

Applications Of Concept Lattices to Code Inspection and Review



Uri Dekel

Department of Computer Science
Technion, Haifa, Israel

M.Sc. research supervised by Dr. Yossi Gil



Objective

- Understanding individual Java classes.
 - For using a third-party class as a black box.
 - Groups of *public* methods with related responsibilities.
 - Purpose of every *public* method.
 - Pre- and post- conditions.
 - For reverse-engineering.
 - Class structure.
 - Separation of concerns.
 - Implementation of all methods.
 - For code review.
 - Code style and correctness.



Formal Concept Analysis

- A mathematical classification technique.
 - Performed on a binary relation (a *context*) between a set of *instances* and a set of *features*.
 - Results in *concepts*, which are presented in a *concept lattice*.
- Used in:
 - Automatic class hierarchy construction.
 - Automatic module discovery.
 - Managing multiple configurations.



Concepts

- $\langle I, F, R \rangle$ is a context.
- $C = \langle I' \subseteq I, F' \subseteq F \rangle$ is a concept if:
 - The maximal set of features shared by all instances in I' is F' .
 - The maximal set of instances that own every feature in F' is I' .
- Alternative definition:
 - Each instance or feature appears in a single concept.
 - Instances that own the same set of features appear in the same concept.
- Concepts are presented in a concept lattice.



Field-Accesses Context

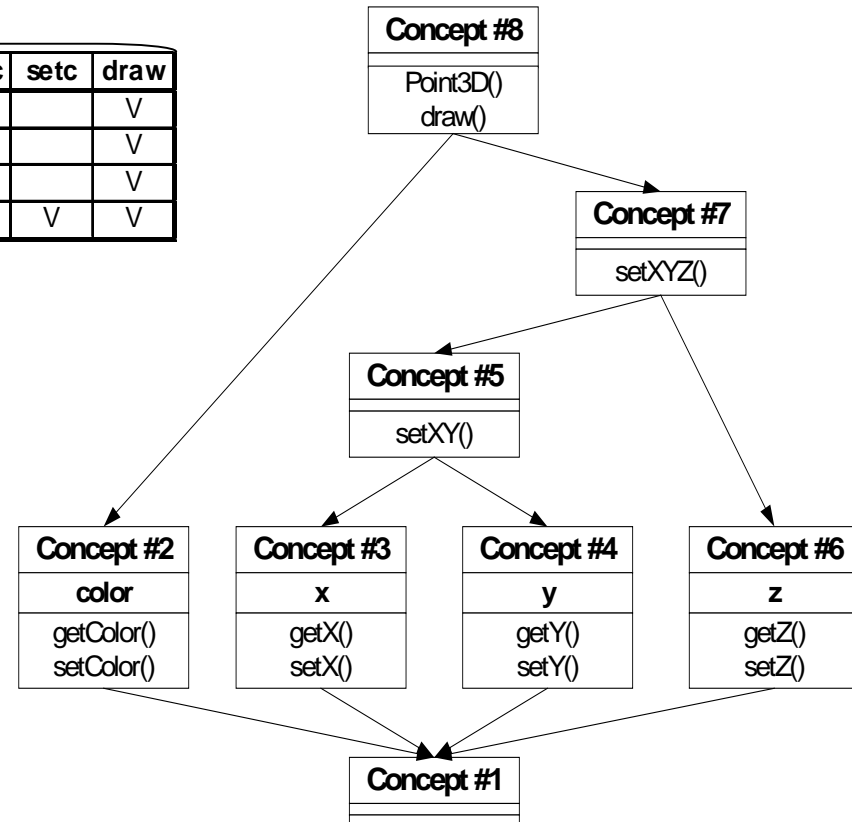
- Captures the relation of *accesses* from methods to fields.
 - Field usage is critical for understand a class.
 - All implementations of an operation use the same fields.
 - Representation changes are rare.
 - Methods that use the same combination are related.
 - Can be calculated directly from the *.class* file.
 - Allows some reverse engineering without source code.
 - Calculated using standard static analysis techniques.
 - Currently restricted to accesses inside the class.

