Computer Science

5. Persistence (Part II)

Engineering Web and Data-intensive Systems

Dr. Volker Riediger - Winter Term 2022/23

Institute for Software Technology

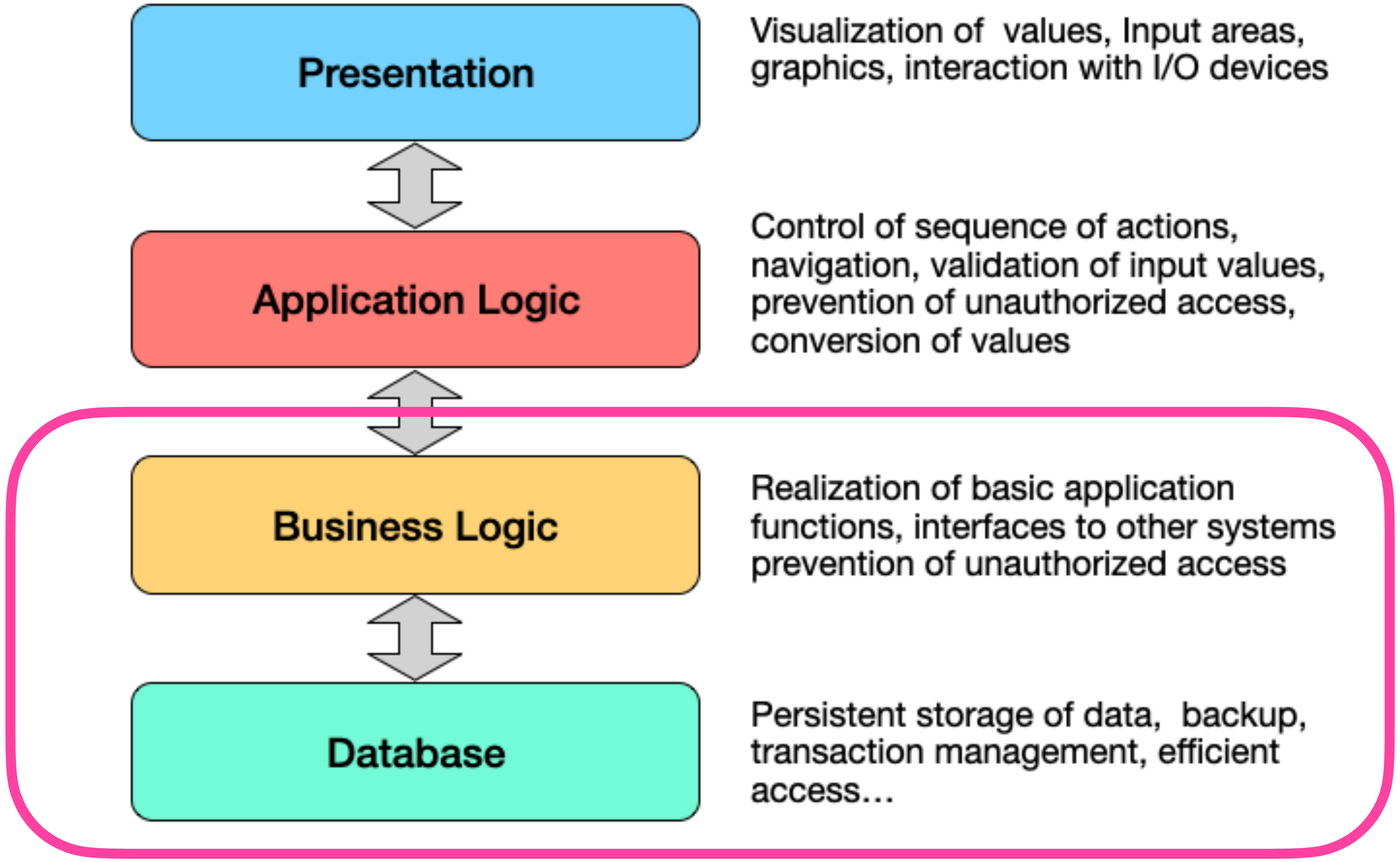
Persistence (Part II)

- Graph Databases
- Graph Types
  Labeled Property Graphs
- Graph Traversal
- Graph Queries
  Introduction to Neo4J
- Graph Schemas,
  Object - Graph Mapping

Image: colourbox.de

Persistence: Overview

- Data Properties (I)
- Persistence Tasks (I)
- Persistence vs. Scaling (III)
  - Data Intensive Systems
  - Distributed Storage
  - CAP, ACID, BASE
- Data Mappings
  - Relational (I)
  - Graph (II)
  - Document (III)



5.4 Data Mapping II - Graphs

5.4.1 Graph Databases

Observations

- In many cases, the information structure of software systems is more graph-like, rather than tabular (and the other way round…)
- When you're bound to store graph data in tabular form (or to store table data in graph form), this feels unnatural.
- It often introduces additional cognitive load to bridge the gap between logical structure and physical storage.
- It also can introduce additional computational load to do so.

Observations

- If you know your logical data model, and you also know the requirements (e.g. which type of queries you might expect, what are performance goals etc.), you can choose the persistence technology that matches best.
- Often, you'll have to store your information in various representations and technologies at the same time to fulfill conflicting requirements. This also means that you have to ensure that those representations are kept in-sync and linked to each other.

Graph Databases

Variant of NoSQL databases

- NoSQL means non-relational
- Other flavors of NoSQL databases are (among others)
    - Object-oriented databases
    - Key-value stores
    - Document stores (⇢ part III)

- Straightforward representation of objects and their relations
- Based on graphs
- Mathematical foundation in graph theory
- Various graph types
- Efficient graph algorithms

Graph Databases

- Various persistence mechanisms
  - in-memory
  - external storage
  - distributed storage
- Various consistency measures, e.g.
  - ACID transactions
  - Schema checks
  - User defined constraints

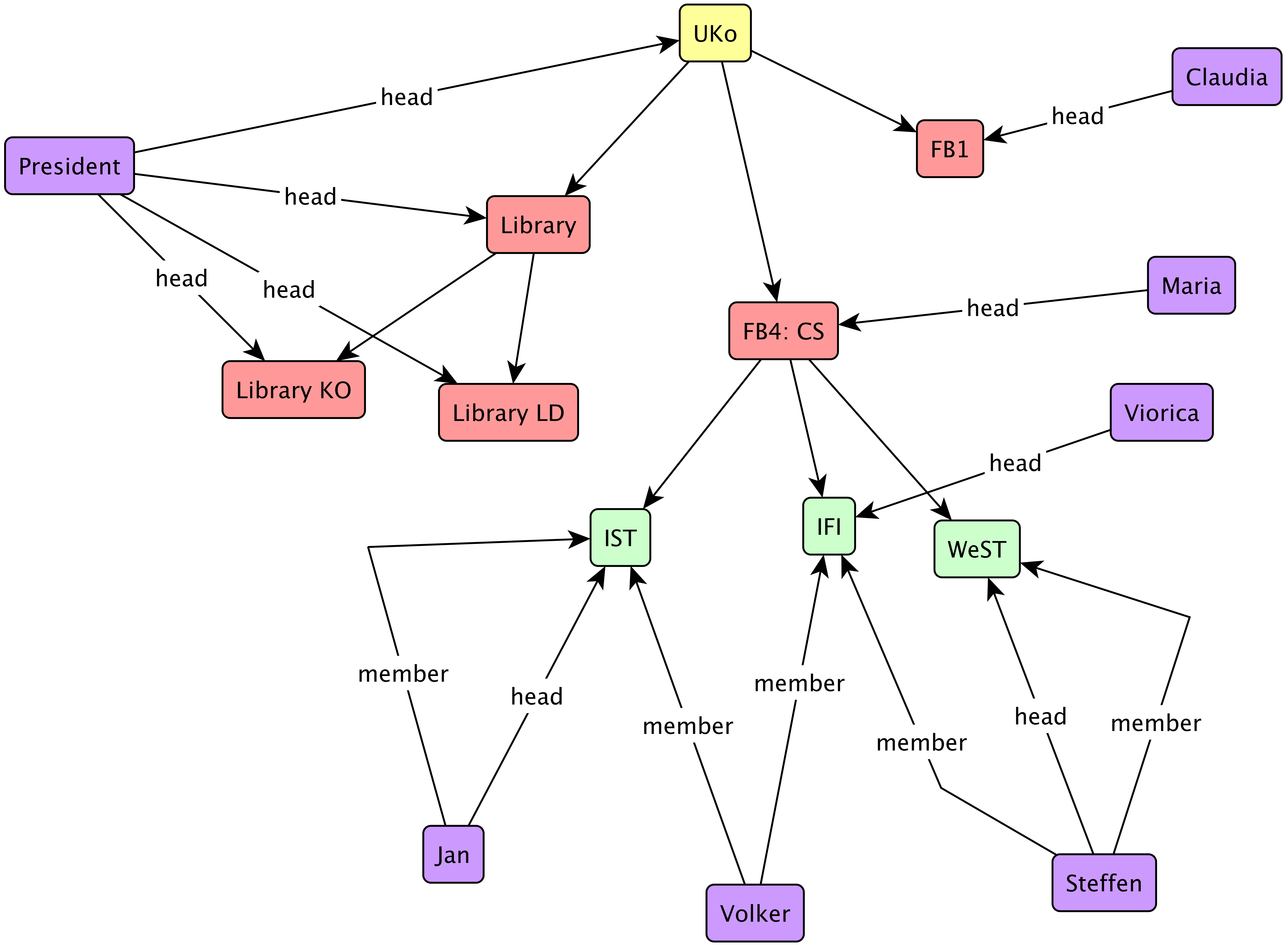
- Various APIs, e.g.
  - Access layer in programming language
  - Object-Graph-Mappers
  - Network access via
    - REST API
    - and/or proprietary protocols

What is a Graph?

- Mathematically, a Graph is a tuple $g = (V, E)$
  - of a set V of vertices (nodes)
  - and a set E of edges (arcs, relations)
- The edge set E is a subset of the cartesian product $V \times V$
  - E is a set of pairs (v, w)
  - of connected vertices $v, w \in V$

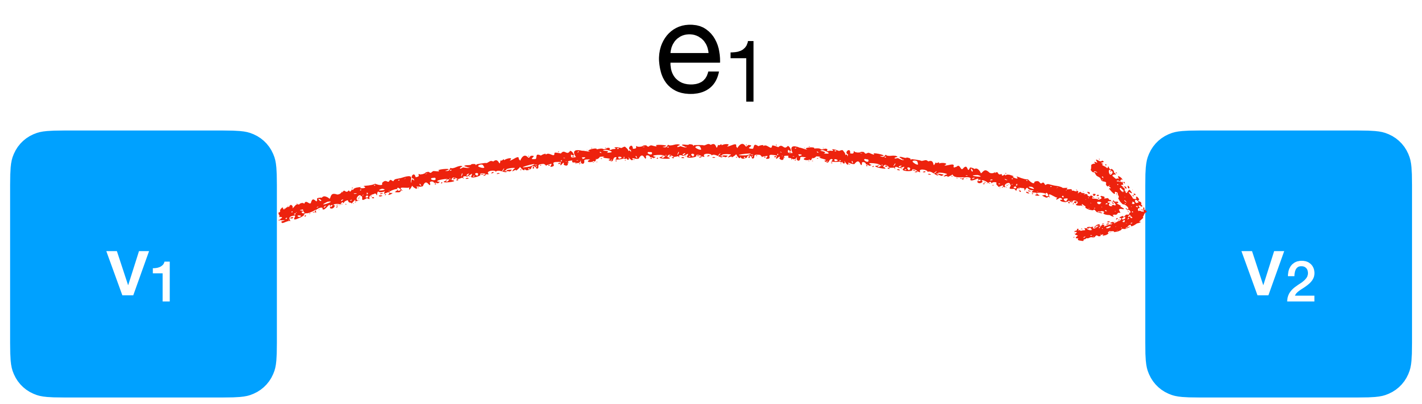
- In contrast to simple references in object networks, edges are “first-class citizens”
  - an edge is an object on it’s own (references are anonymous pointers)
  - an edge has an identity (references have no identity)
  - edges may have attributes (references only point to an object)

Simple Graph Example: University Structure



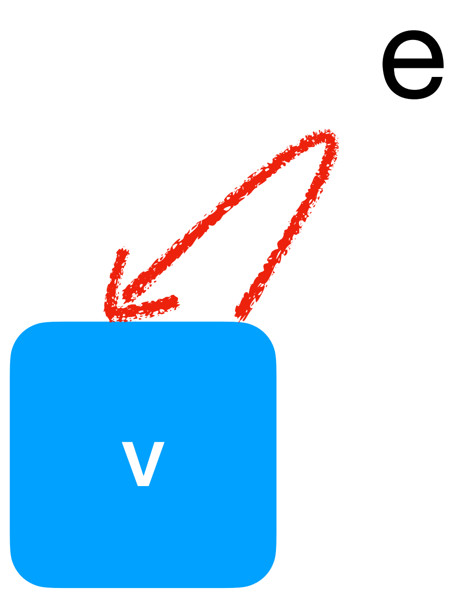
Some simple definitions

- If E contains an edge $e_1 = (v_1, v_2)$
  - we say that v_1 is adjacent to v_2 (and vice versa)
  - and that e_1 is incident to v_1 and v_2



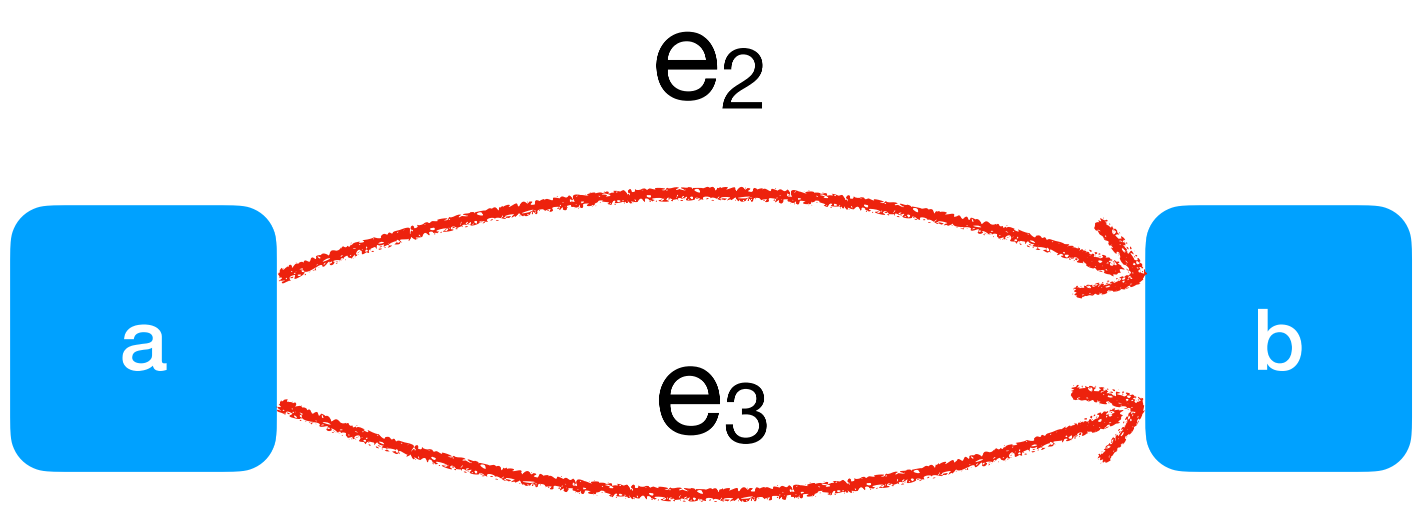
- All incident edges e_i of a vertex v form the incidence set $\Lambda(v)$

- An edge $e = (v, v)$ is called a loop



- Different edges $e_2 = (a, b)$ and $e_3 = (a, b)$ connecting the same nodes are called parallel

  (some graph types allow/disallow parallel edges)

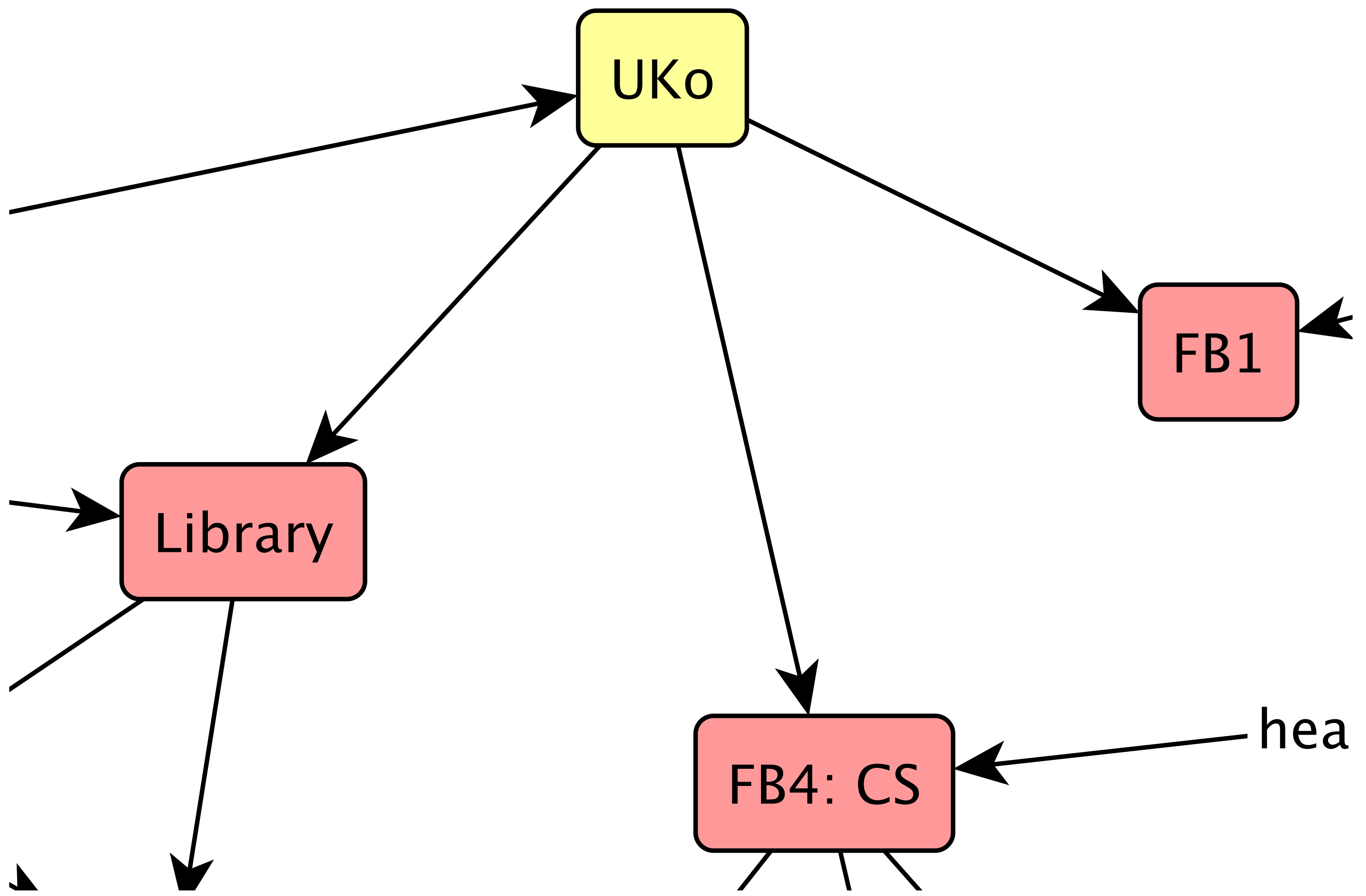


Example Instance in Math….

$$universitygraph = (V_1, E_1)$$

$$V_1 = \{\text{UKo}, \text{FB1}, \text{FB4}, \text{Library}, \ldots\}$$

$$E_1 = \{(\text{UKo}, \text{FB1}), (\text{UKo}, \text{FB4}), (\text{UKo}, \text{Library}), \ldots\}$$



- This representation still misses many important features, e.g.
  - type (color) of vertices and edges
  - attributes
  - direction of edges
- order of vertices and edges
- order of incidences
- constraints, e.g. number of incidences, which edge type may connect which vertex type, …

Various internal representations (examples)

$$examplegraph = (V_1, E_1)$$

$$V_1 = \{\mathrm{UKo}, \mathrm{FB1}, \mathrm{FB4}, \mathrm{Library}, \ldots\}$$

$$E_1 = \{(\mathrm{UKo}, \mathrm{FB1}), (\mathrm{UKo}, \mathrm{FB4}), (\mathrm{UKo}, \mathrm{Library}), \ldots\}$$

Representation determines computational complexity and efficiency w.r.t. time/space

Adjacency Matrix

	UKo	FB1	FB4	Library	IST	IFI	...
UKo		e1	e2	e3			
FB1							
FB4					e4	e5	
Library							
IST							
IFI							
...							

