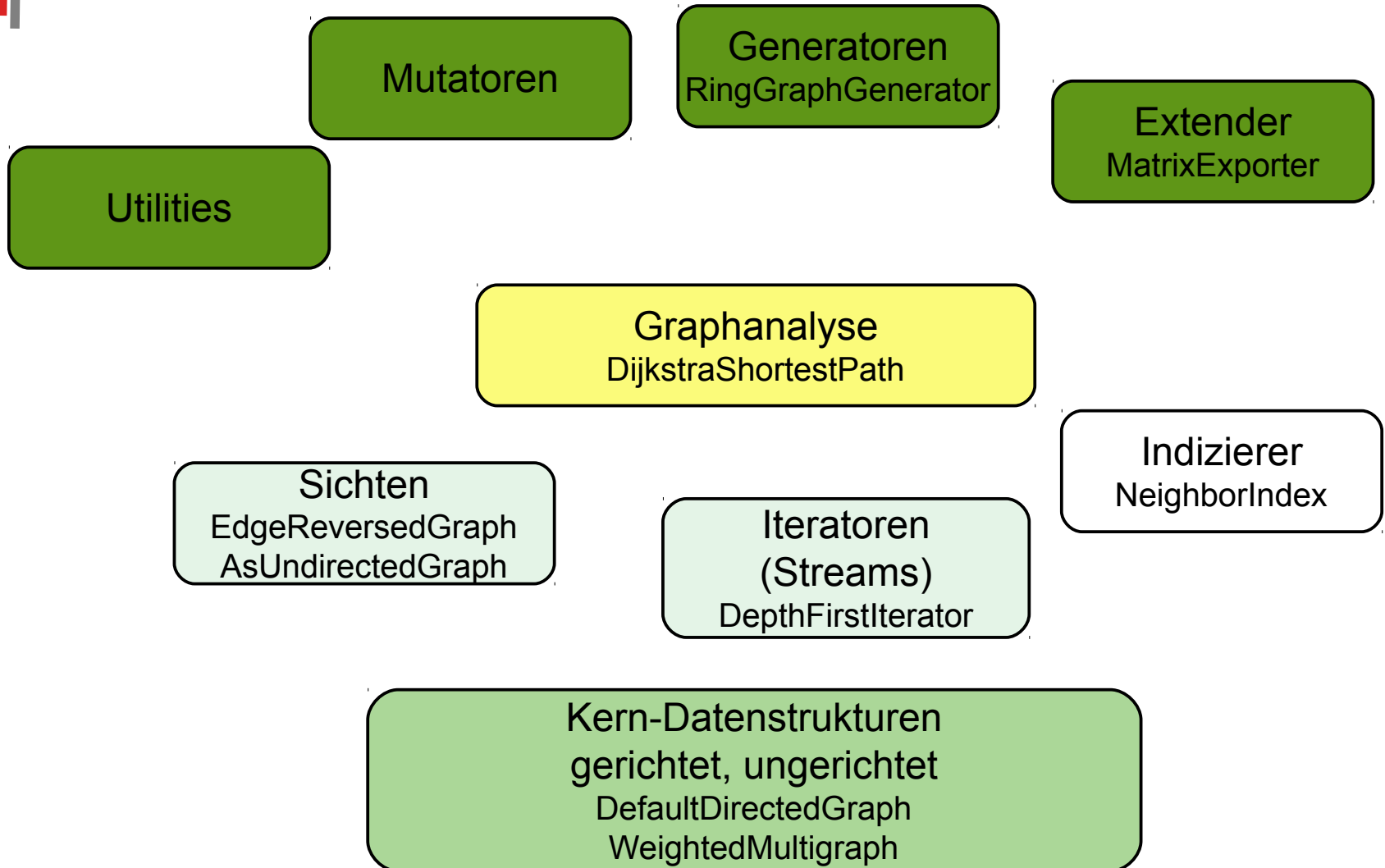
16

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

Kategorien von Graphalgorithmen in JGraphT

17

▶ mit Beispielen



Graphen dienen dem Aufbau von Objektnetzen

18

Allokation eines Graphen spezifischer
Kategorie