Class Outline Example

Context for class Point3D

		features											
		Point3D	getX	setX	getY	setY	setXY	getZ	setZ	setXYZ	getc	setc	draw
instances	x	V	V	V			V			V			V
	y	V			V	V	V			V			V
	z	V						V	V	V			V
	c	V									V	V	V

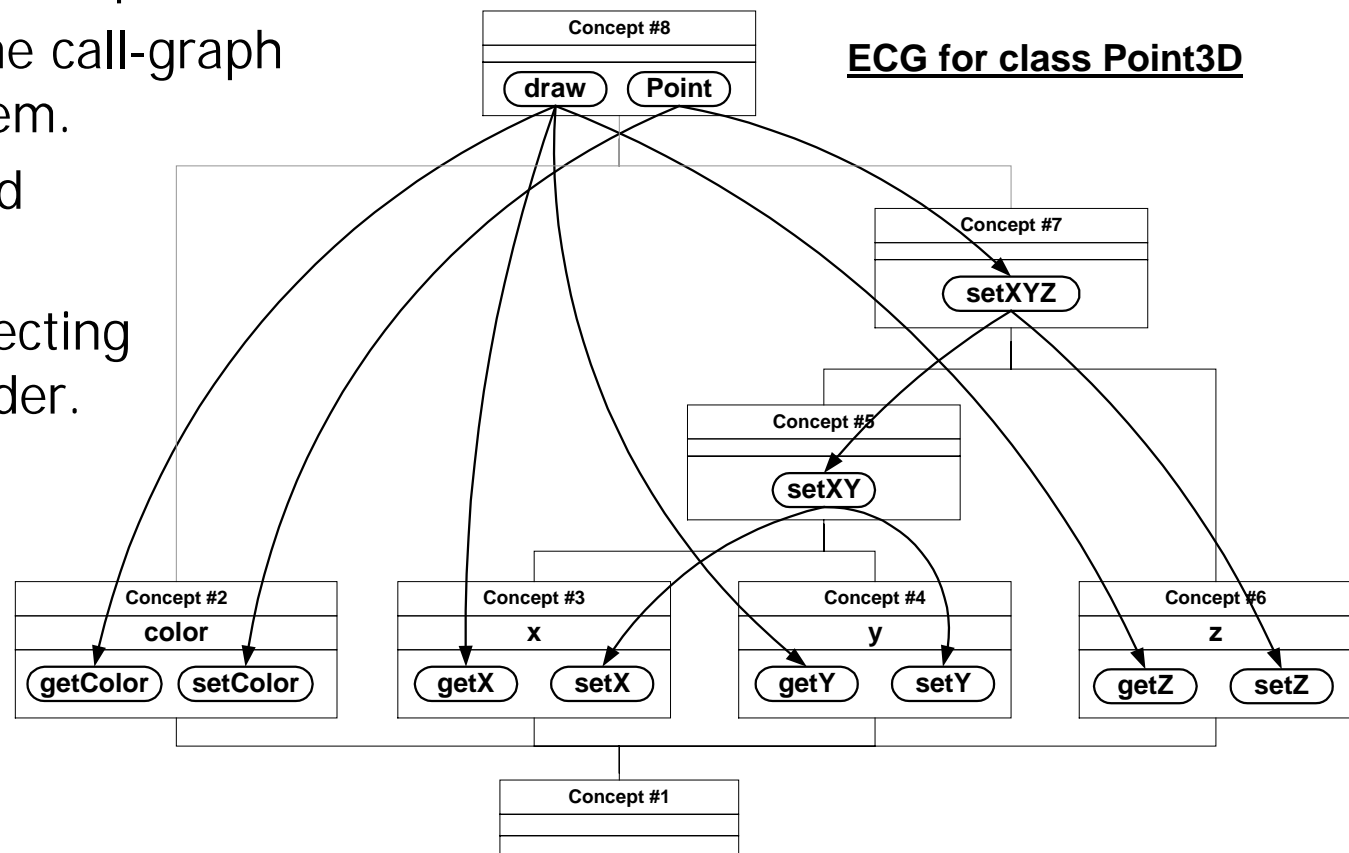
Concept Lattice for class Point3D



- All the methods of a concept use the same combination of fields.
- We derive the following information:
 - Low cohesion between color and coordinates.
 - Lack of symmetry between coordinates.

Embedded Call Graph

- An *Embedded Call Graph (ECG)* is a superposition of a call-graph on a concept lattice.
- Addresses the call-graph layout problem.
- More detailed than lattice.
- Useful in selecting a reading order.