Incidence Lists

	incoming	outgoing
UKo		e1, e2, e3
FB1	e1	
FB4	e2	e4, e5
Library	e3	
IST	e4	
IFI	e5	
...		

Graph Algorithms

- Well-known graph algorithms such as
  - BFS, DFS,
  - Dijkstra’s algorithm (shortest path), A*,
  - Min-Cut,
  - Max-Flow,
  - and many more

  form a base toolkit for efficient processing of graphs

5.4.2 Graph Types

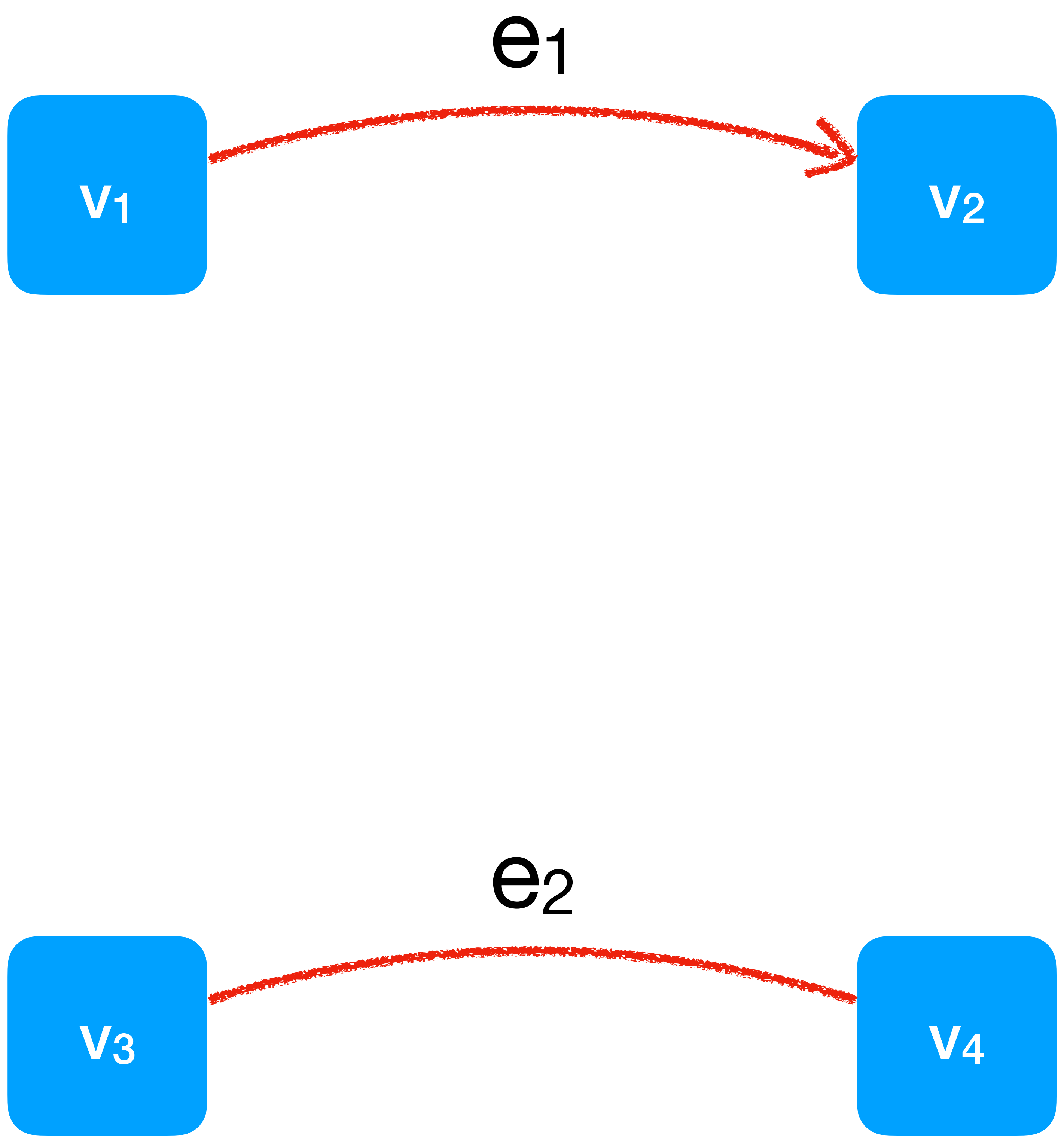
→ Labeled Property Graphs

Graph Type Features

- Additional features extend the basic graph model, i.e. graphs come in different flavors
  - directed vs. undirected edges
  - typed vertices and/or edges
  - type system capabilities, e.g. support for generalization
  - labeled vertices and/or edges
  - ordered vertex/edge/incidence sets
  - binary edges vs. hyper edges (with more than two ends)
  - hierarchical graphs (e.g. nodes can contain graphs)
  - …
- Following the basic definition, the features can also be specified mathematically
- Such a specification is a formal base for theorems, proofs, algorithms, queries, …

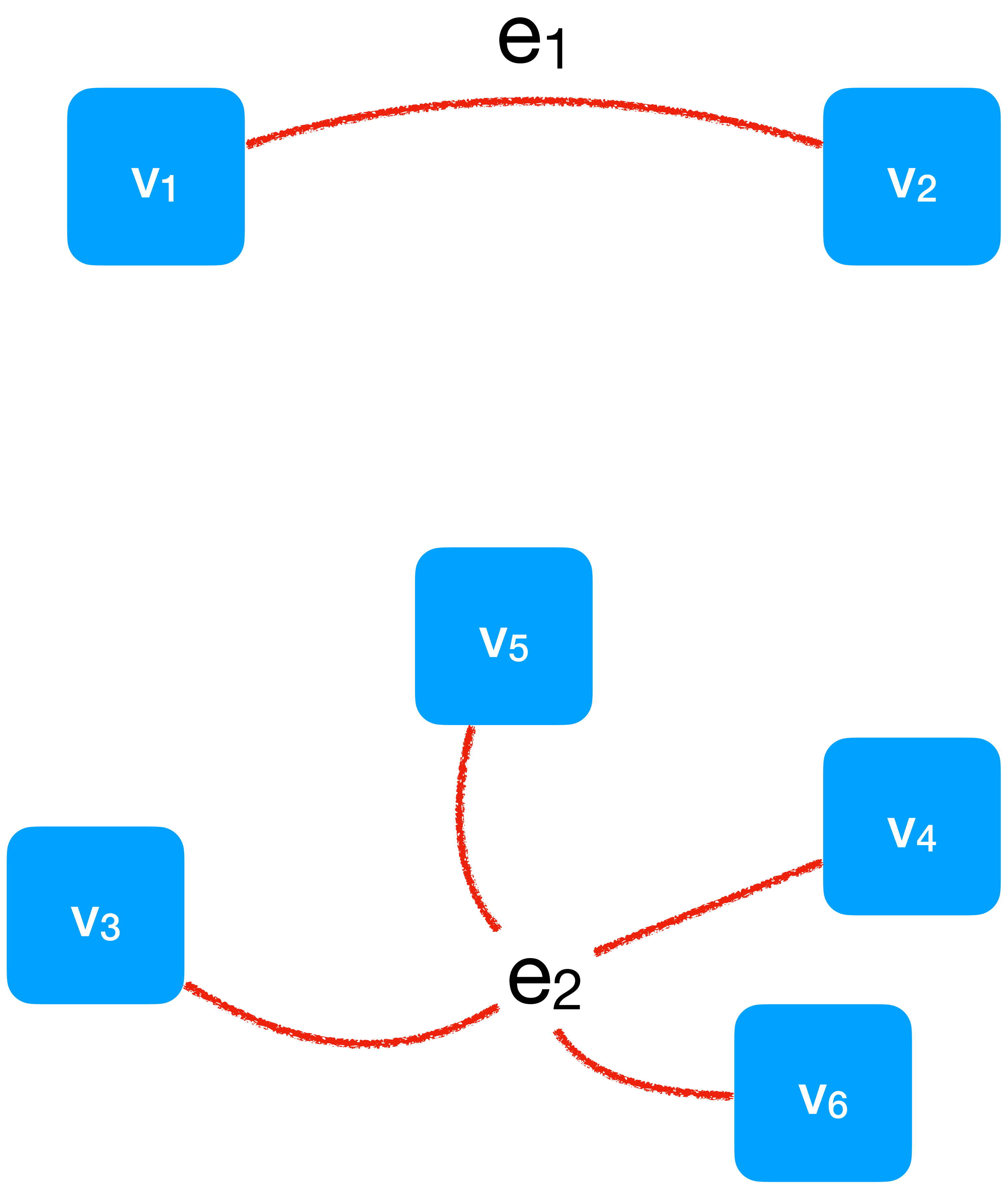
Directed vs. undirected

- **Directed**
  - Edge has start vertex α and end vertex ω
  - “e1 goes from v1 to v2”
  - Traversal usually in edge direction, but also possible in opposite direction
- **Undirected**
  - Edge simply connects vertices
  - “e2 connects v3 and v4”



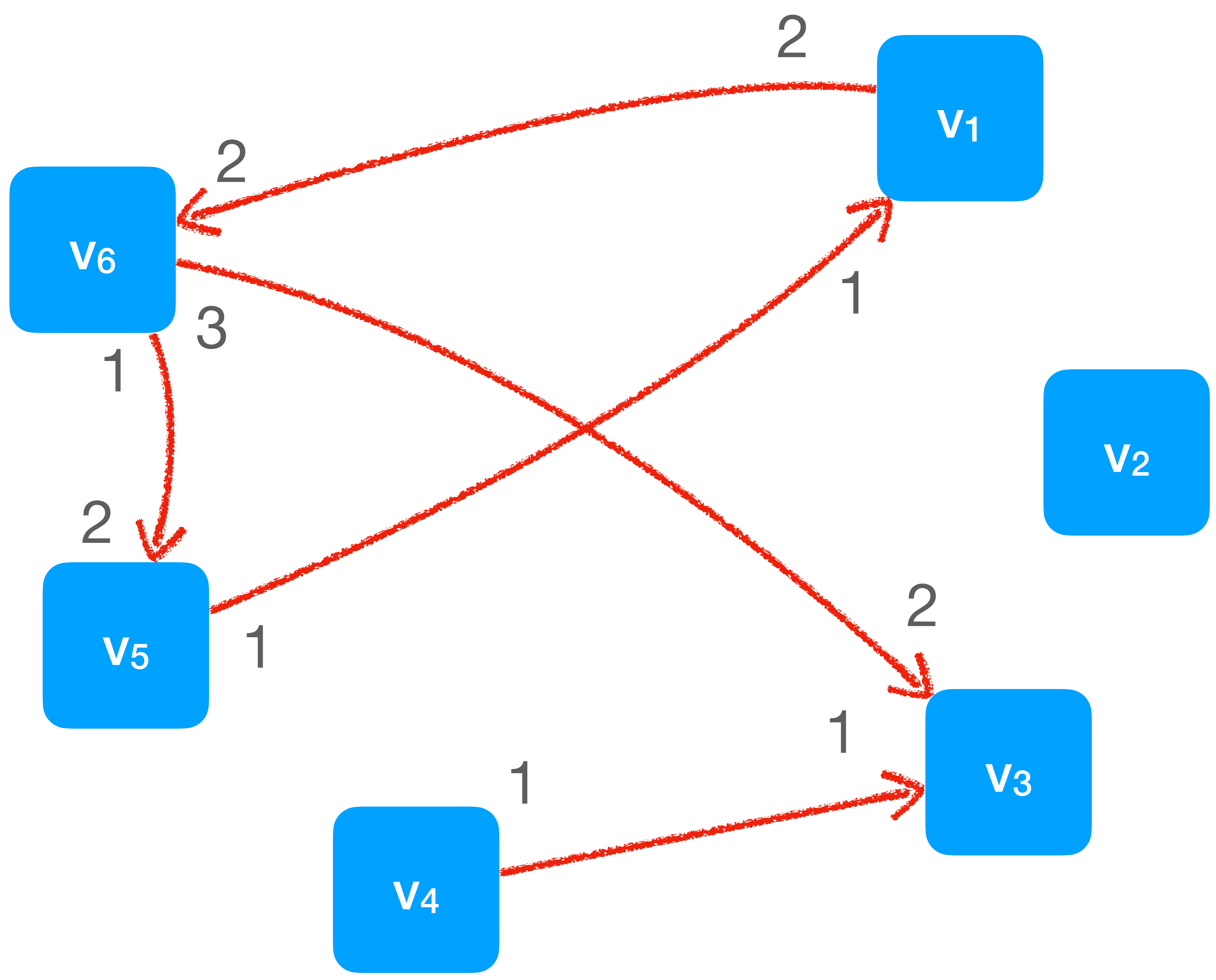
Binary vs. Hyper-Edges

- **Binary**
  - Edges have exactly two ends
  - Variants with/without loops permitted
  - Variants with/without parallel edges
- **Hyper Edges**
  - Edges can connect arbitrary many nodes



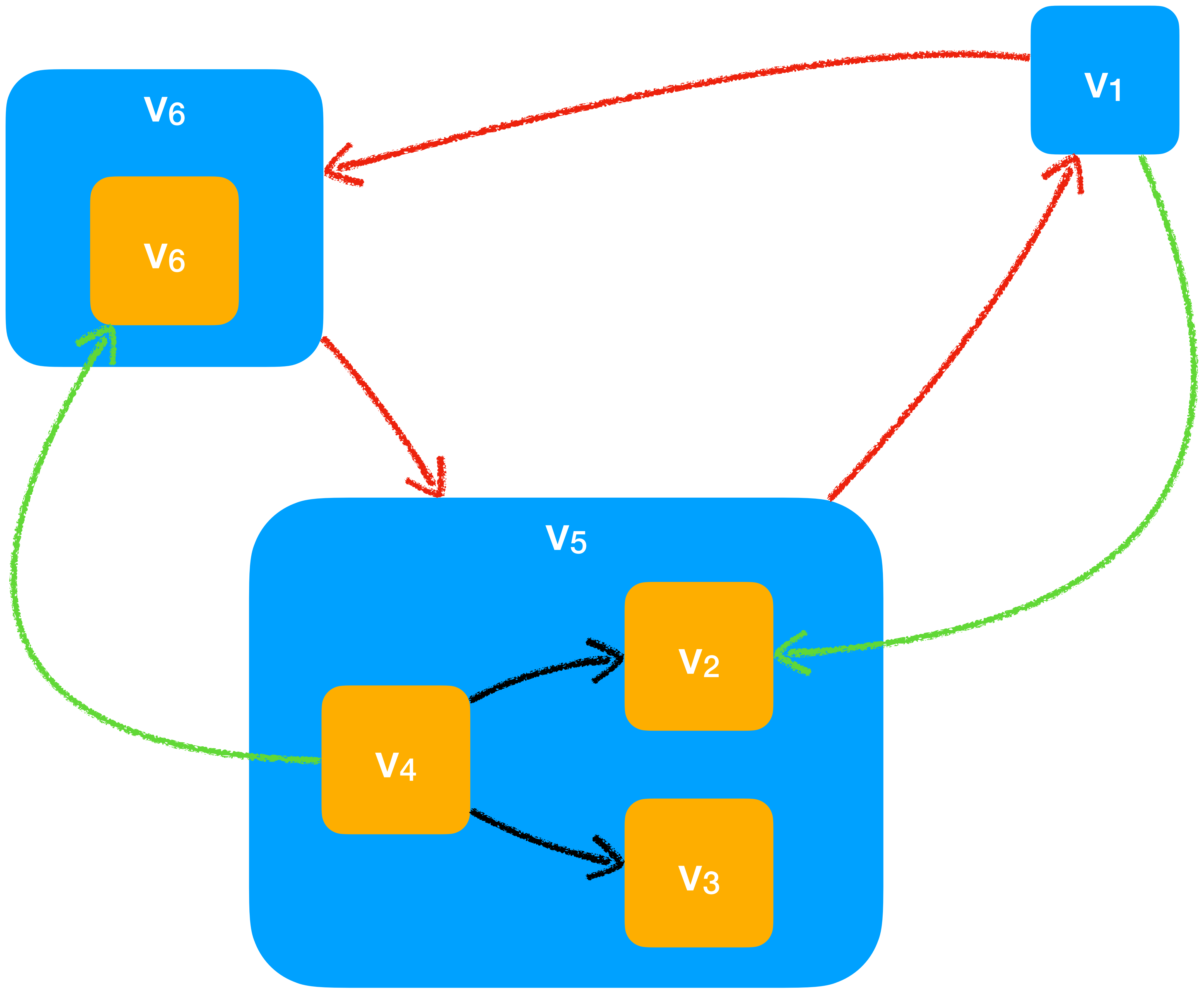
Ordered vs. unordered

- **Ordered graphs**
  - Deterministic order of
    - nodes and/or
    - edges and/or
    - incidences
- **Unordered graphs**
  - Order is undefined
  - Usually, creation order is preserved



Hierarchical Graphs

- Nodes and/or edges can contain subgraphs
- Variants with/without edges across hierarchy levels
- Variants with limited/unlimited nesting levels



Example: TGraphs

(⇢ https://github.com/jgralab)

- Typed nodes and edges
  node/edge has exactly one type
- Typed, directed, binary edges
  can be traversed in both directions
- Properties (attributes) for both nodes and edges stored
- Extensible attribute domains
- Traversal of incident edges to adjacent nodes is constant-time operation
- Strict schema and typing
- Deterministic ordering of nodes, edges, and incidences
- Type system supports generalization with multiple inheritance
- Consistency constraints defined by schema enforced
- Complex constraints defined by graph queries
- No hyper edges
- No hierarchy

Example: Labeled Property Graphs

Graph model of Neo4J (⇢ https://neo4j.com, section 5.4.4)

- Labeled nodes
  a node can have multiple labels
- Typed, directed, binary edges
  can be traversed in both directions
- Properties (attributes) for both nodes and edges stored as key-value pairs
- Traversal of incident edges to adjacent nodes is constant-time operation