Knoten addieren

Kanten addieren

Sichten einziehen

Graphanalyse

<<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

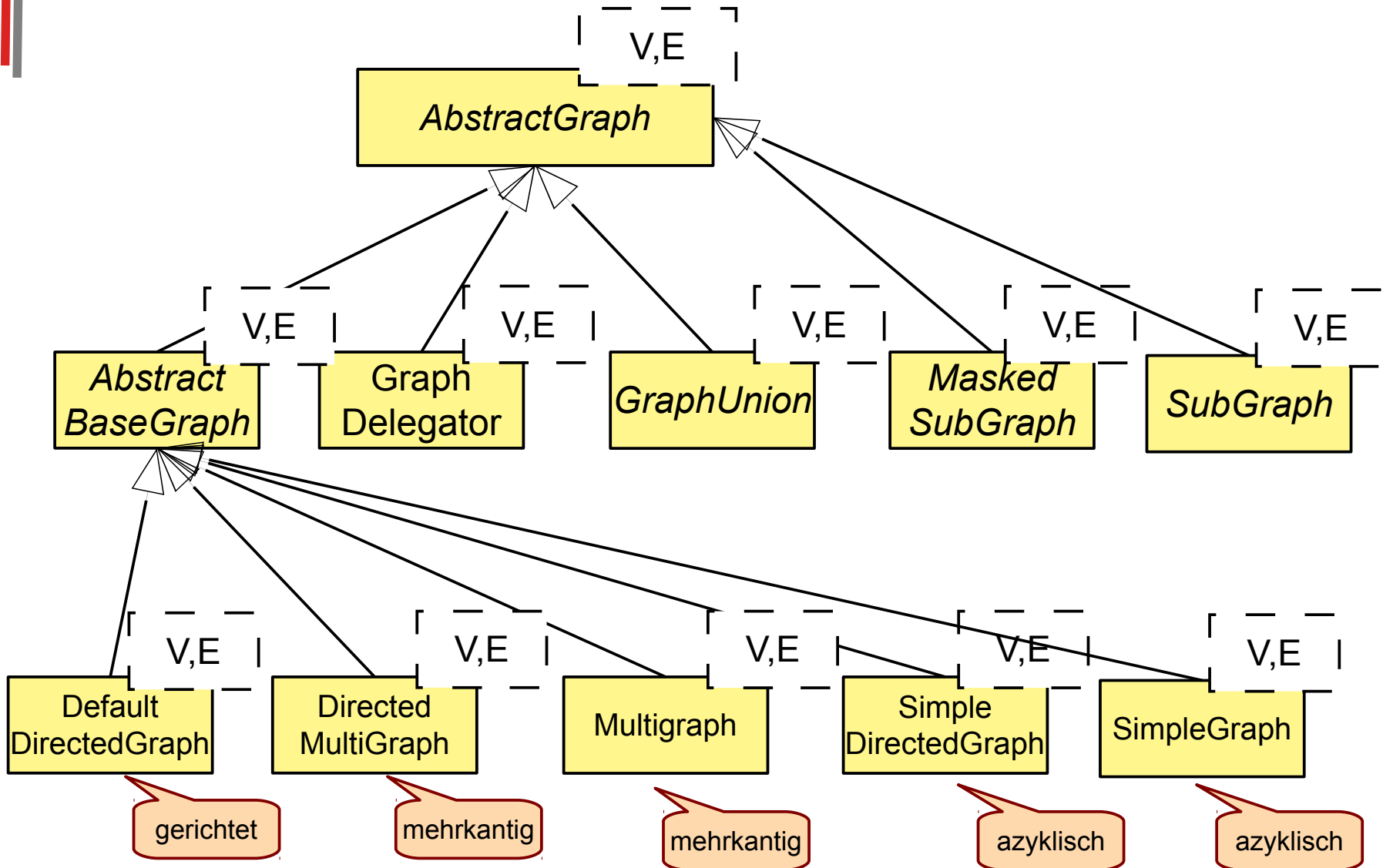
20

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 Implementierungshierarchie Graph