Reading Source Code

- The reading order problem.
 - Efficient and effective code inspection.
 - e.g. eliminating duplicates.
- Original source code order not optimal.
 - Co-definitions.
 - No incremental order.
 - All class members are defined simultaneously.
 - Perturbations to original order.
 - Maintenance.
 - Language issues (e.g. inheritance).
 - Style issues (e.g. public before private).



Suggested Reading Order

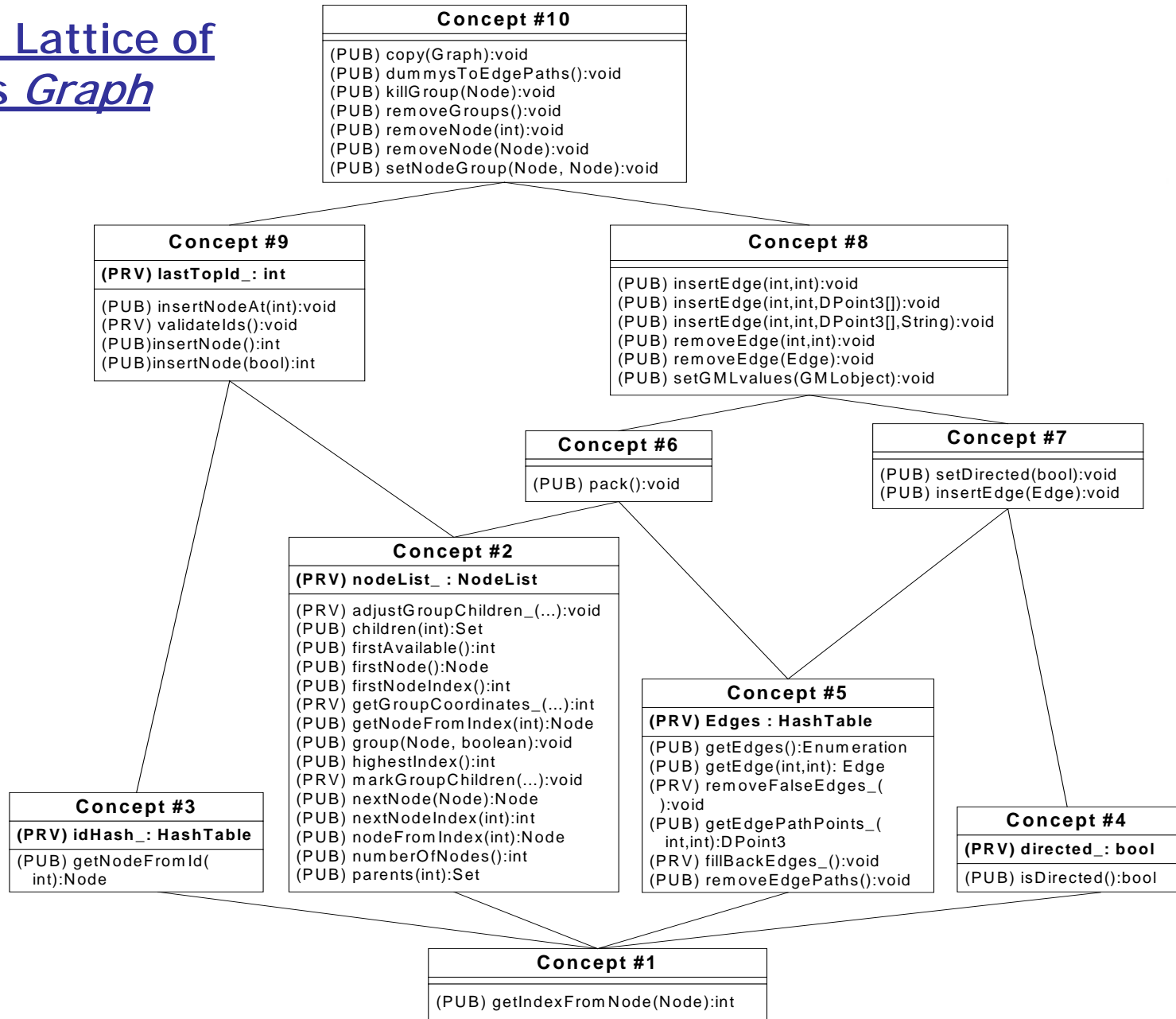
- General strategy:
 - Global order – between groups of methods.
 - Local order – between methods in each group.
- Guidelines (by importance):
 - Methods of each concept read consecutively.
 - Methods read in (reverse) topological order.
 - Concepts of each layer read consecutively.
 - Methods read from simplest to most complex.
 - Each “horizontal component” inspected separately.