- No ordering
- No generalization in type system
- No hyper edges
- No hierarchy
- No consistency constraints, e.g. on uniform properties per label and/or type

5.4.3 Graph Traversal

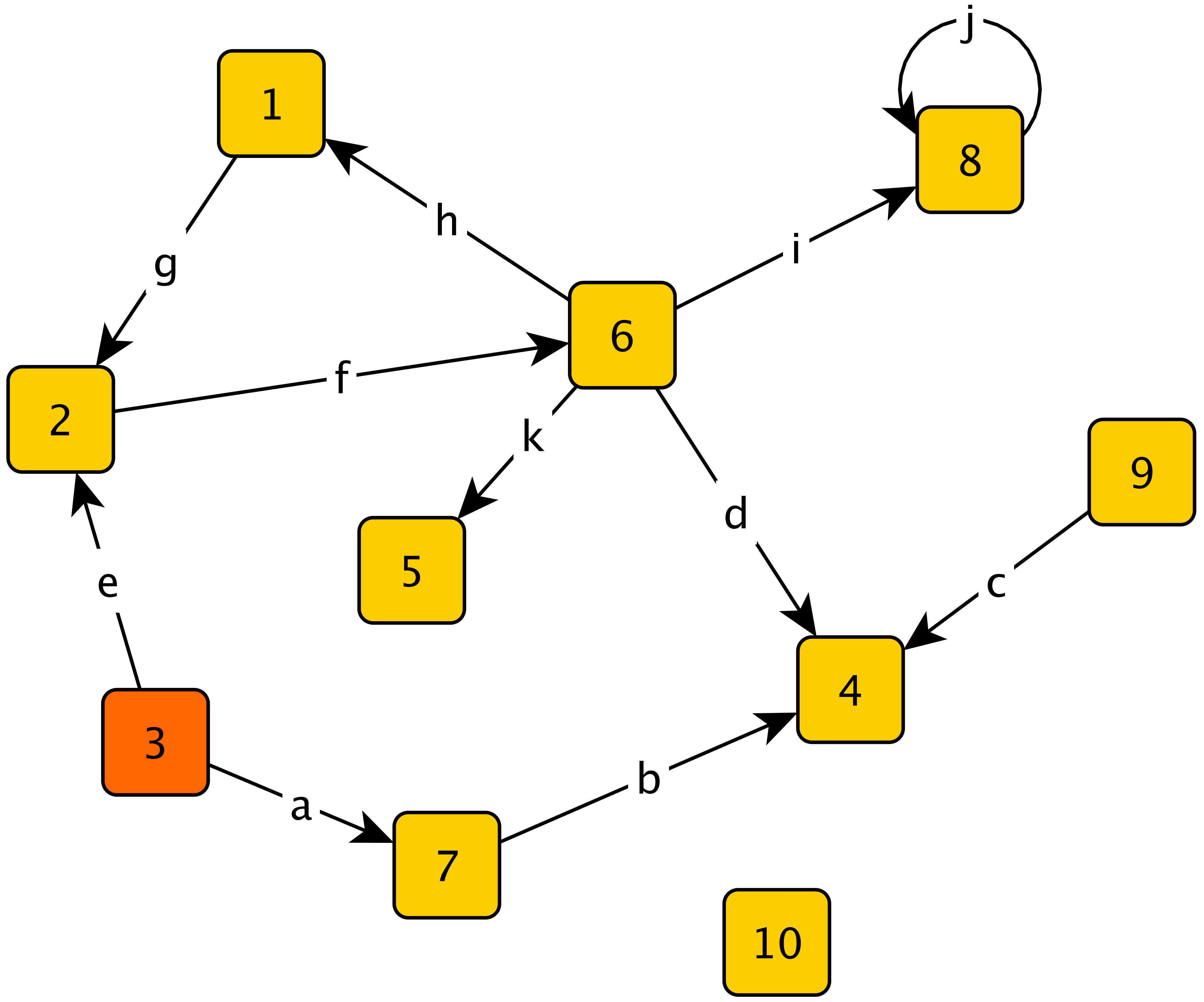
Navigation in Graphs

- Basic navigation is based on a traversal of the graph
- Traversal means visiting (all) elements of a data structure
- In general, traversal starts at a given vertex v, follows it's incident edges $e \in \Lambda(v)$ and visits the neighboring vertices
- For the neighboring vertices, the traversal proceeds until all reachable vertices have been processed
- Since edges are first class citizens, they can be "visited" as well

DFS

Depth First Search

- Visits descendants before siblings
- Determines reachable subgraph ("spanning tree")
- Search algorithms must consider edge direction, multiple paths, loops and cycles, unreachable nodes



Example: DFS visiting order starting at node 3 (assuming no specific edge order)

3 a 7 b 4 e 2 f 6 i 8 (j) h 1 (g) k 5

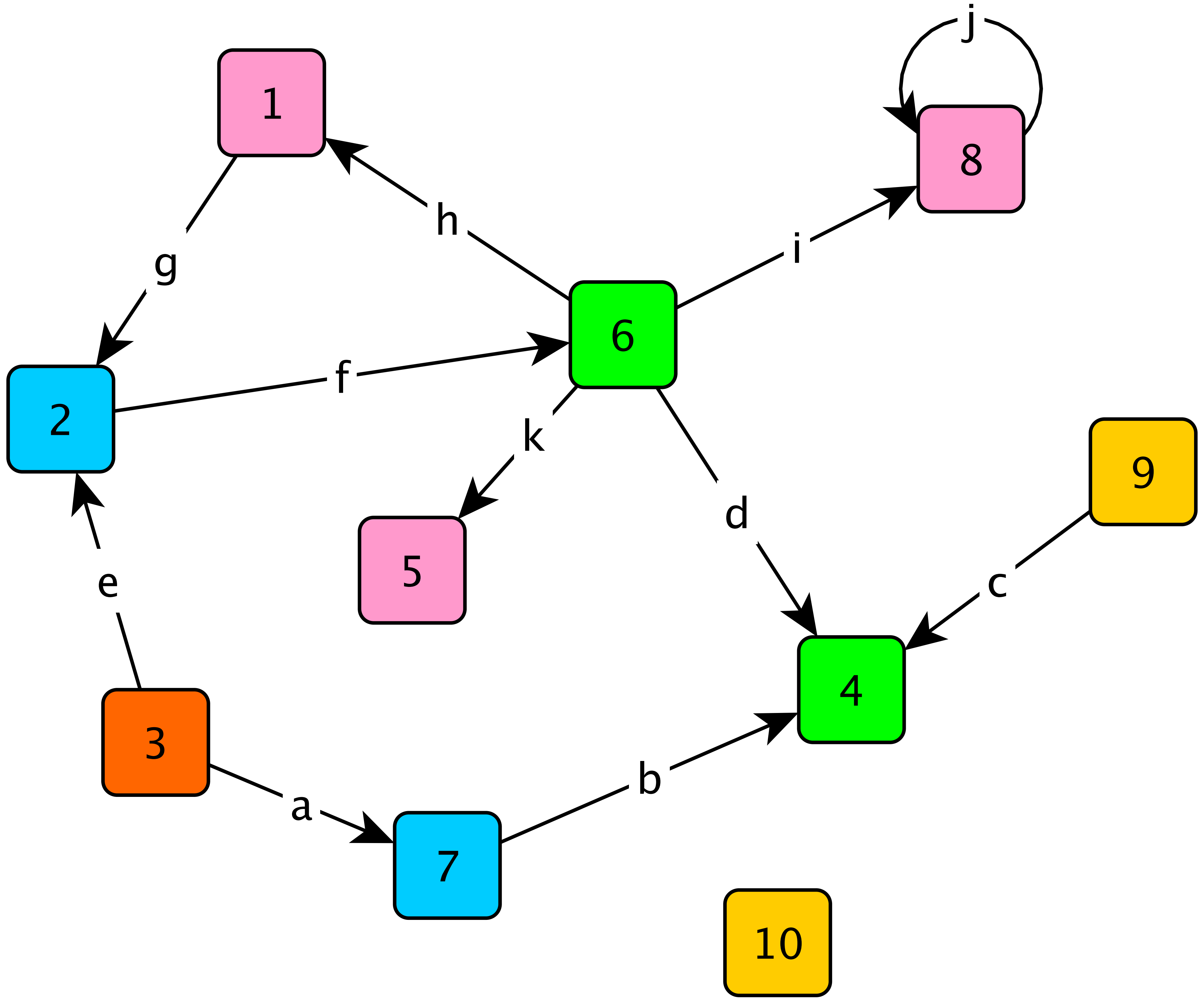
Unreachable: 9, 10

BFS

Breadth First Search

- Starts at a specific node
- Visits siblings before descendants
- Determines reachable subgraph and shortest paths

- BFS variant to determine shortest path w.r.t. an edge weight
  - Dijkstra's Algorithm
  - A* algorithm



Example: BFS visiting order starting at node 3 (assuming no specific edge order)

3 a 7 e 2 b 4 f 6 i 8 h 1 k 5 (j) (g)

Unreachable: 9, 10

Distances from start vertex: 0, 1, 2, 3

Parameterized Search

- If the graph model supports types, a traversal can be, e.g.,
  - using outgoing edges, incoming, or both
  - only visit nodes/edges with a specific type
  - only visit nodes/edges based on conditions on property values
  - limit path length
- specify patterns on edge type sequences of paths, e.g.,
  - mandatory, optional, or repeated edges
  - specifying with minimum and/or maximum occurrences
  - regular path expressions
- …and more possibilities
- Parameterized search forms base for graph queries

5.4.4 Graph Queries

→ Introduction to Neo4J

A Graph Database Engine

- In the tutorials and assignments, we'll use Neo4J as graph database engine (see https://neo4j.com)
- Neo4J is a production-strength graph database
  - huge community
  - ongoing active development
  - enterprise features like capability to accommodate billions of vertices and edges, transactions, replication, load balancing, …
- Implements labeled property graphs
- No fixed schema required

- Neo4J implements GQL, the Graph Query Language (formerly called Cypher)
  - GQL was adopted by ISO in 2019, timeline expects international standard in 2021
  - Manipulation and querying of graphs
  - Queries based on matching of vertices, relations, and paths

Review: Navigation and Querying in Relational DBs

- Based on relational calculus
- Relational data model describes relations, attributes, keys, and dependencies
- Relations with their attributes contain data as sets of tuples (table rows)
- Basic relational operators
  - projection π,
  - selection σ,
  - join $\bowtie$,
  - and rename ρ
- Applied to relations and combined to form complex expressions

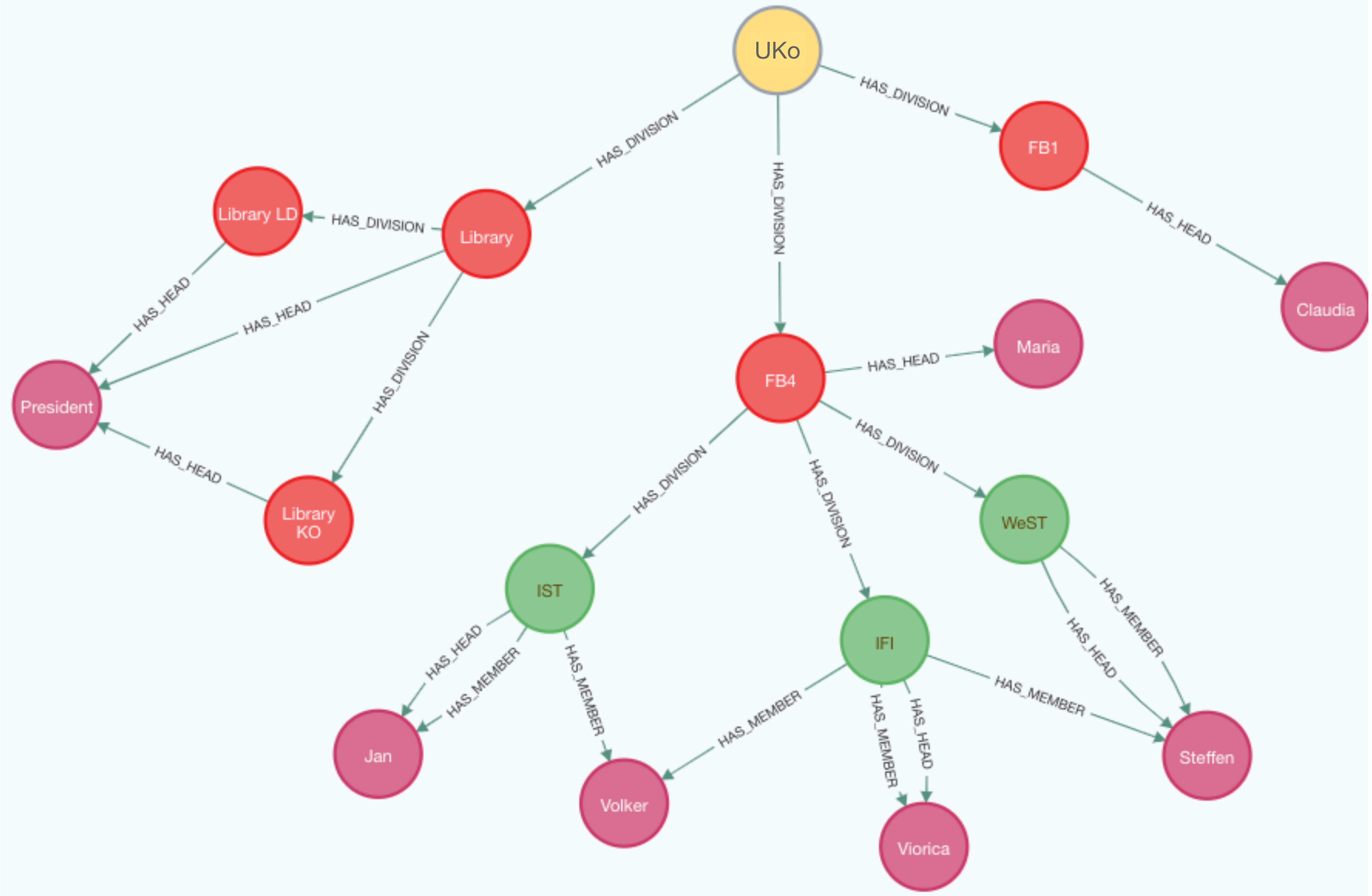
- In relational database systems (implementation of the relational model), a query language, e.g. SQL, facilitates data access and calculations.
  - Relations implemented as tables
  - SQL implements the relational operators (and much more…)
- Properties of objects often scattered over many tables
- Navigation of related data via join operations and result sets
- Hard to tell “technical” joins from “semantic” joins

Graph Queries

- In graph representations, usually objects are nodes and their relations are edges
- All properties of an object in a single location (not scattered)
- Related objects can be determined efficiently (constant time)
- Straightforward mapping of conceptual domain models to persistent entities (nodes, edges)

- Graph query engines usually compute (typed) paths that connect vertices over more than one edge, using parameterized BFS
- Problems to solve:
  1. Find the start node(s)
  2. Specify search pattern
  3. Describe desired result

Example Query: Determine all Employees of UKo



Step 1: Find the start node

- Graph search algorithms need a start node
- Finding the start node in a huge graph can be inefficient
- Usually, nodes and edges have an internal numeric ID
  - When this ID is known, the DB can locate the node fficiently
  - Sadly, the ID is usually unknown…
- Linear search (look at all nodes) usually too inefficient
- Indexes (comparable to those in relational databases) improve lookup
- In some queries, multiple start nodes, each triggering an individual search, can be useful
- Instead of searching the nodes, relations can also be used as a starting point
- In our example, we need to find a node of type "University" with name "UKo"

GQL (Cypher) Queries

- GQL - a declarative graph query language
- General query pattern:
  - **`MATCH`** <pattern>
    `WHERE` <condition>
    `RETURN` <result values>
    `ORDER BY` <sort criteria>
  - `WHERE` and `ORDER BY` parts optional
  - Many variants for complex, expressive queries
  - `MATCH` tries to find graph paths that fit the pattern

- The pattern can define variable names for use in `WHERE`, `RETURN`, and `ORDER BY` parts
- The `WHERE`, `RETURN`, and `ORDER BY` parts are evaluated for each match
- `RETURN` only gives values if the `WHERE` condition evaluates to true
- Simple values, expressions, and aggregates (`count`, `sum`, `min`, `max`, `avg`) can be computed
- Matching algorithm avoids cycles and multiple matches of the same path

GQL (Cypher) Queries

- Example: matching any node

  `MATCH ()`

  Pattern parts in parantheses `(…)` match a node

- Optionally, a variable can be defined that binds to the matched nodes, one after the other

  `MATCH (n)`

- This variable can then be used in the rest of the query

- Example: finding the start node

```
MATCH (u :University
         { name: 'UKo'}
      )
```

  - `u` - variable name
  - `:University` - node label
  - `{ … }` property value(s)
  - Matches any node with the specified label and properties (multiple matches possible)

Step 2: Specify the search pattern

- In GQL, the pattern consists of a path description
- Alternate sequence of node and relation (edge) parts, ends with a node
- `-->` `<--` `---`
  `-[…]->` `<-[… ]-` `-[…]-`
  match a relation, with and without considering the direction
- `-[:Type]-` looks for an edge of the specified type
- At most one type can be matched

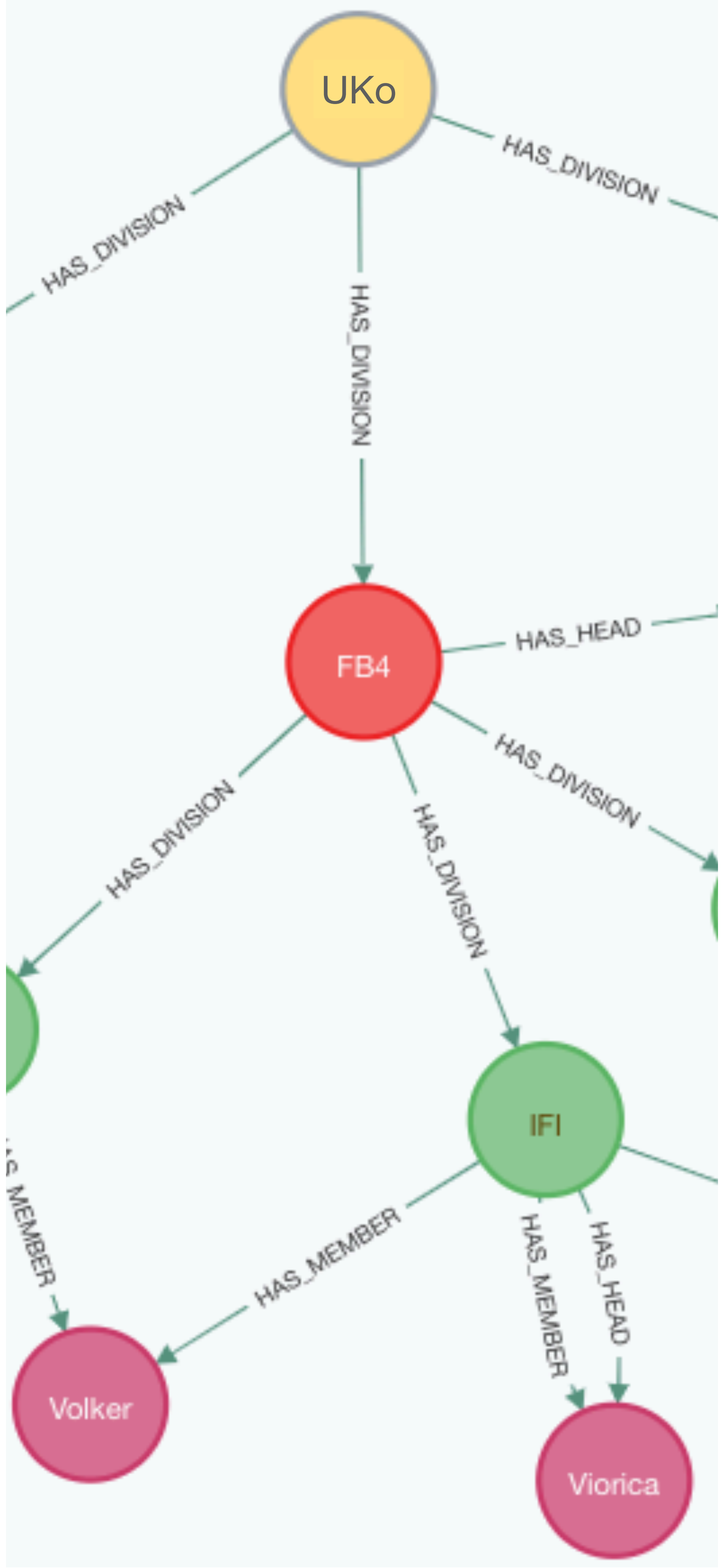
- Variables and properties can be specified as with node patterns
- Match repeated edges (optionally with type and direction)