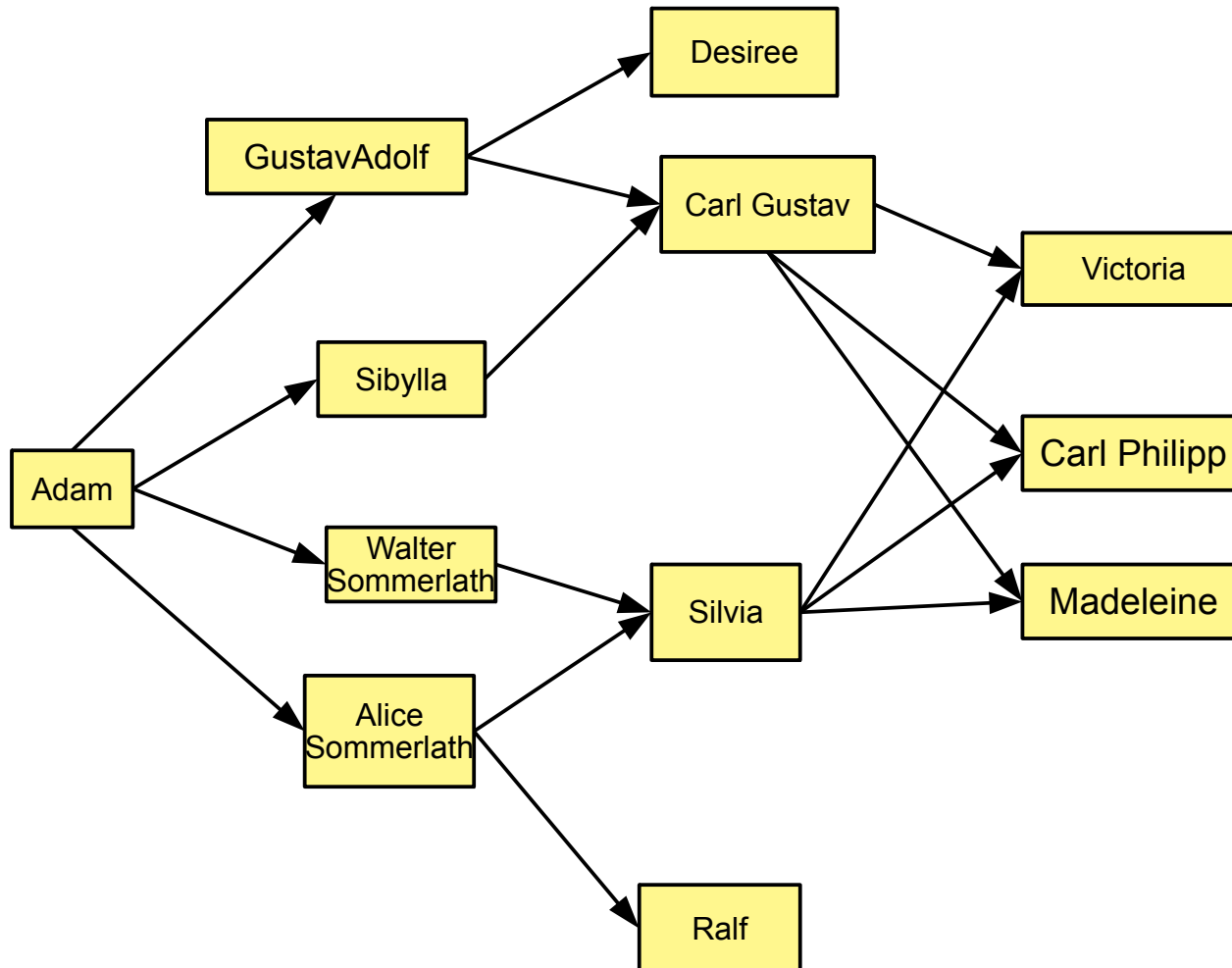
22



Beispiel: Verwandtschaftsbeziehungen

23

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



Aufbau gerichteter Graphen

24

```
// 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);
```


Konsistenzprüfung und Navigation

25

- ▶ Die meisten generischen Algorithmen von jgraphT sind Funktionalobjekte (Entwurfsmuster Command)
- ▶ `CycleDetector` findet Zyklen im Graphen, jenseits von Selbstkanten