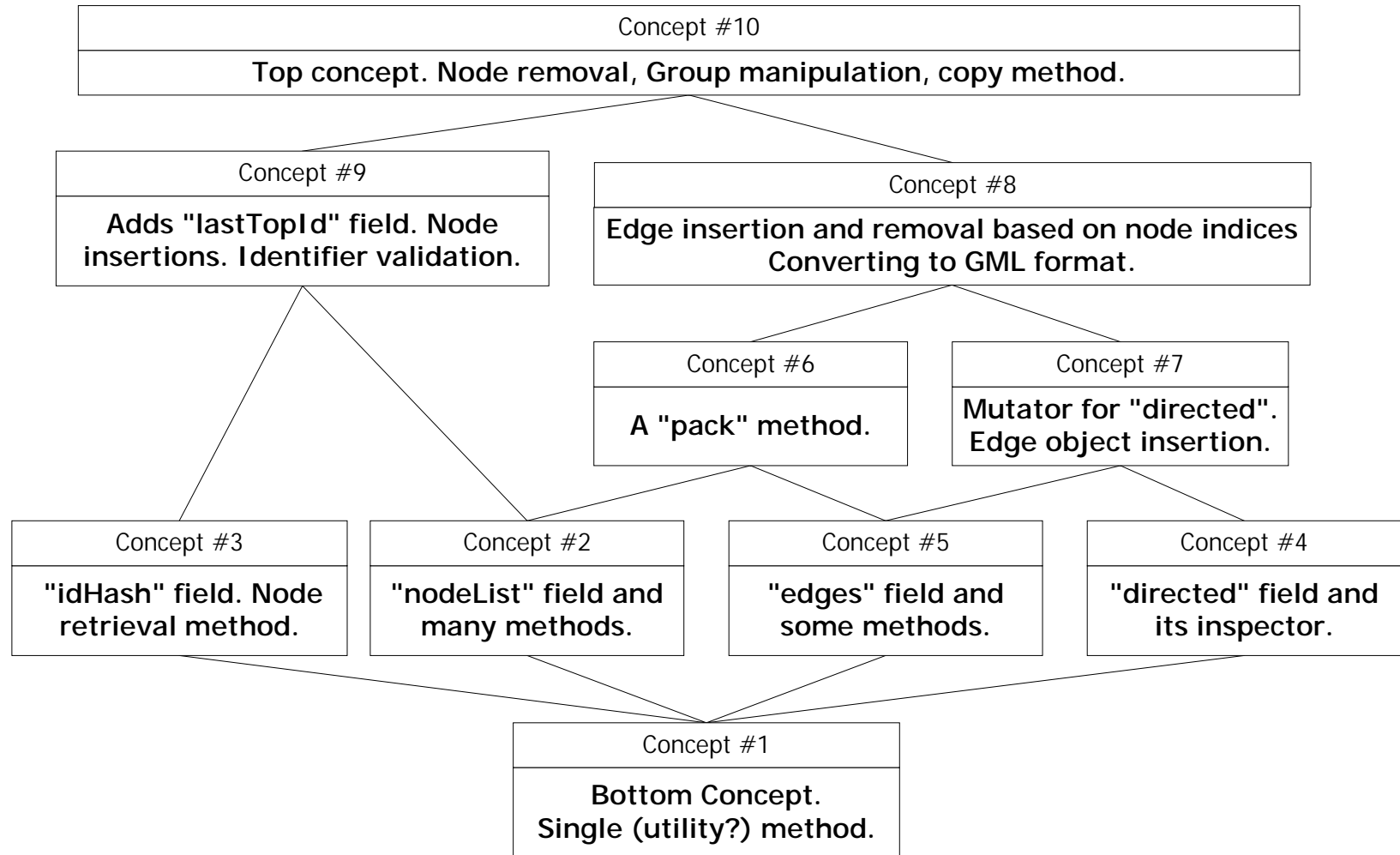
Case Study: VGJ

- VGJ = “Visualizing Graphs with Java.”
 - Java applet for graph drawing and layout.
 - Developed in Auburn University, adapted and extended in many institutions under GNU license.
 - Consists of two parts: GUI and Algorithms.
- We investigate the *Graph* class.
 - Primary data structure used by both parts.
 - 37 *public* and 6 *private* methods, 5 *private* fields.