  `-[*min..]-`
  at least `min` occurences

  `-[*min..max]-`
  at least `min`, at most `max` occurences
- `-[*1..5]->` matches 1 to 5 outgoing edges

Example: Look for Employees

- In our example, we need to find a path from the start node "UKo"
- To find arbitrary deeply nested employees, we need to match
  a sequence of HAS_DIVISION relations,
  then a HAS_MEMBER or HAS_HEAD relation,
  and then the employee node
- ```
  MATCH (u :University {name: 'UKo'})
     -[:HAS_DIVISION*0..]-> ()
     --> (e :Employee)
  ```


  ```

Step 2: Specify the search pattern

- Optionally, the `WHERE` part can contain a boolean expression denoting a condition on the variables defined in the path pattern
- Only matches that fulfill the condition will be processed
- Conditions can be combined by logical operators `NOT`, `AND`, `OR`
- For example, we could look for employees whose name starts with “Ma”
-

```
MATCH (u :University {name: 'UKo'})
  -[:HAS_DIVISION*0..]-> ()
  --> (e :Employee)
WHERE e.name STARTS WITH ‘Ma’
```

Step 3. Describe desired result

- The `RETURN` part of a GQL query defines the result
- `RETURN` can be followed by one or more expressions
  - scalar values
  - node or relation variables
  - property access
  - operator expressions
  - aggregate functions
  - …
- Duplicates in the result can be eliminated by `DISTINCT`
- Optionally, the result may be ordered with `ORDER BY`

Example: Get all Employees in ascending Order

```
MATCH (u :University {name: 'UKo'})
  -[:HAS_DIVISION*0..]-> ()
  --> (e :Employee)
RETURN e
ORDER BY e.name
```

- Returns matched nodes (not only the names)
- Duplicates due to multiple matching paths

"e"
{"name":"Claudia"}
{"name":"Jan"}
{"name":"Jan"}
{"name":"Maria"}
{"name":"President"}
{"name":"President"}
{"name":"President"}
{"name":"Steffen"}
{"name":"Steffen"}
{"name":"Steffen"}
{"name":"Viorica"}
{"name":"Viorica"}
{"name":"Volker"}
{"name":"Volker"}

```
{
  "identity": 7,
  "labels": [
    "Employee"
  ],
  "properties": {
"name": "Claudia"
  }
}
```

detailed view
of 1st result

Example: Get all Employees in ascending Order