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

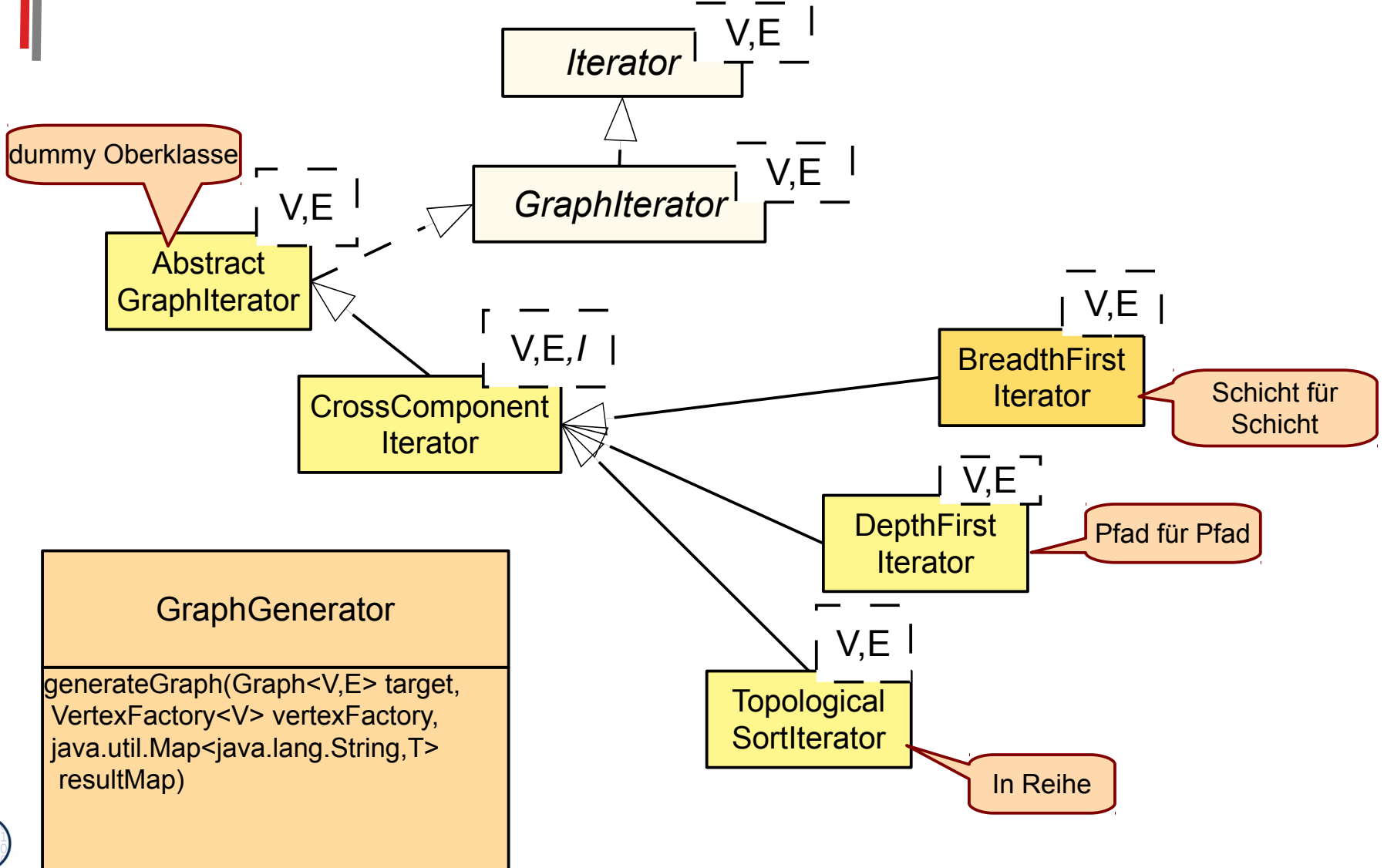
```
// (b) breadth-first iteration in graph parentOf

System.out.println("breadth first enumeration: ");
BreadthFirstIterator<String,DefaultEdge> bfi = new BreadthFirstIterator(parentOf);
for (String node = bfi.next(); bfi.hasNext(); node = bfi.next()) {
    System.out.println("node: "+node);
}
```