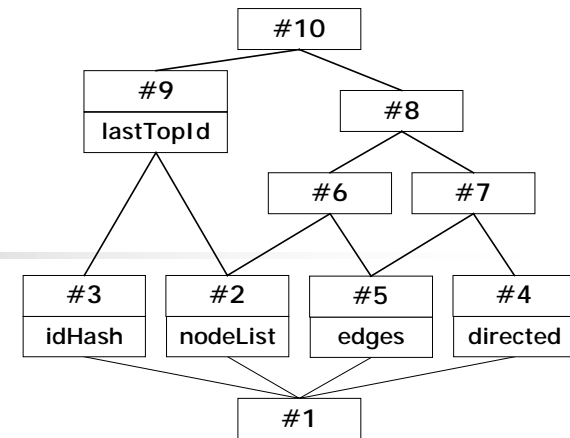
Concept Lattice of class *Graph*



Simplified Representation of the Lattice of *Graph*

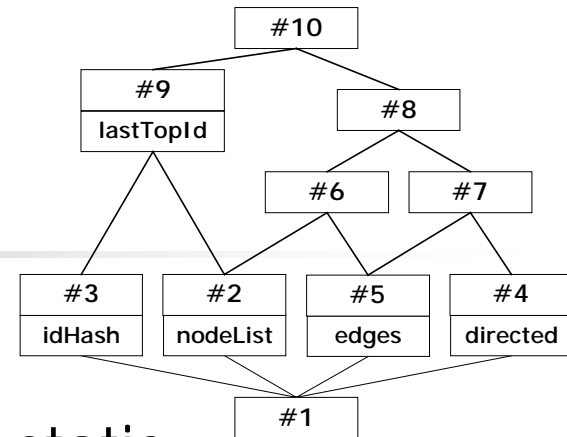


Case Study: VGJ



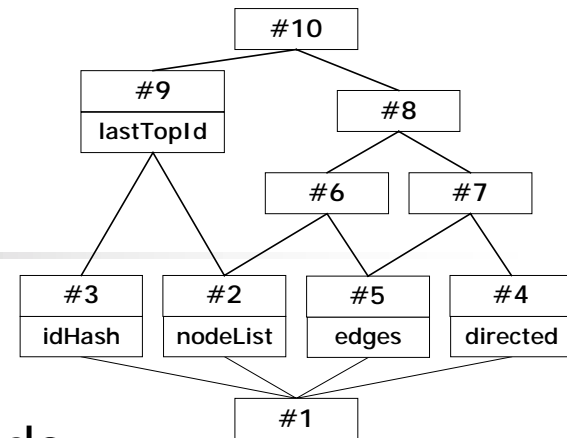
- Inspection of Lattice structure:
 - Three parts:
 - Left – Node identifiers and hash.
 - Right - Edges and directionality.
 - Middle – Node list (used by both).
 - Conclusions:
 - Possible redundancy in nodes.
 - Nodes and edges are maintained by the graph.
 - Edge manipulation affects nodes.
- Ordered concepts inspection:
 - Each concept focused on a specific responsibility.
 - Allows implementation prediction and verification.

Case Study: VGJ



- Bottom concept
 - Methods not using fields, usually static.
 - We find a simple macro that is mistakenly not static.
- Bottom layer
 - Methods that use a single field.
 - (Example) Findings in concept #2:
 - 9 of 12 methods are simple delegators providing iteration capabilities – Replaceable with an iterator.
 - Two duplicate methods, separated by ~500 LOC.
 - Comparing methods *children* and *parents* reveals significant performance differences.
 - *Children* and *parents* bypass the delegators.

Case Study: VGJ



■ Middle layers

- Methods that combine few of fields.
- (Example) Findings in concept #7:
 - Class maintains a directed graph.
 - Undirected graphs are simulated with additional edges.
 - Simulation is not complete. Potential errors for clients.