```
MATCH (u :University {name: 'UKo'})
   -[:HAS_DIVISION*0..]-> ()
   --> (e :Employee)
RETURN DISTINCT e
ORDER BY e.name
```

"e"
{"name":"Claudia"}
{"name":"Jan"}
{"name":"Maria"}
{"name":"President"}
{"name":"Steffen"}
{"name":"Viorica"}
{"name":"Volker"}

- Eliminating duplicates by `DISTINCT` finally gives the desired result

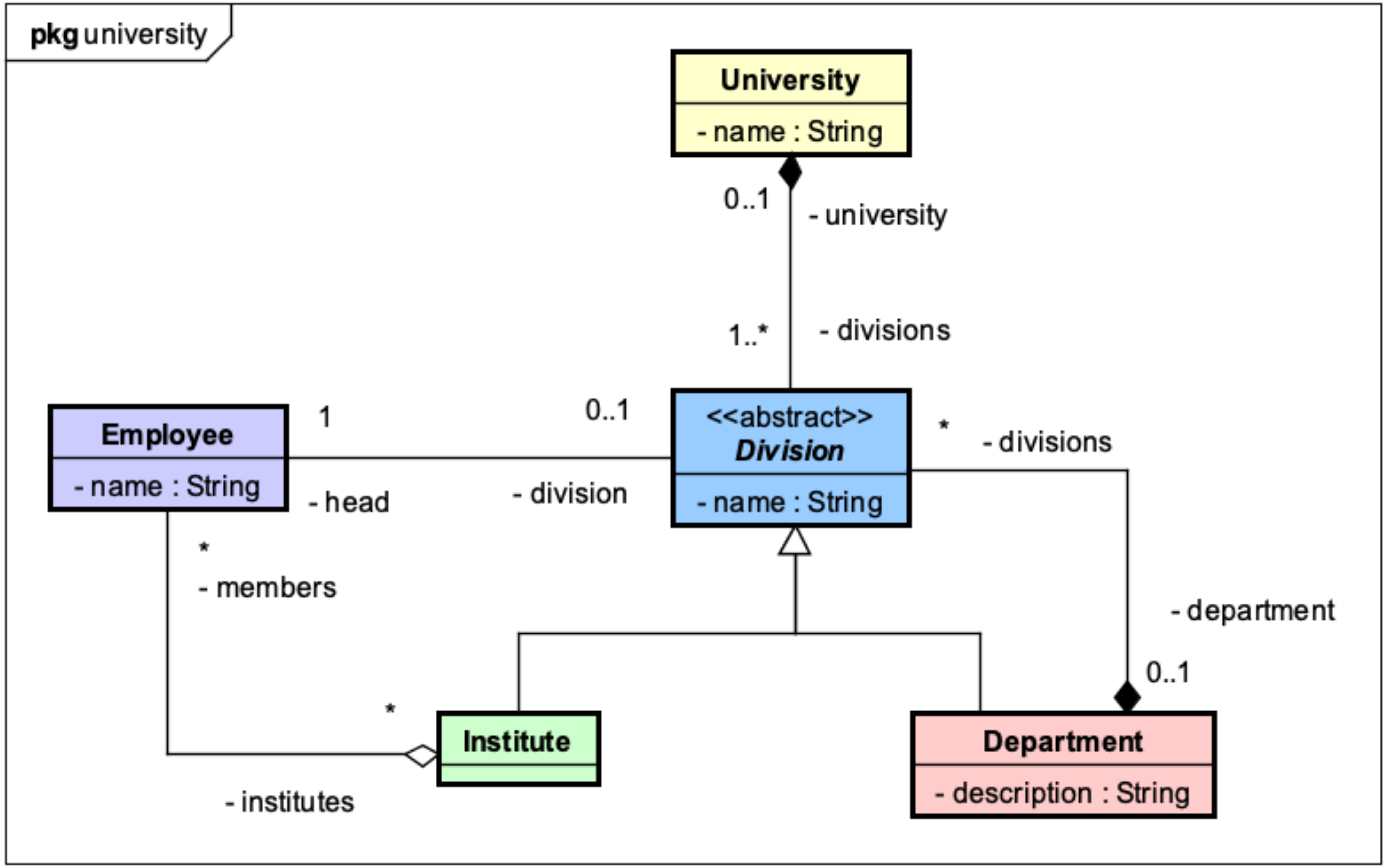
More options…

- GQL is a fully fledged query language with complex syntax
- Not only querying, but also creation, modification, and removal of nodes and edges
- More examples will be provided in the tutorial sessions
- Also consider the **step-by-step tutorial tours** of Neo4J
- Specification can be found at https://neo4j.com/docs/cypher-manual/current/

5.4.5 Graph Schemas

→ Object - Graph Mapping

University Domain Model

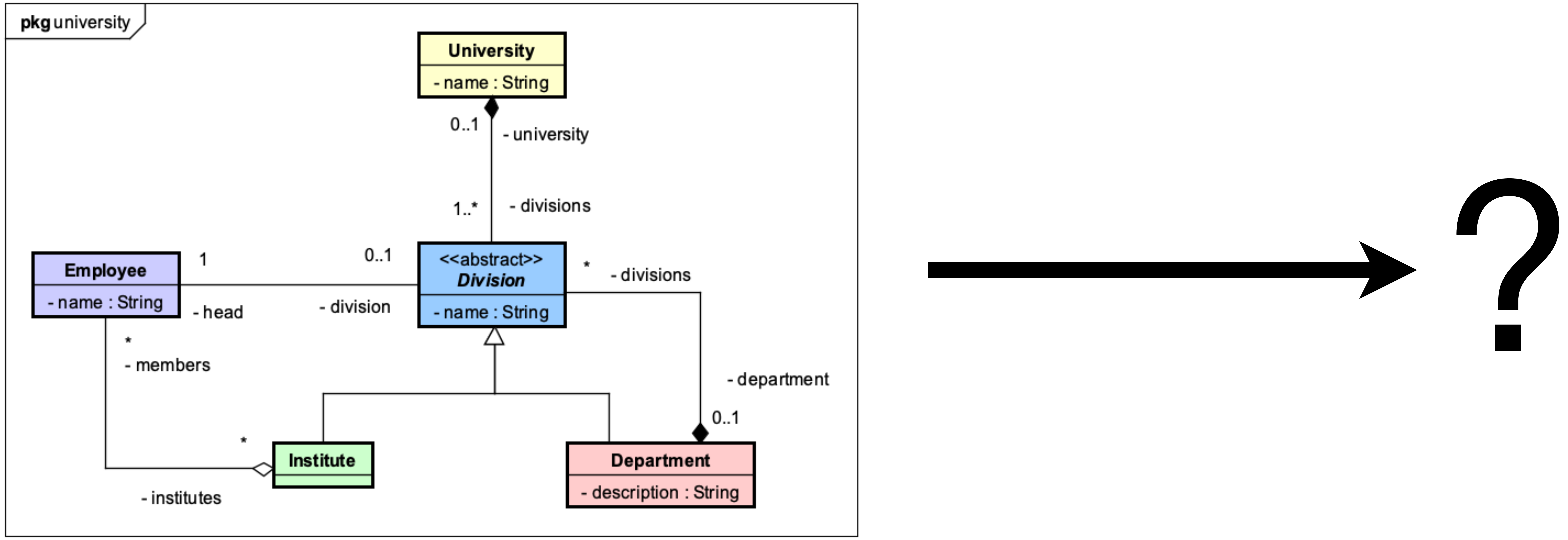


How to map an OO schema to a graph schema

- Impedance mismatch similar to Object-Relational mapping
- Properties of graph model determine mapping strategies and alternatives
- Three steps (as with O/R mapping) have to be conducted:

1. **Schema**: map OO domain models to labeled property graphs