Iteratoren laufen Graphen ab

26

- Man kann den Graphen ablaufen, ohne seine Struktur zu kennen

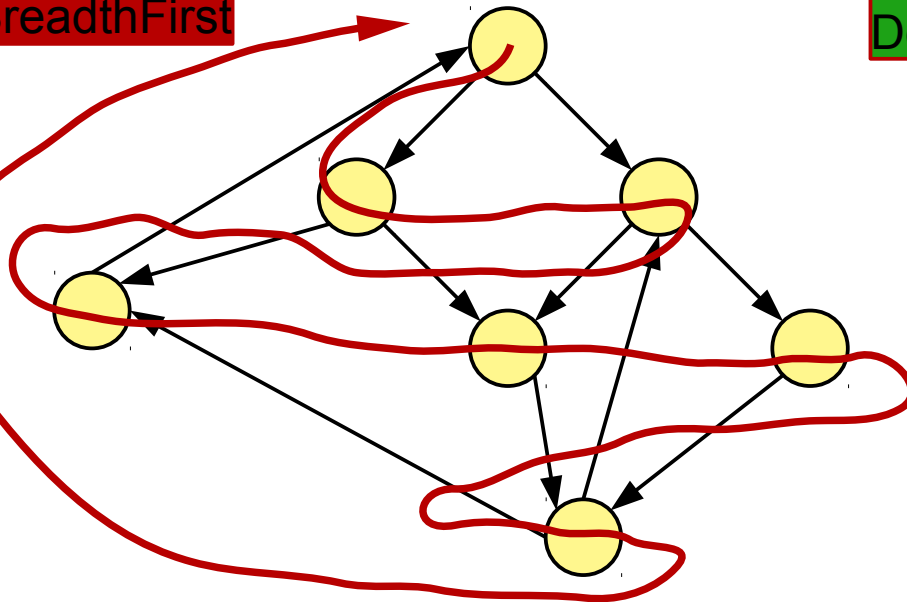


Arten von Durchläufen mit Iteratoren

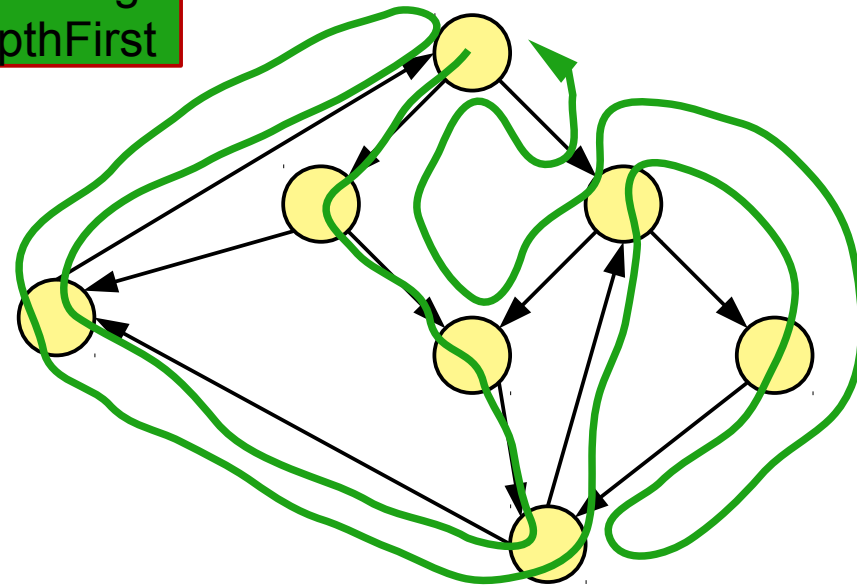
27

- ▶ `BreadthFirstIterator` läuft über den Graphen in Breitensuche, sozusagen “Schicht für Schicht”
- ▶ `DepthFirstIterator` läuft über den Graphen in Tiefensuche, sozusagen “Pfad für Pfad”