■ Top layer

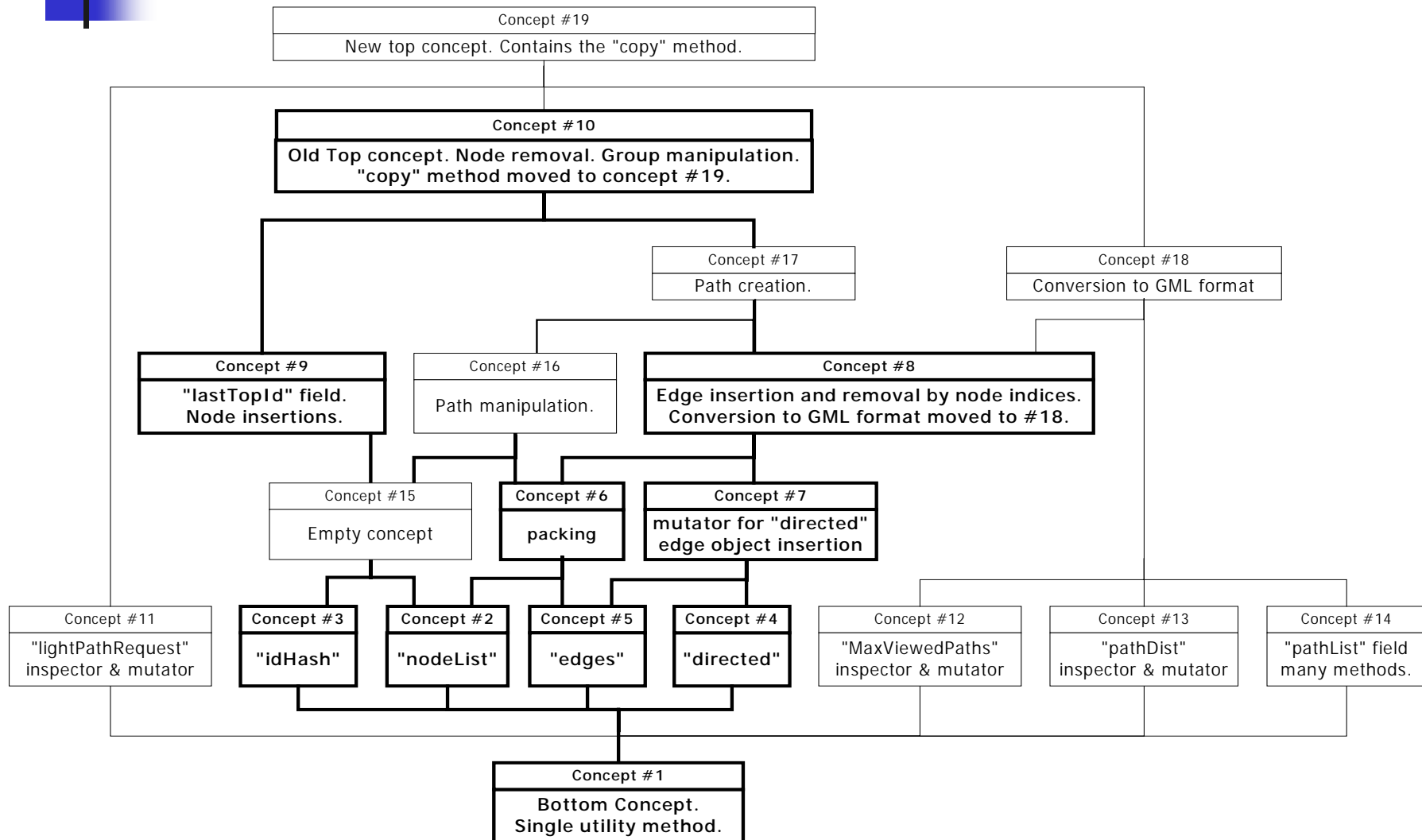
- Methods that use maximal combinations of fields.
 - Can be used to abstract the lattice.
- (Example) Findings in concept #8:
 - Parallel edges not allowed in Graph.
 - Two fields are redundant: not stored to file.



Version Comparisons

- Lattices provide an outline of changes.
 - Areas of additions and modifications.
 - Implementation changes that modify field usage.
- Demonstrated on a modified version of VGJ.
 - Modified by at least one group of Technion students.
 - Versions history is lost.
 - 9 fields and 69 methods.
 - vs. 5 fields and 43 methods in the original.

Case Study: Modified VGJ





Future Research

- Integration with development environments.
- Applicability to class design in CASE tools.
 - An interactive lattice-based single class editor.
 - Allows semantic assignment to methods.
 - Facilitates the creation of wide interfaces.
- Defining metrics for lattices of classes.
 - Foundation for a new metric suite for classes.
 - Prediction of lattice usefulness for specific classes.
- Simplifying and abstracting complex lattices.
- Conducting user studies.
- Extension to interaction with other classes.



Questions / Comments ?