2. **Instance**: represent objects and relations as nodes and edges
3. **Processing**: store new objects in graph, retrieve objects from graph

How to map an OO schema to a graph schema



Graph Schemas

- Given a specific graph type, graphs can be constrained (or defined) by a graph schema
  - graph schema defines types, attributes, constraints, and other features
  - given a schema, the set of instances - all graphs that correspond to the schema - can be defined
  - a schema restricts the possible instance graphs
- Schemas are essential to make use of graphs and to formulate and calculate meaningful queries efficiently
- Strong advice: read this general discussion on graph data modeling

  https://neo4j.com/developer/data-modeling/

UML class diagrams vs. Labeled Property Graphs

- The graph model misses features of OO models:
  - role names
  - multiplicity constraints
  - abstract types
  - generalization

Recall: Features of Labeled Property Graphs

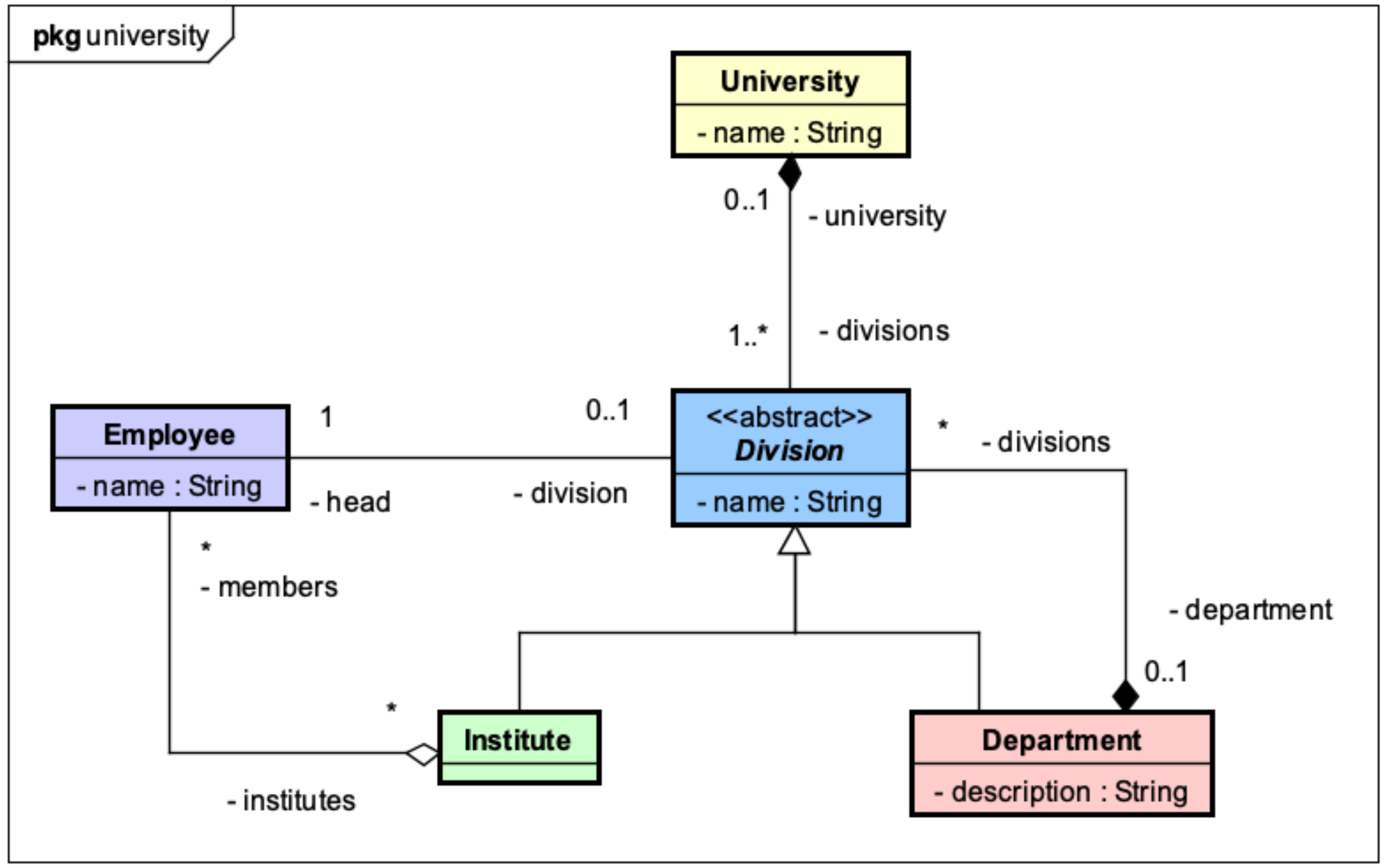
- Labeled nodes
  a node can have multiple labels
- Typed, directed, binary edges
  can be traversed in both directions
- Properties (attributes) for both nodes and edges stored as key-value pairs