Left-to-right
BreadthFirst



Left-to-right
DepthFirst



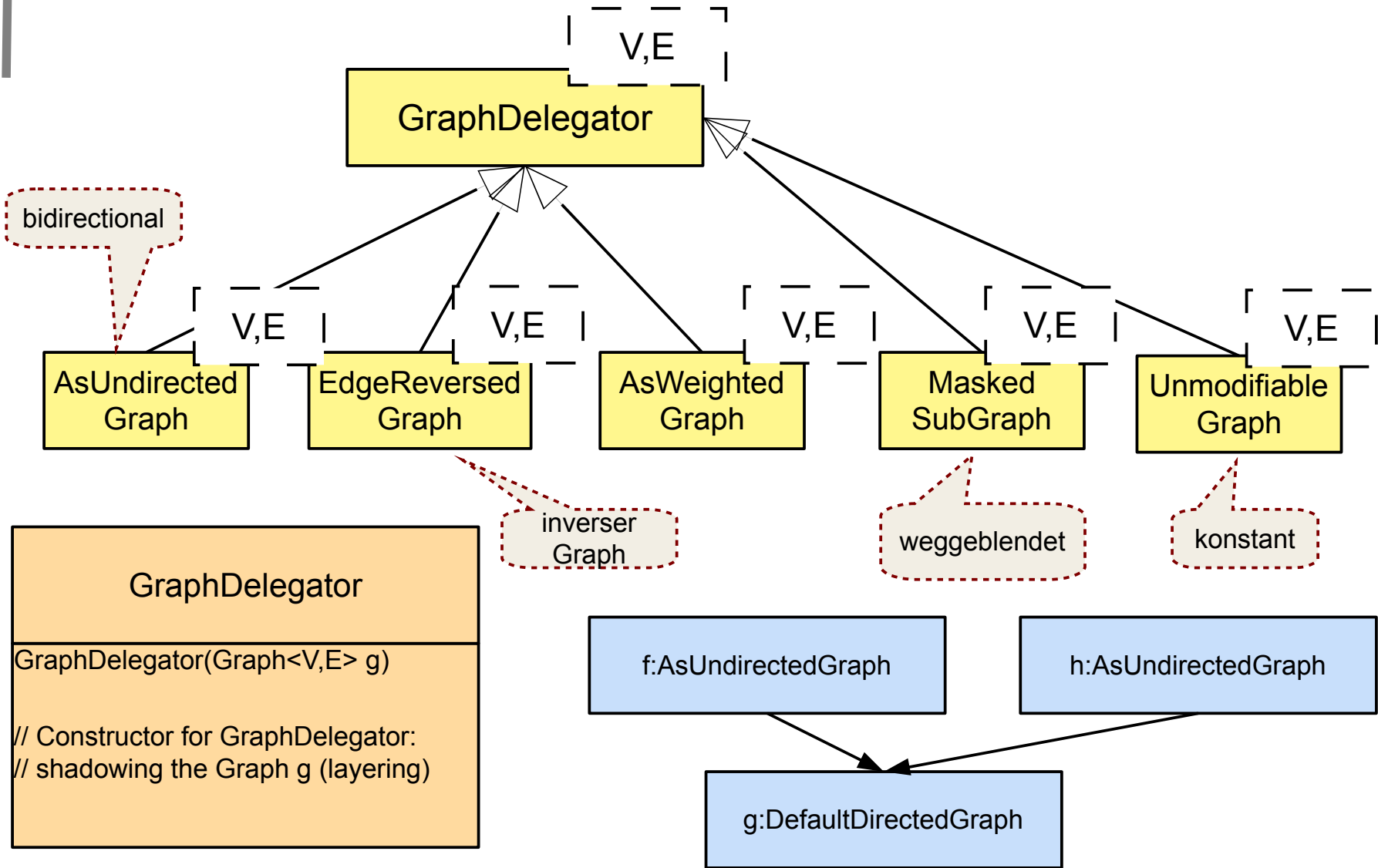
Finden kürzester Pfade

28

- ▶ Dijkstra's algorithmus findet zwischen 2 Knoten den kürzesten Pfad

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

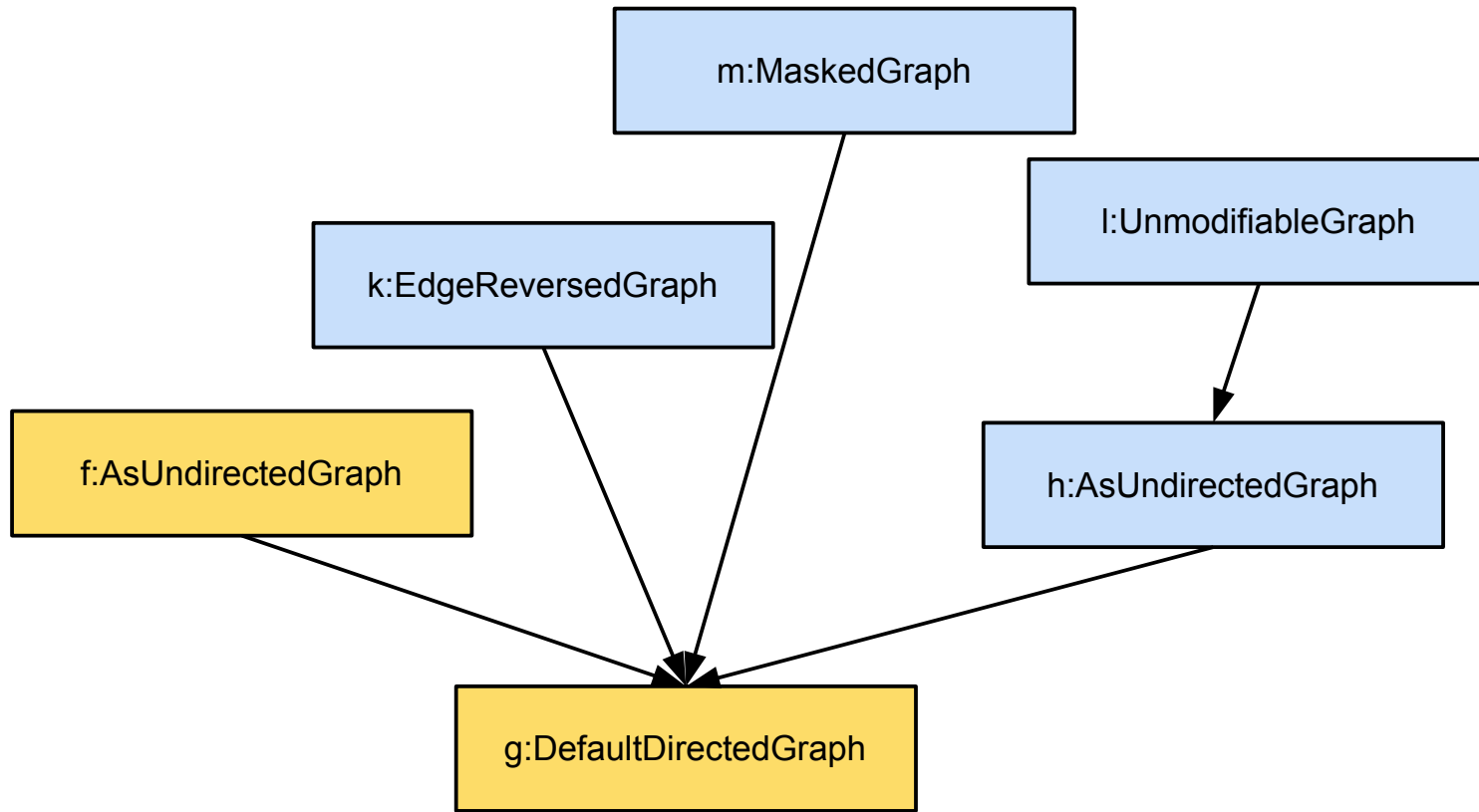
Delegatoren erzeugen Sichten



Schichtung von Graphen (Layering of Graphs)

30

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



Finden kürzester Pfade im ungerichteten Graphen (Sicht)

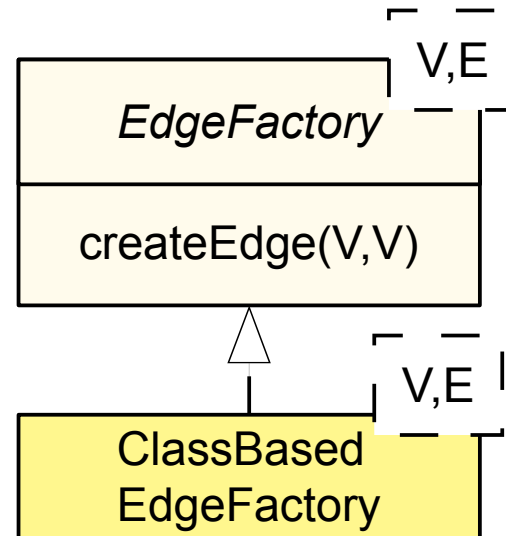
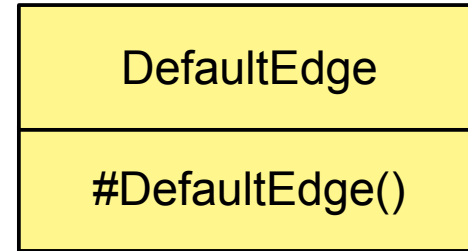
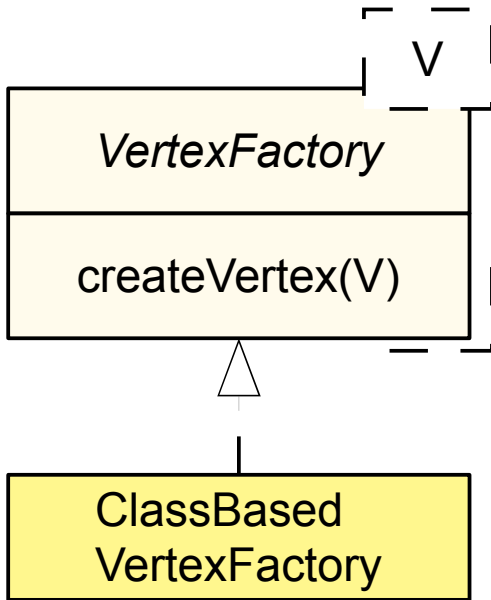
31