- Traversal of incident edges to adjacent nodes is constant-time operation

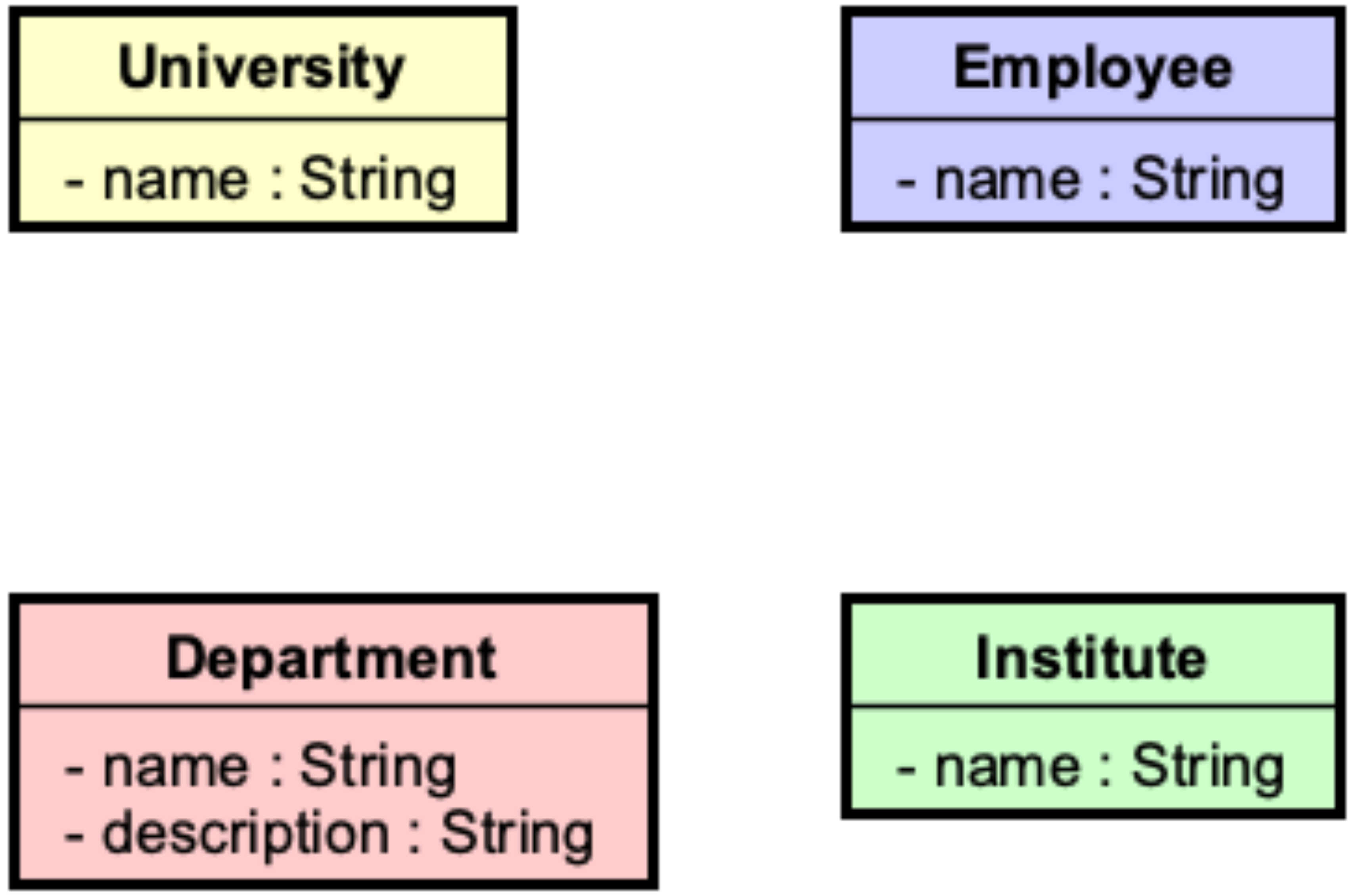
- No ordering
- No generalization in type system
- No hyper edges
- No hierarchy
- No consistency constraints, e.g. on uniform properties per label and/or type

- Several decisions have to be taken to map domain models; the following pages describe a possible solution
- We also use UML class diagrams to model graph schemas

Mapping Classes and Attributes

- Classes can be mapped straightforward to node types
- Class name → node label
- Attributes → properties
- No mapping of abstract classes
- Inherited attributes have to be added to subclasses



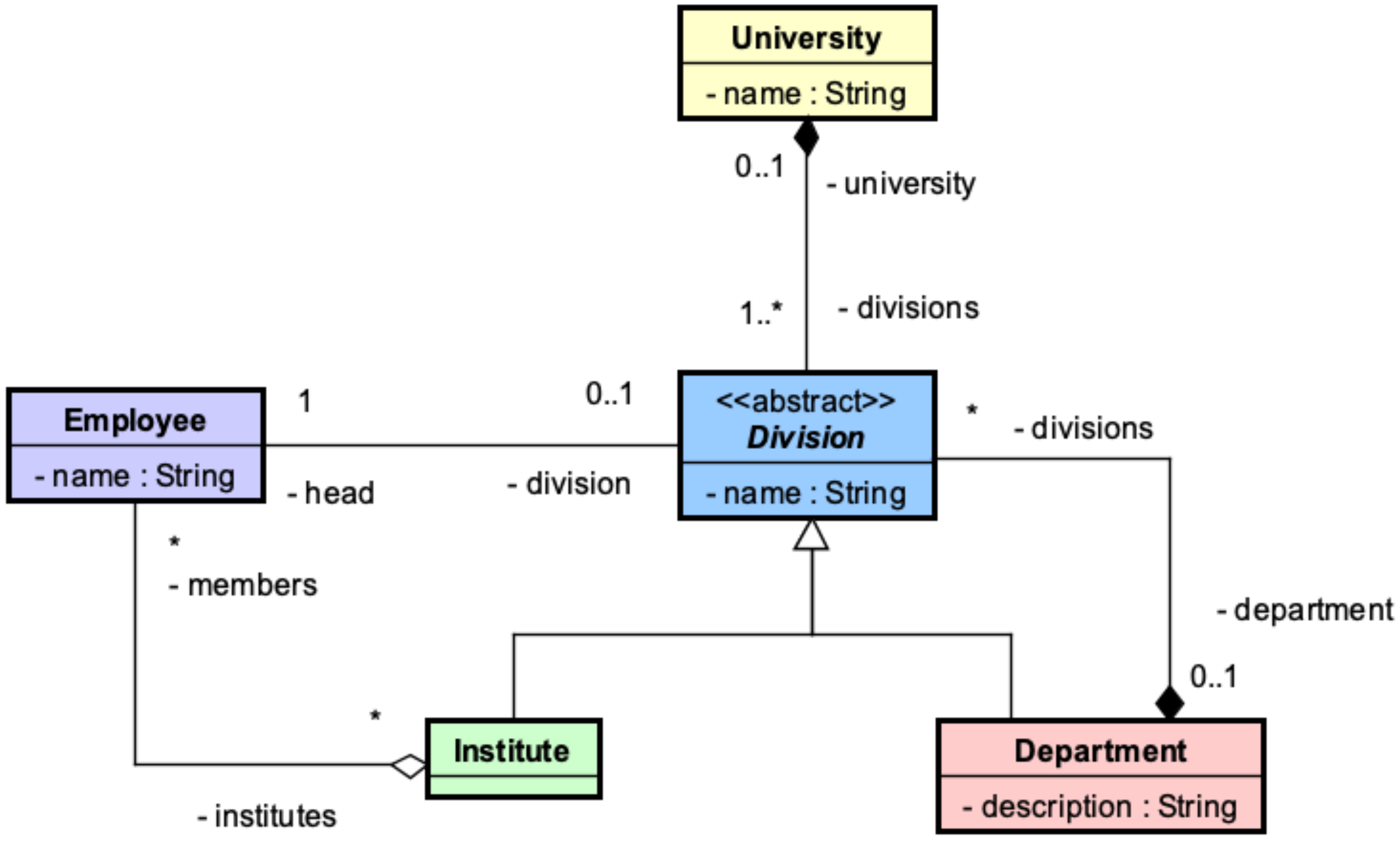


Mapping Associations

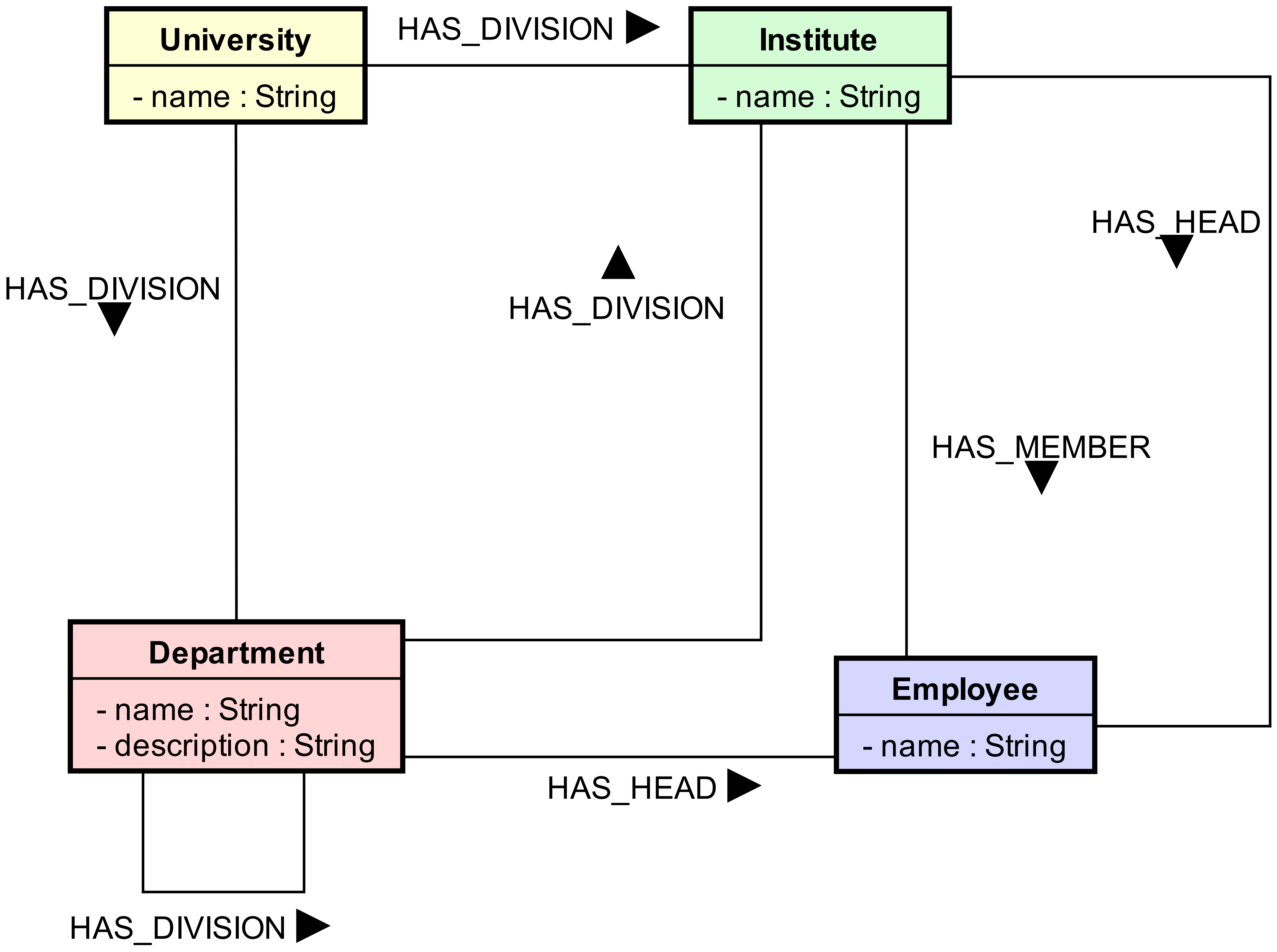
- For each association, we need to create an edge type
- Usually, one of the role names can be used as type name, prefixed by a verb
- Specify direction of edges
  - **Aggregations** and **compositions** from container to elements
  - Simple **associations** pick meaningful direction, take uniform decisions

- Associations of abstract classes
  - Abstract class not part of graph schema