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

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

Fabriken für Knoten und Kanten

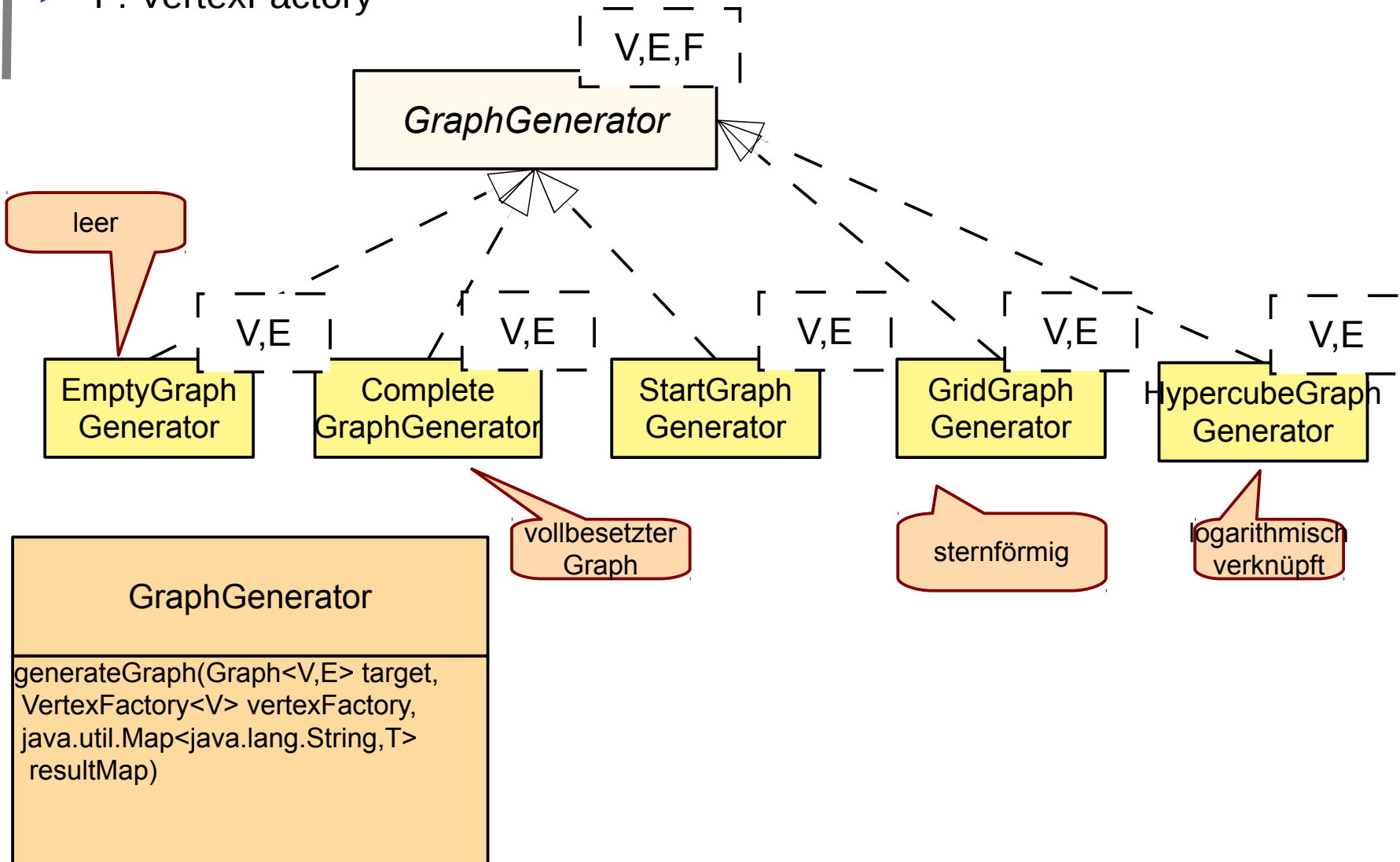
32



Generatoren erzeugen verschiedene Arten von Graphen

33

► F: VertexFactory



- ▶ `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.

Was haben wir gelernt?

35

- ▶ 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
- ▶ Analysen durch Funktionalobjekte
- ▶ Analysen sind weitreichend nutzbar (s. Vorlesung Softwaretechnologie-II)