  - Need to copy edge types for all non-abstract subclasses
- Multiplicities
  - Dropped…
  - Maintaining multiplicity constraints in the graph is up to the applications

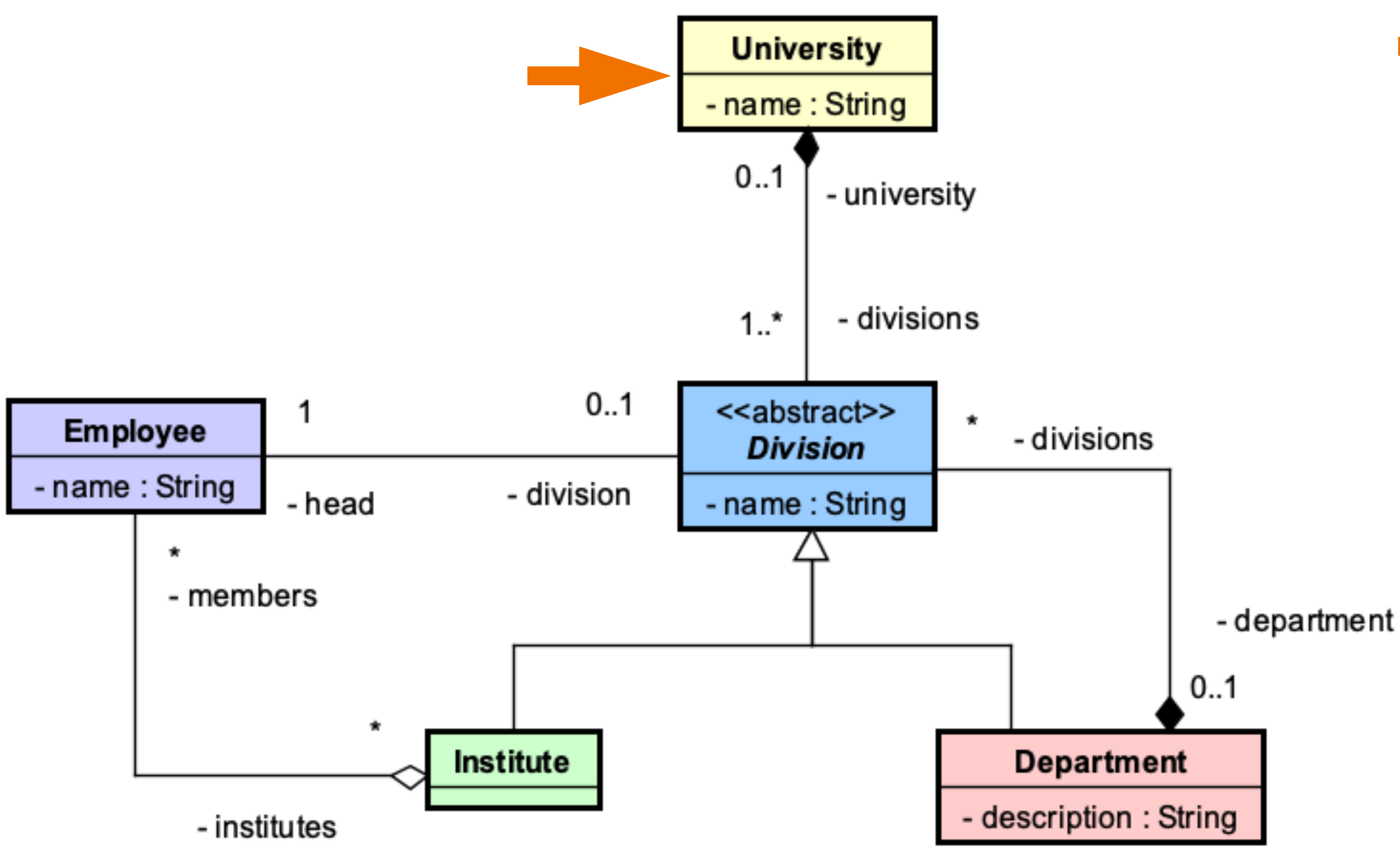
OO Model



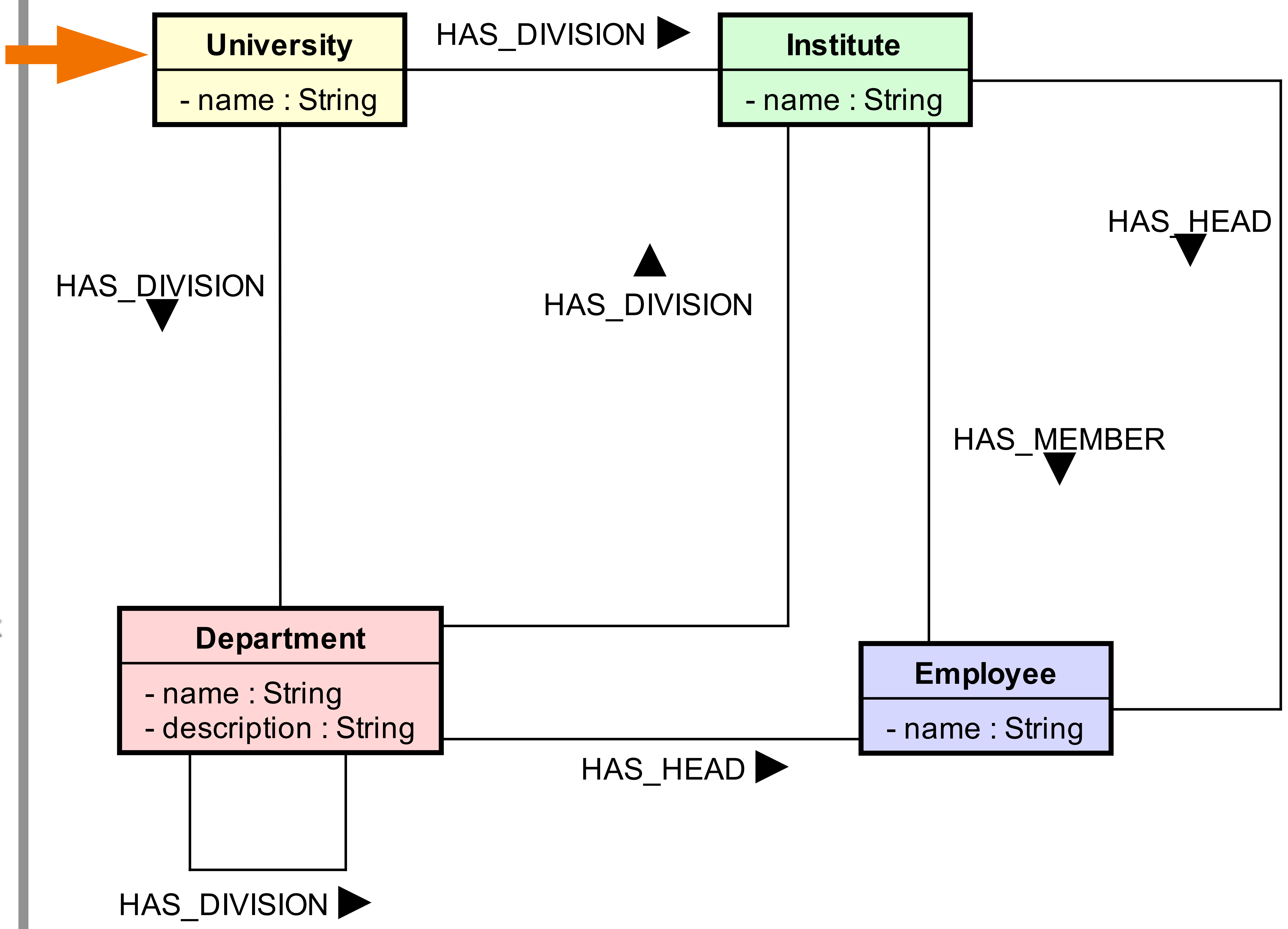
Graph Model



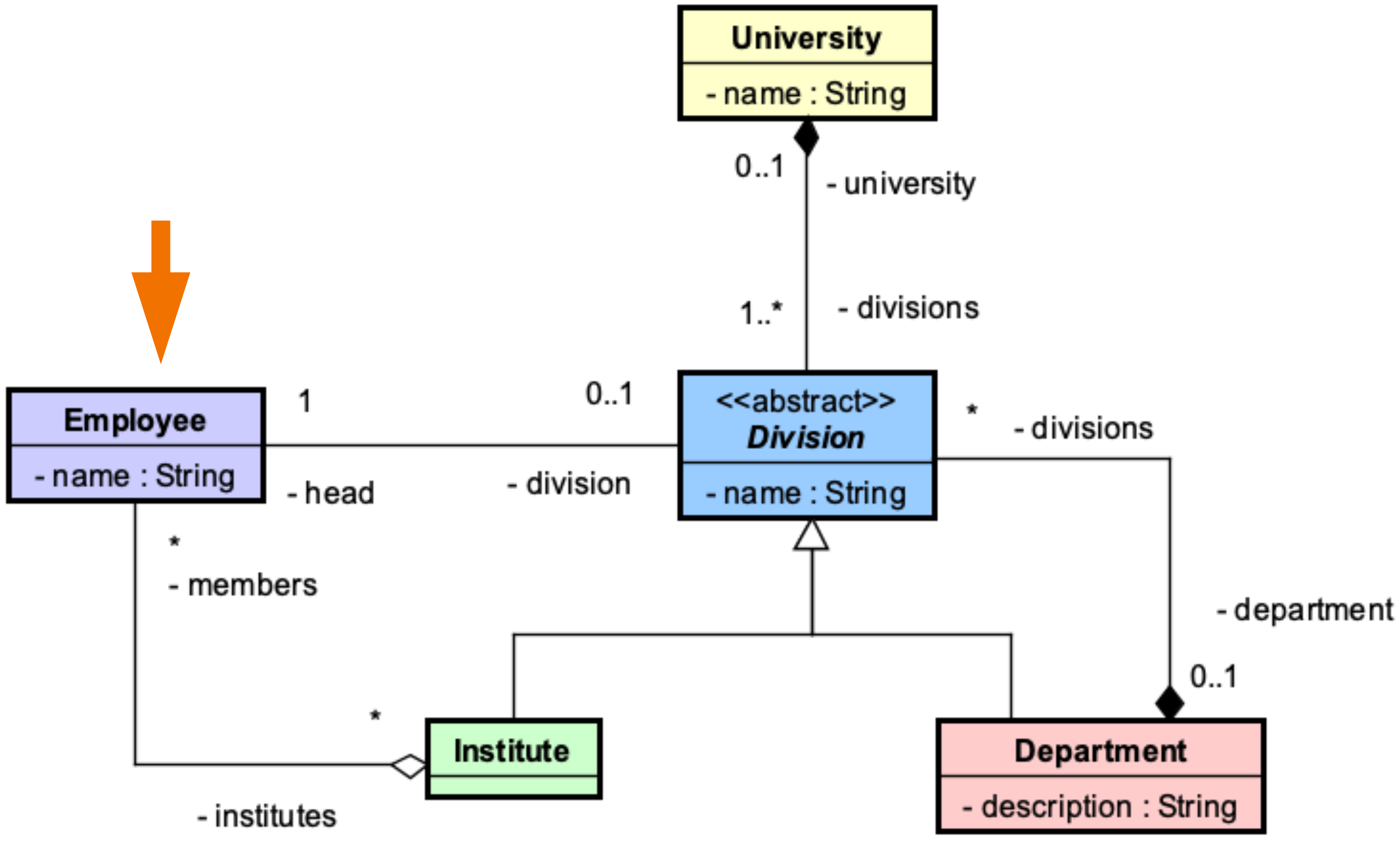
OO Model



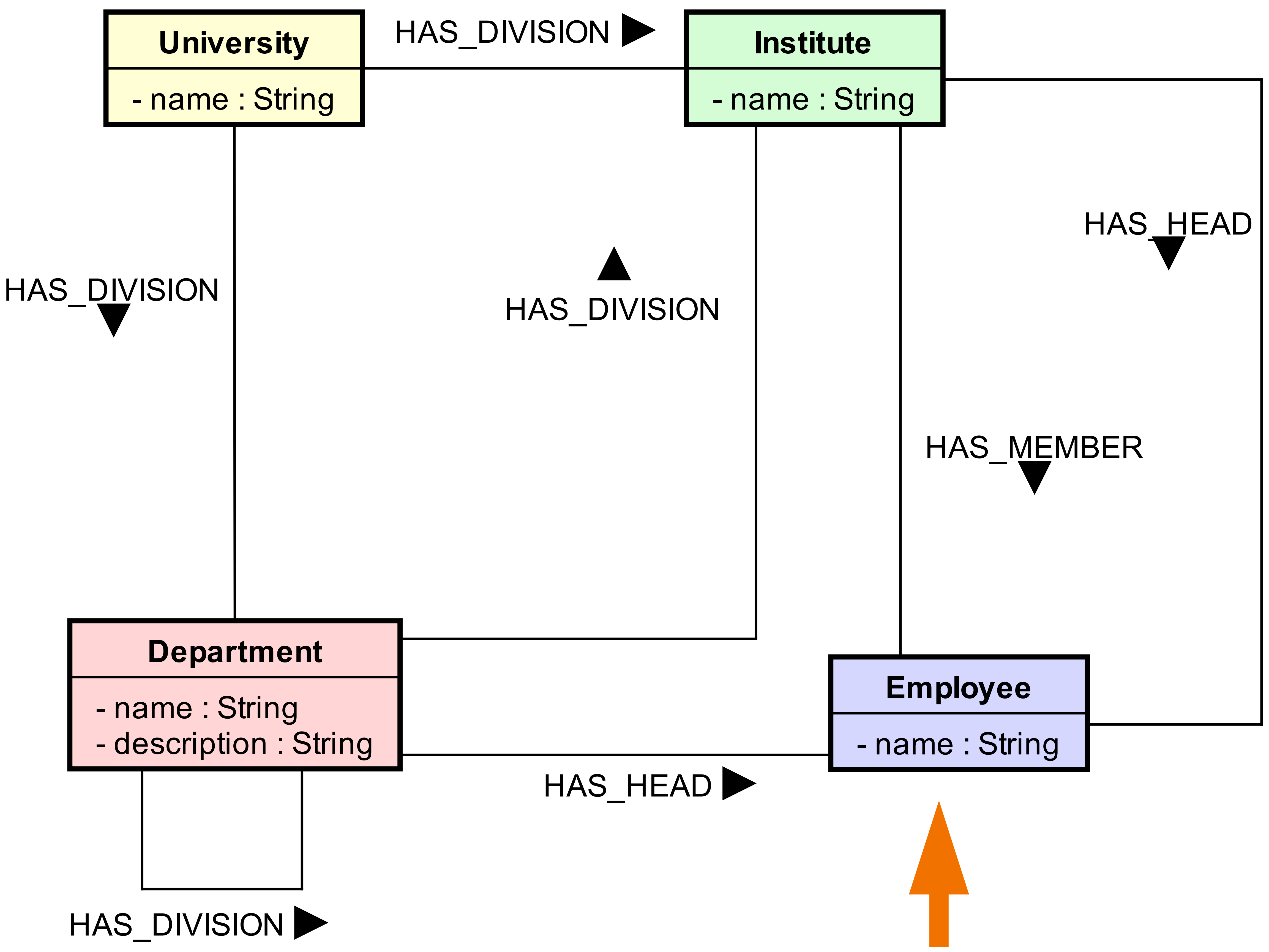
Graph Model



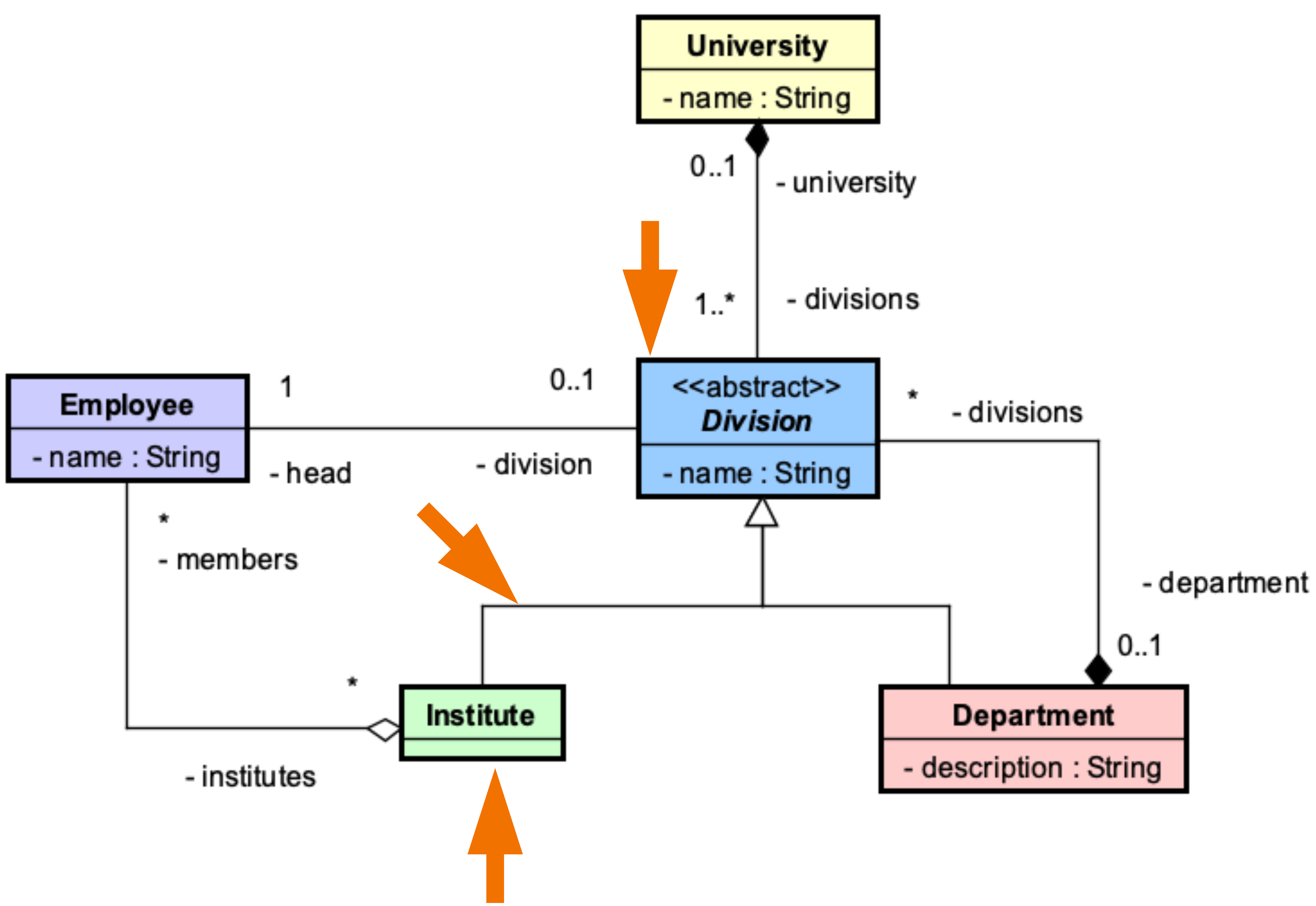
OO Model



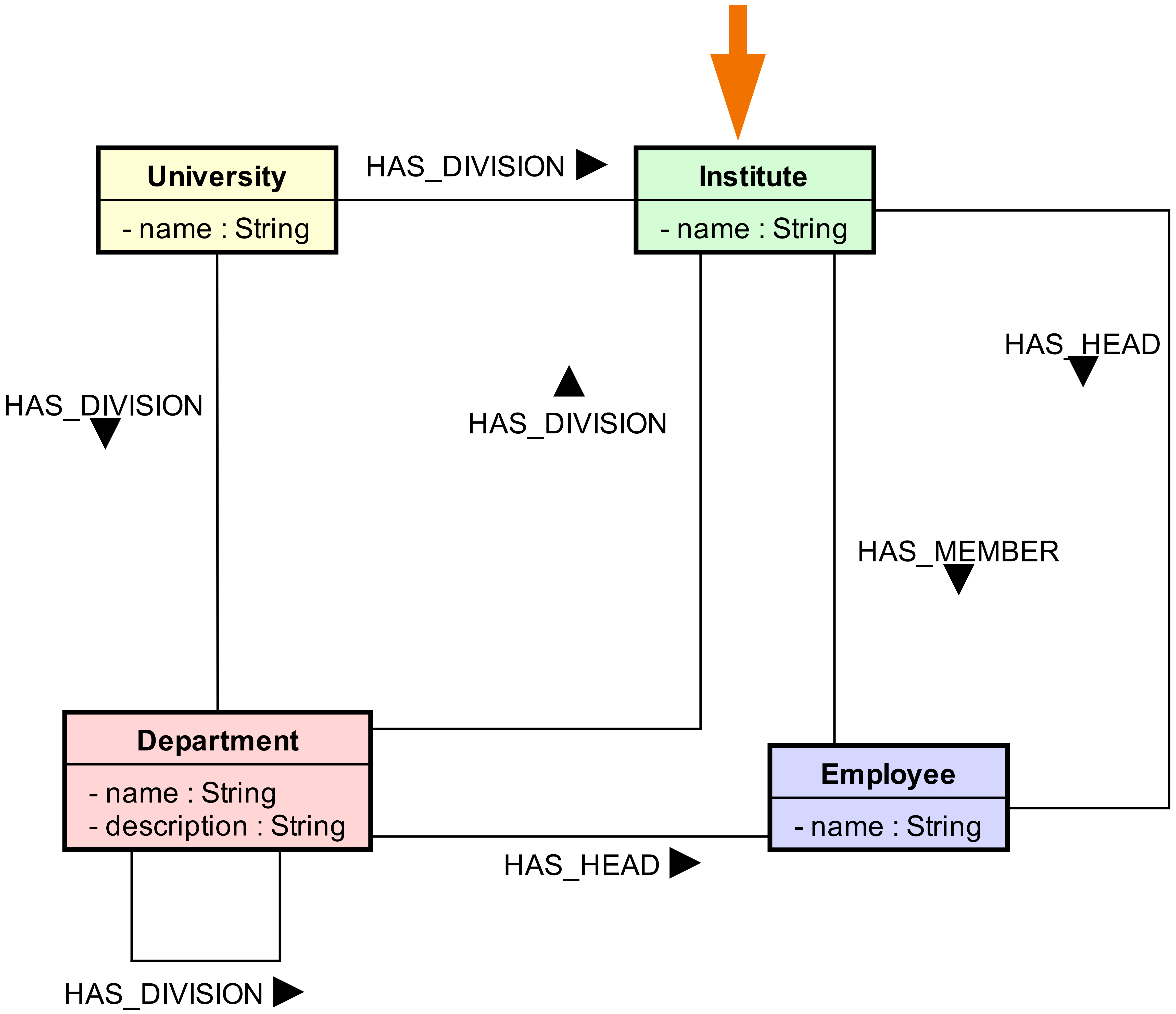
Graph Model



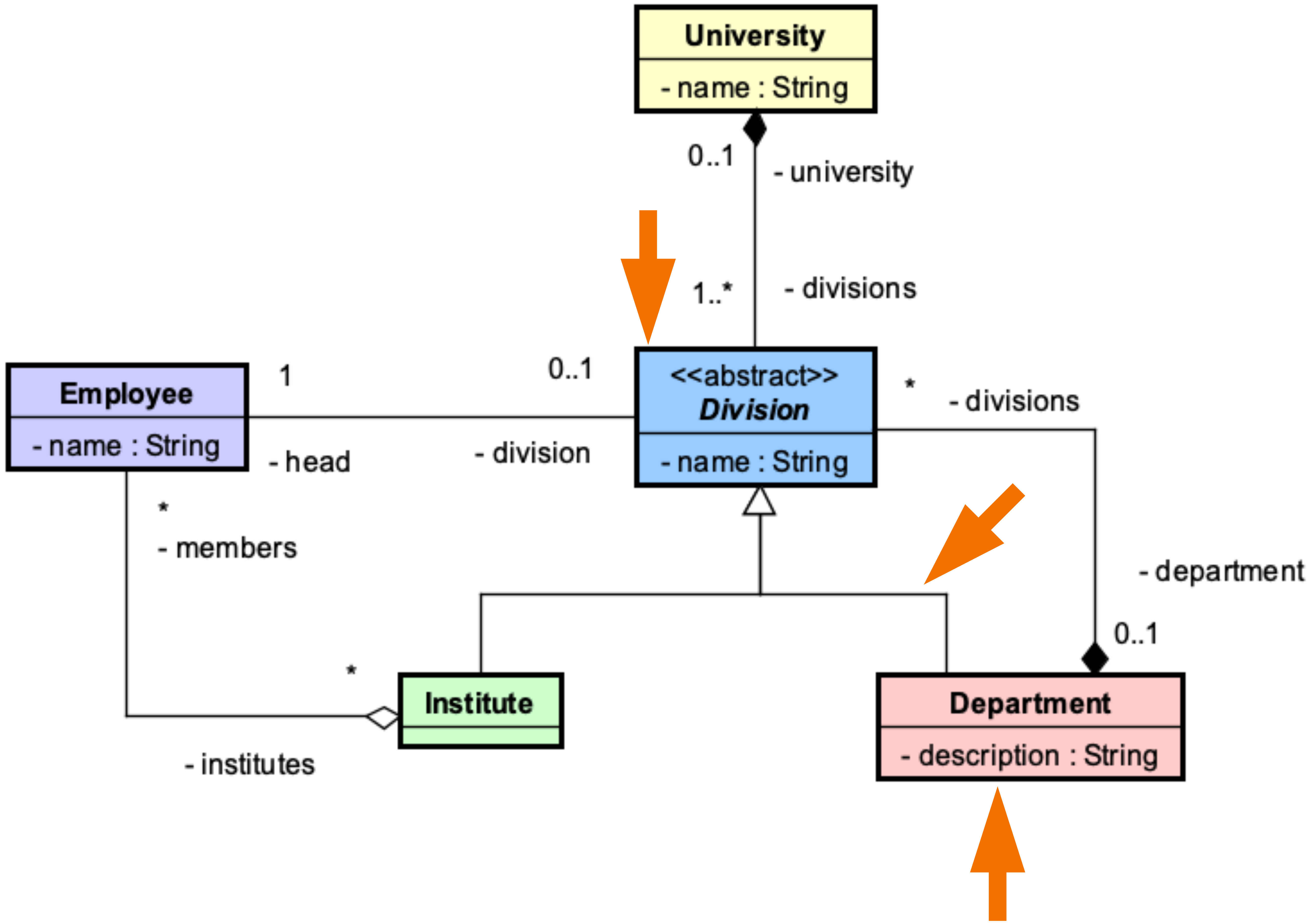
OO Model



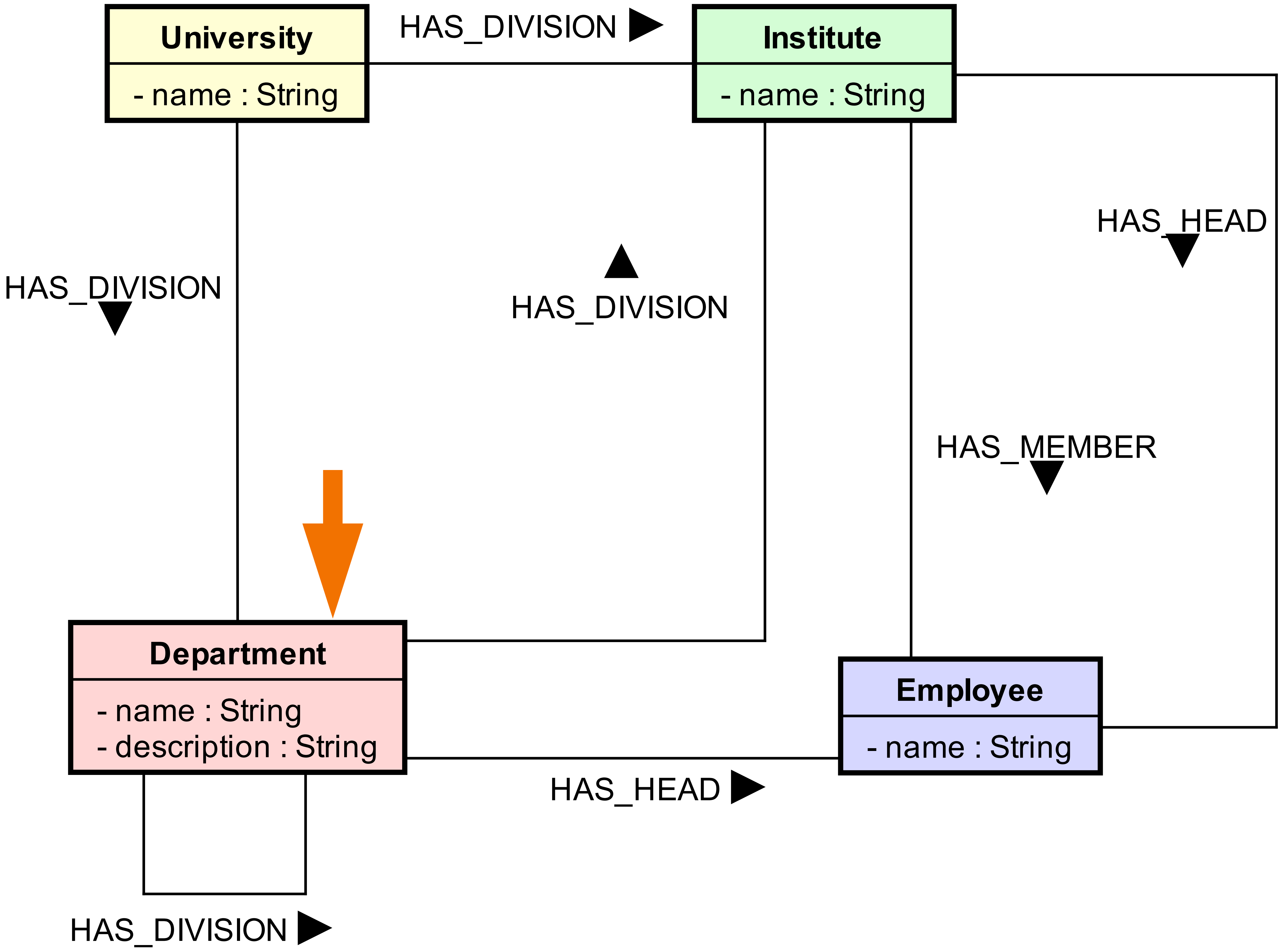
Graph Model



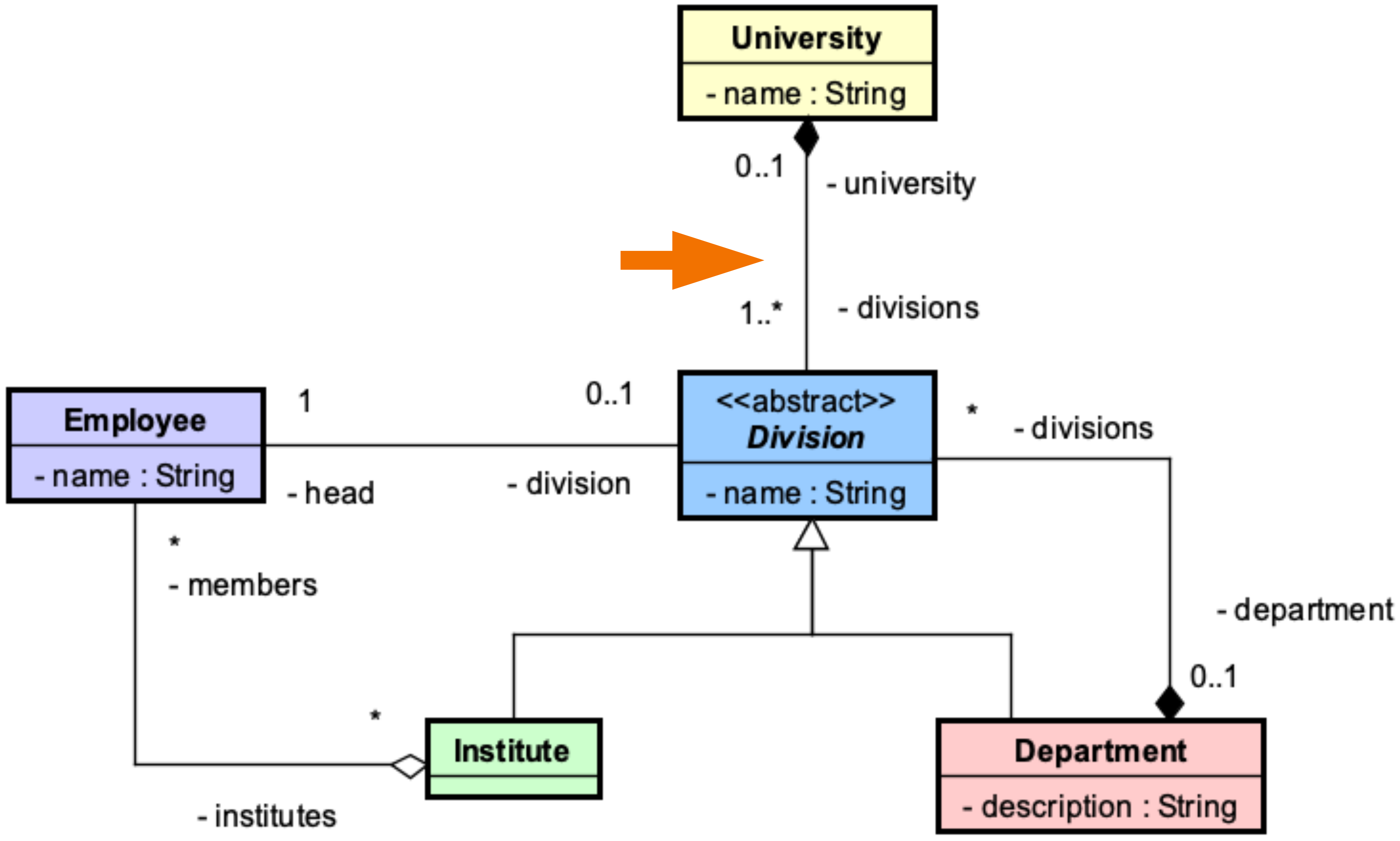
OO Model



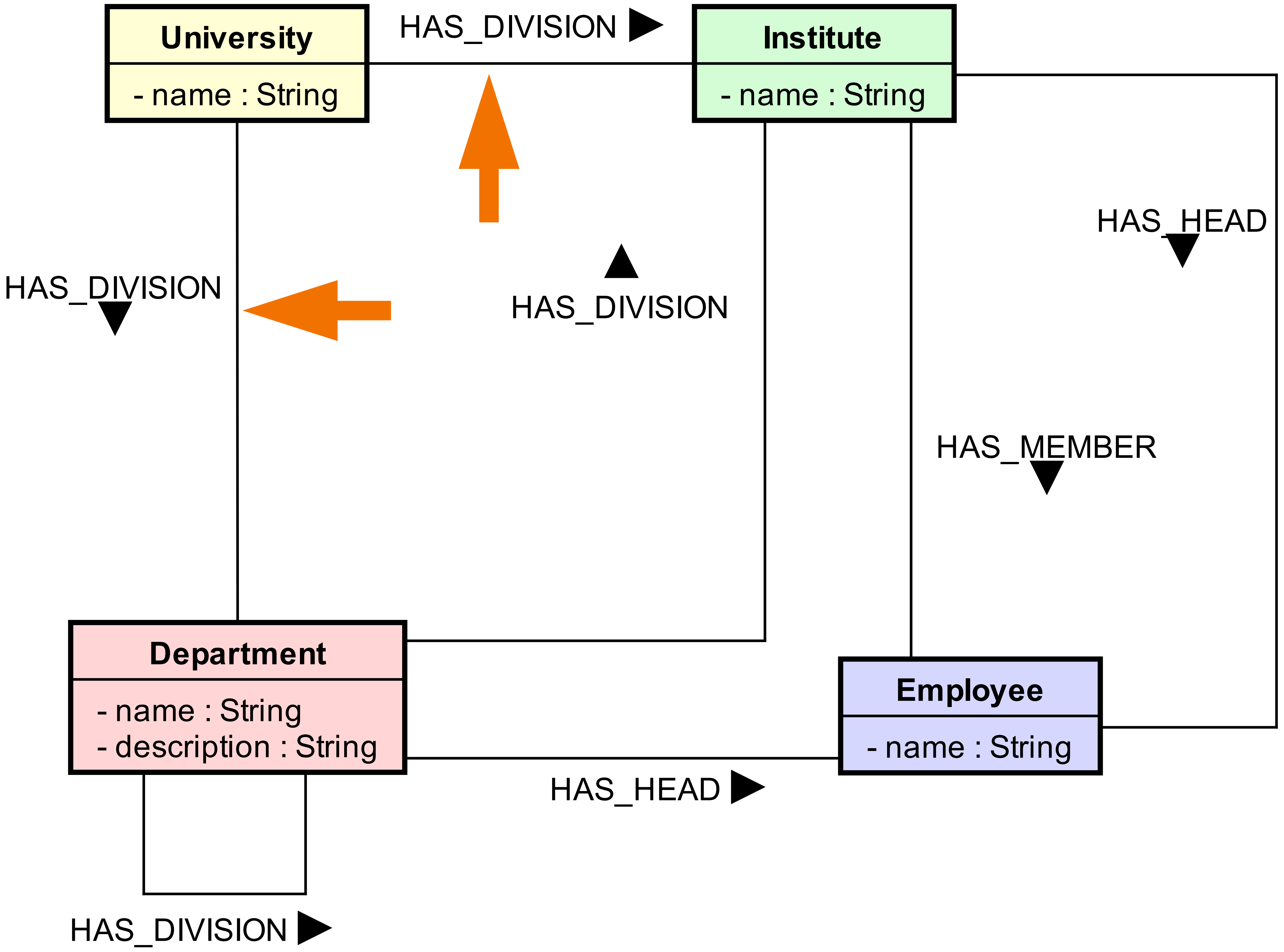
Graph Model



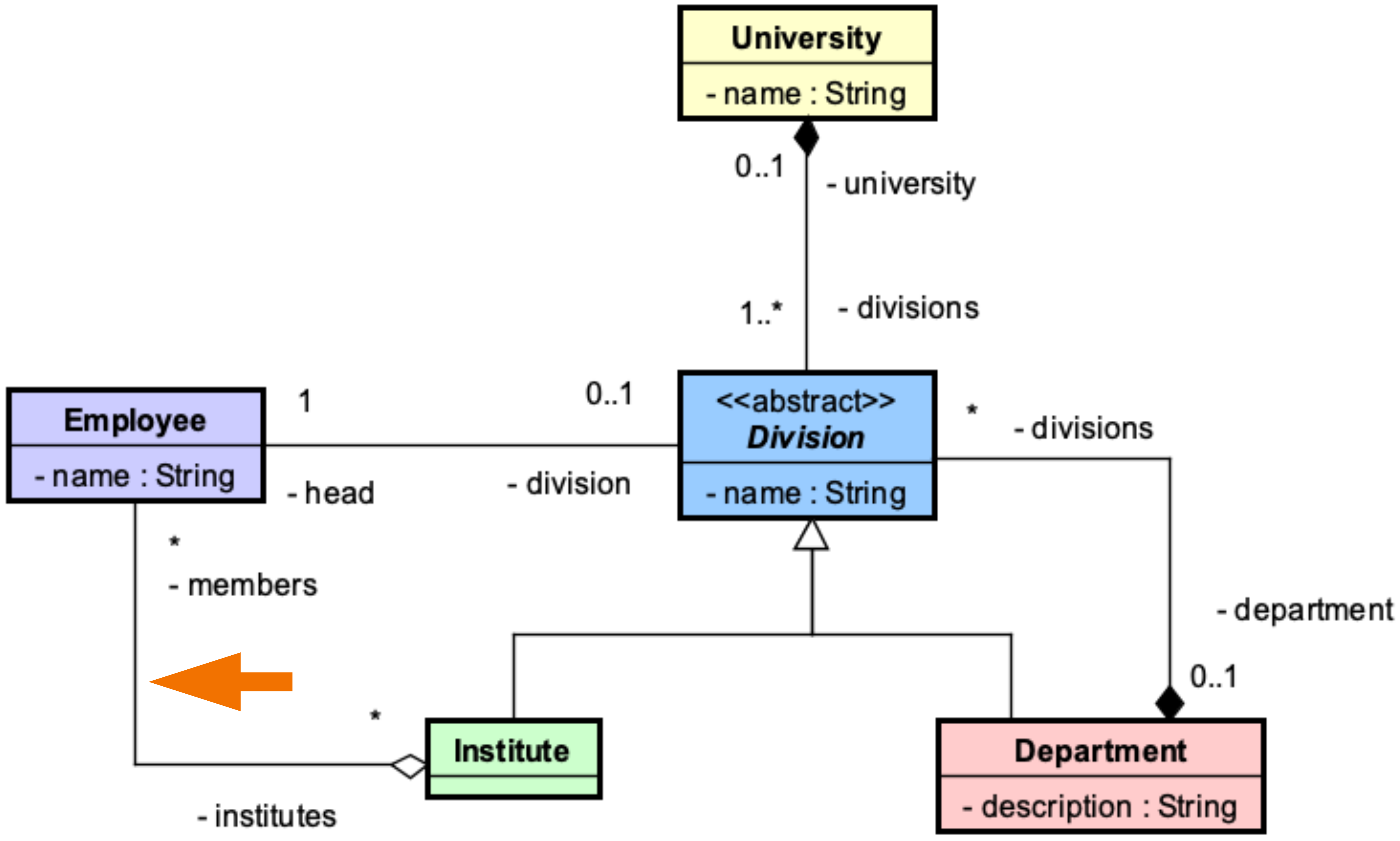
OO Model



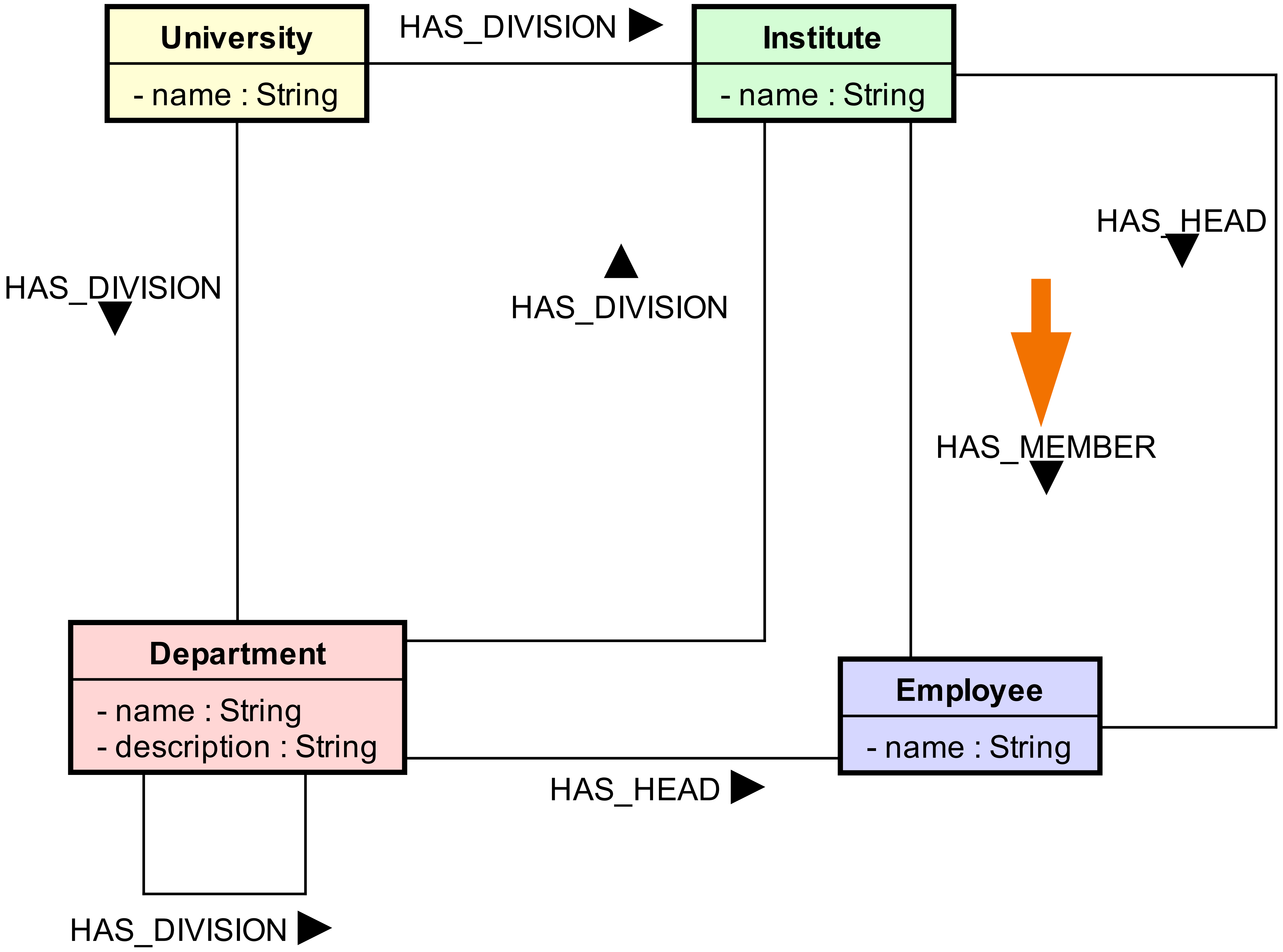
Graph Model



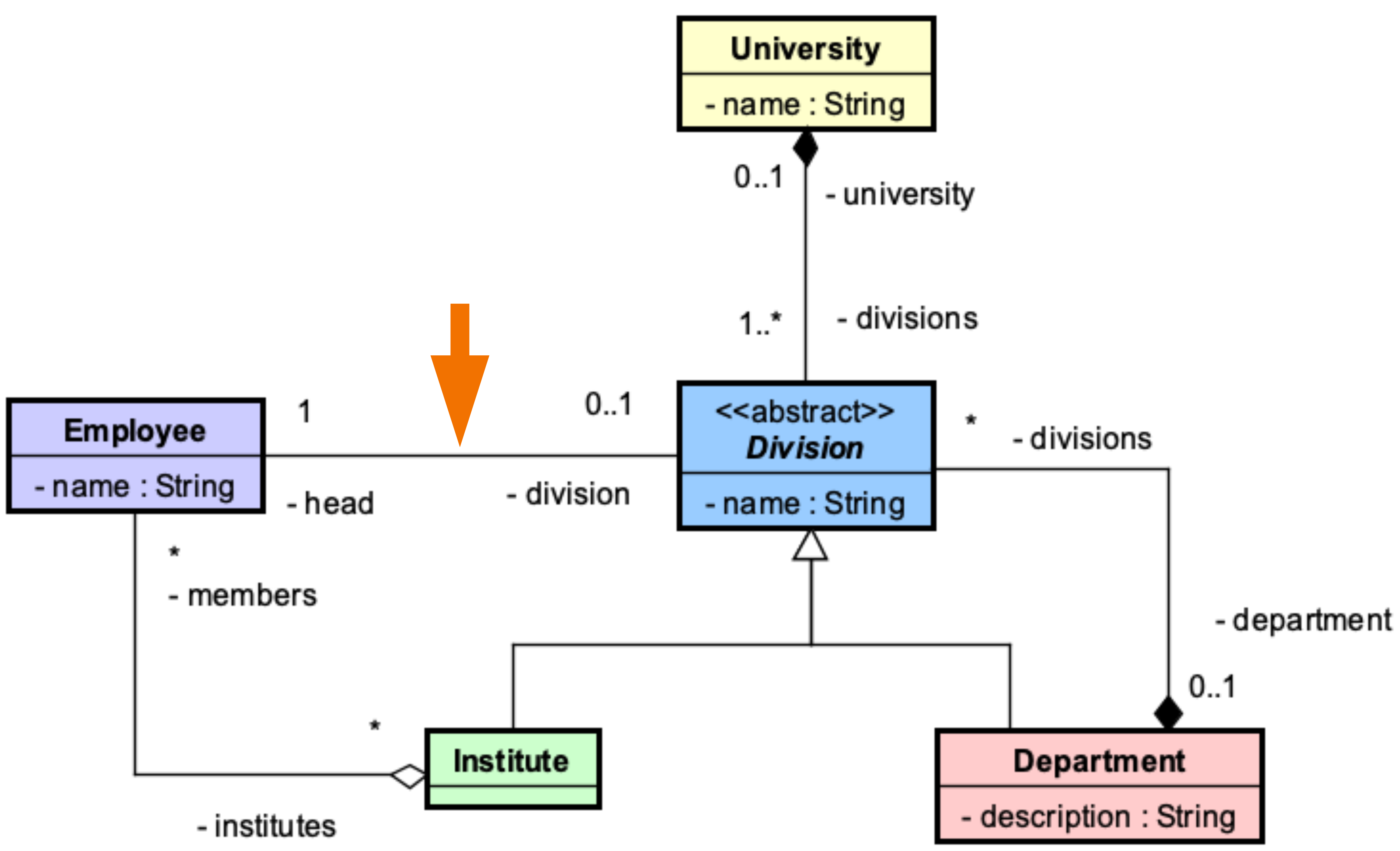
OO Model



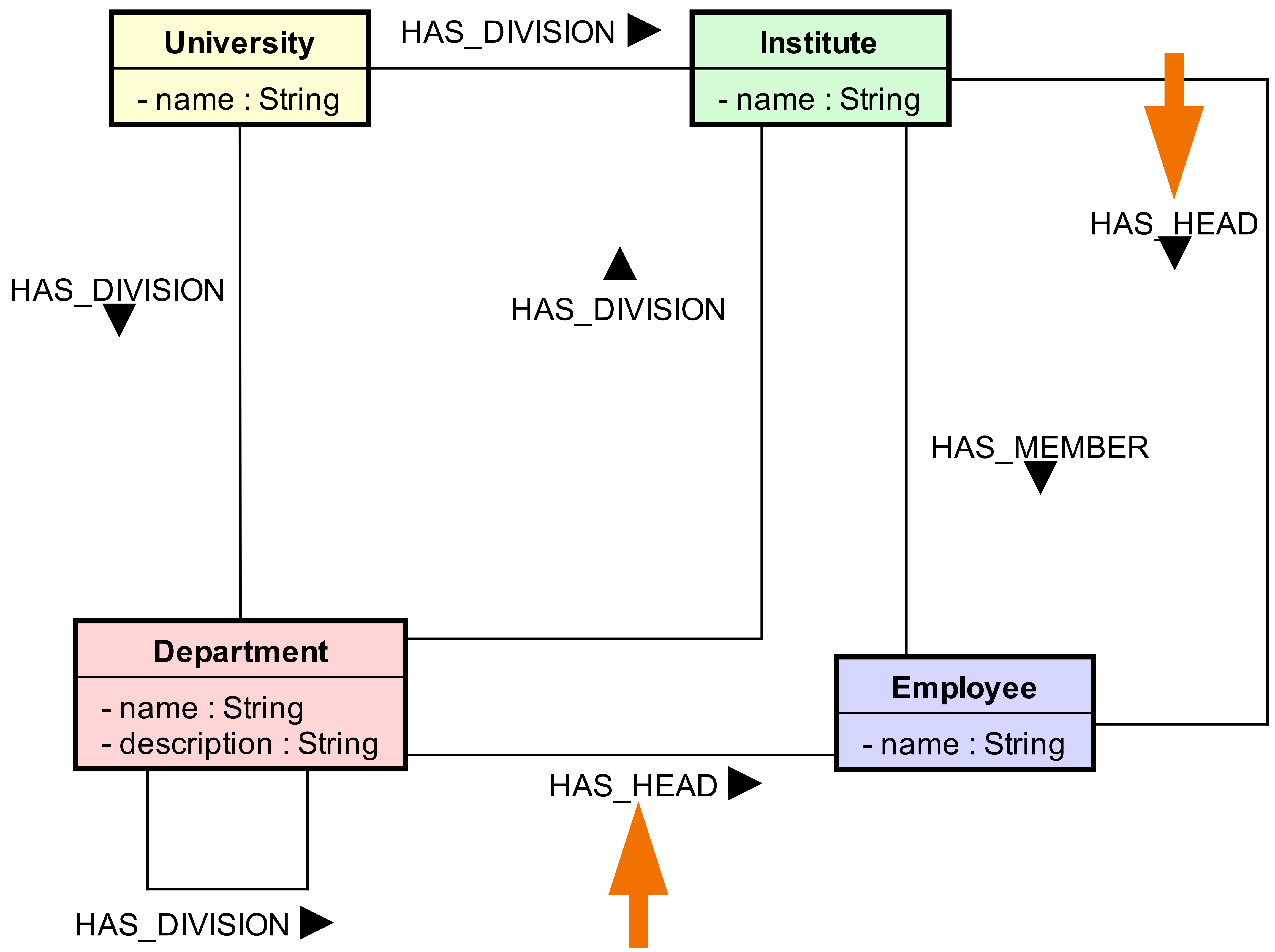
Graph Model



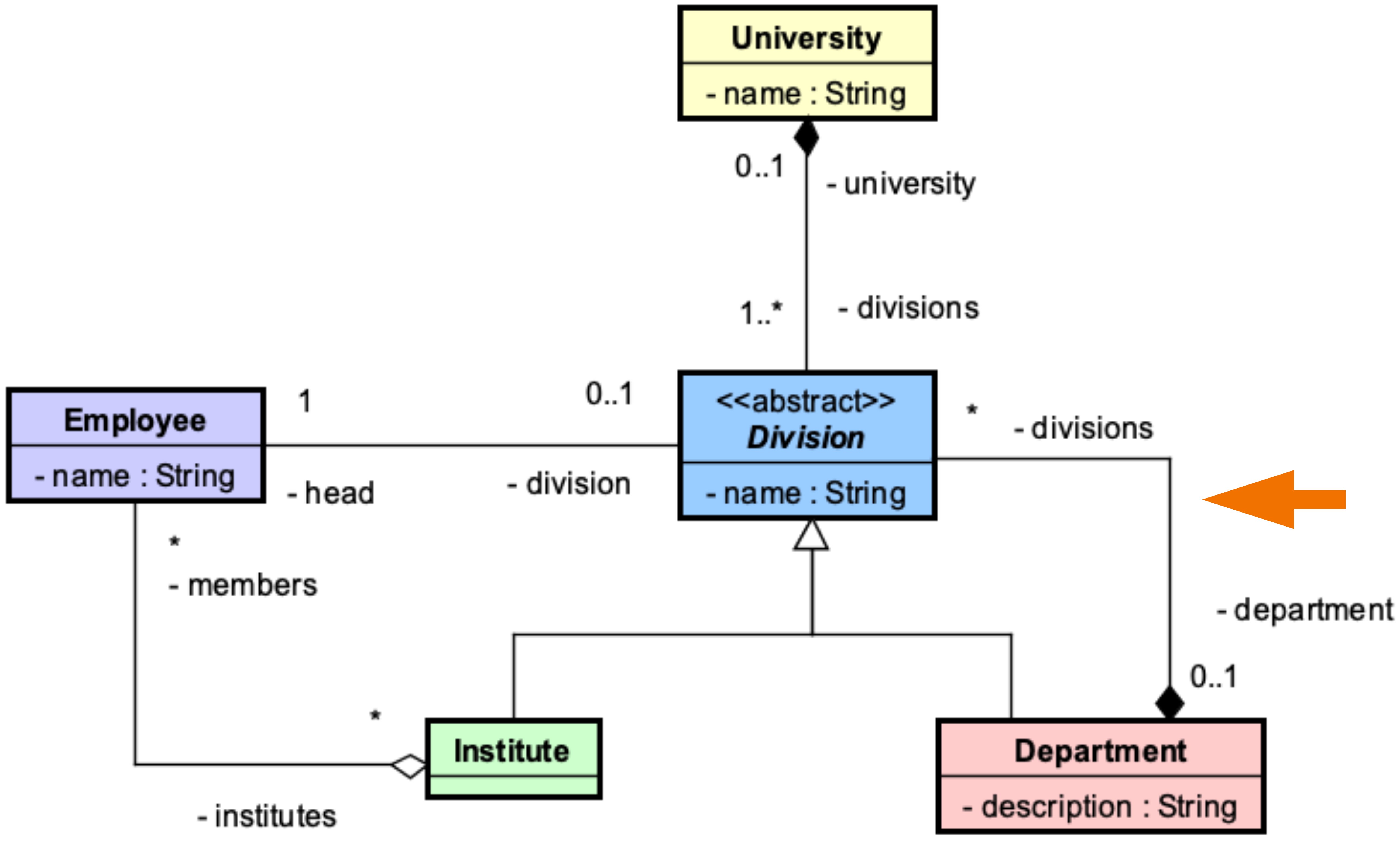
OO Model



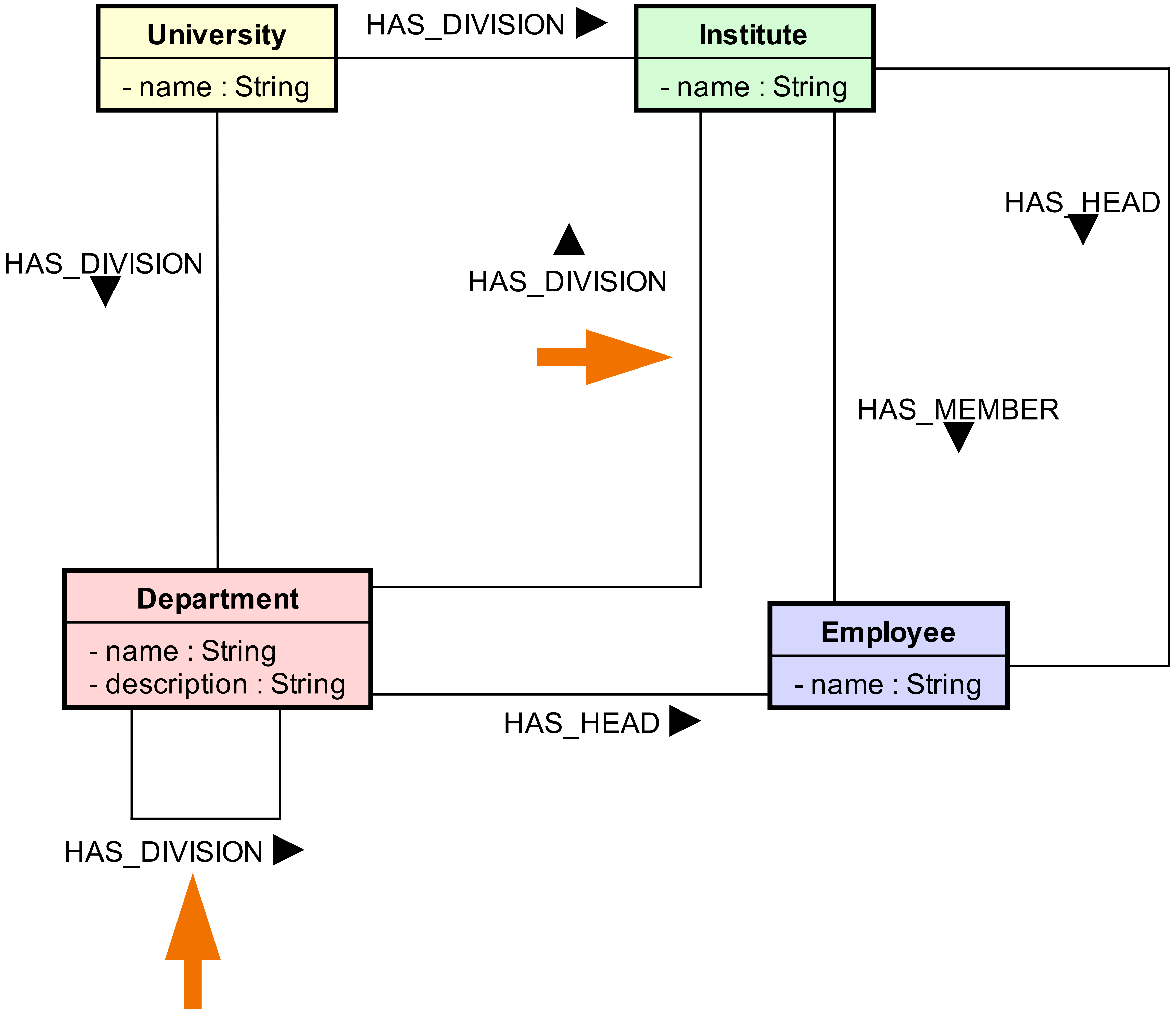
Graph Model



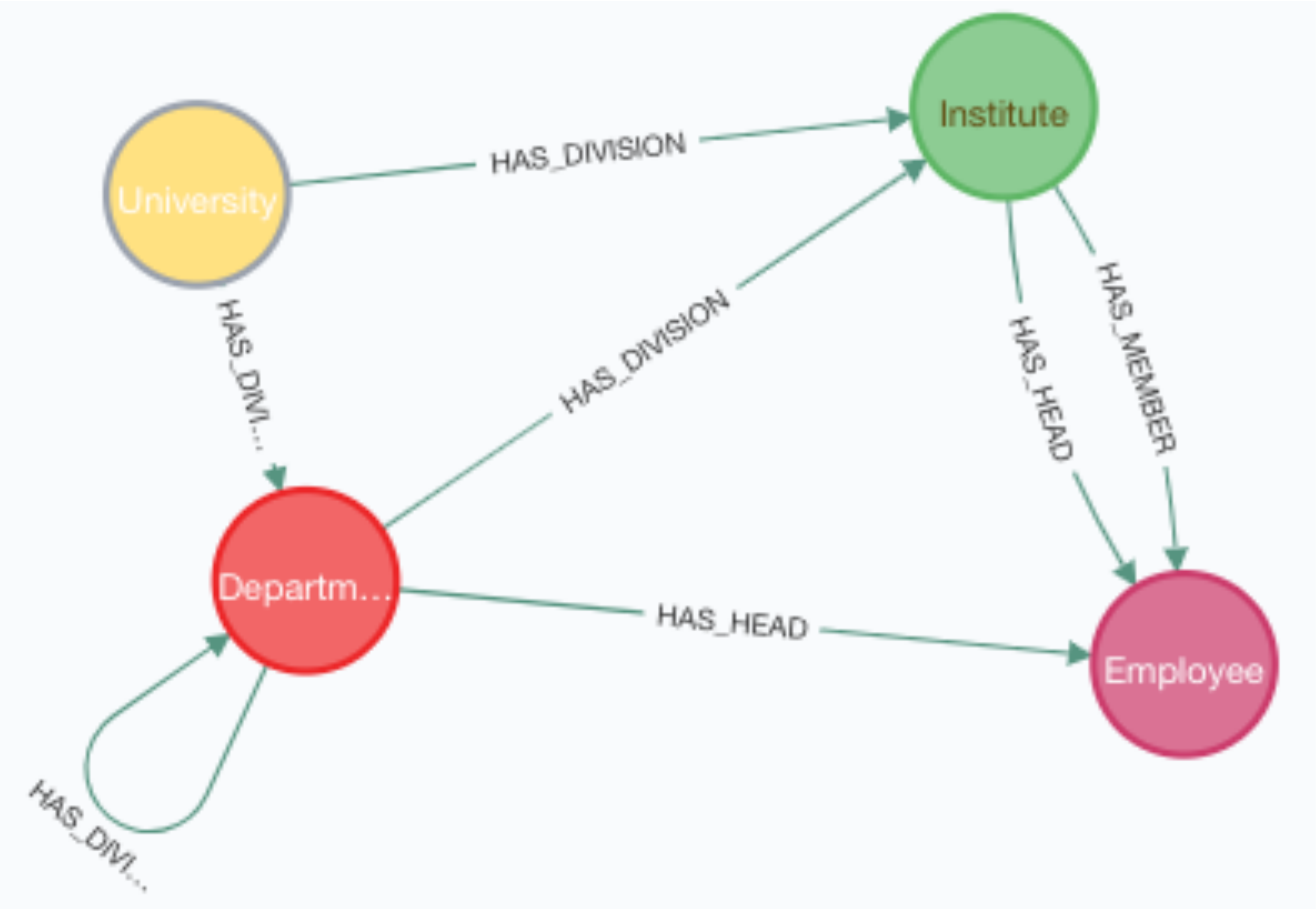
OO Model



Graph Model

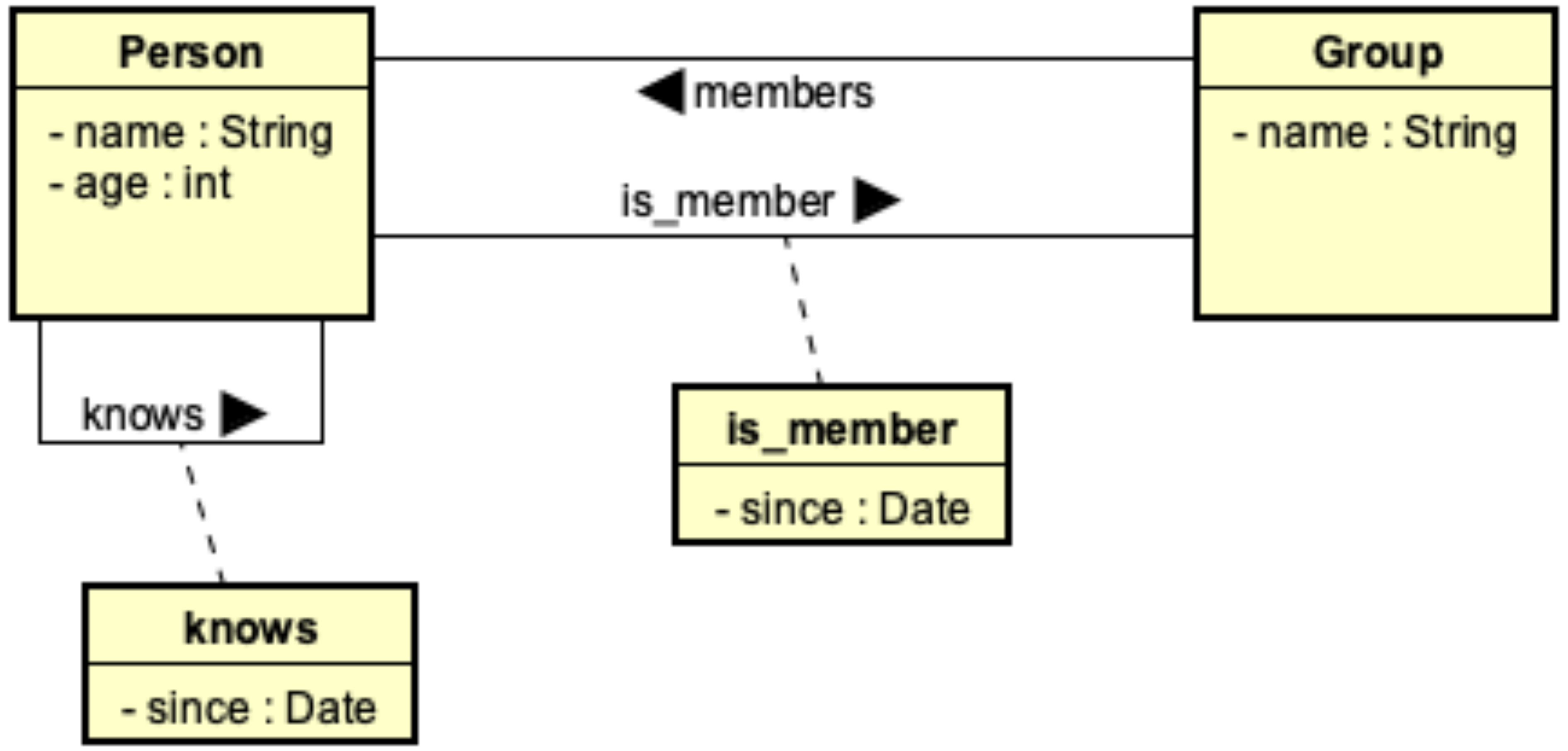


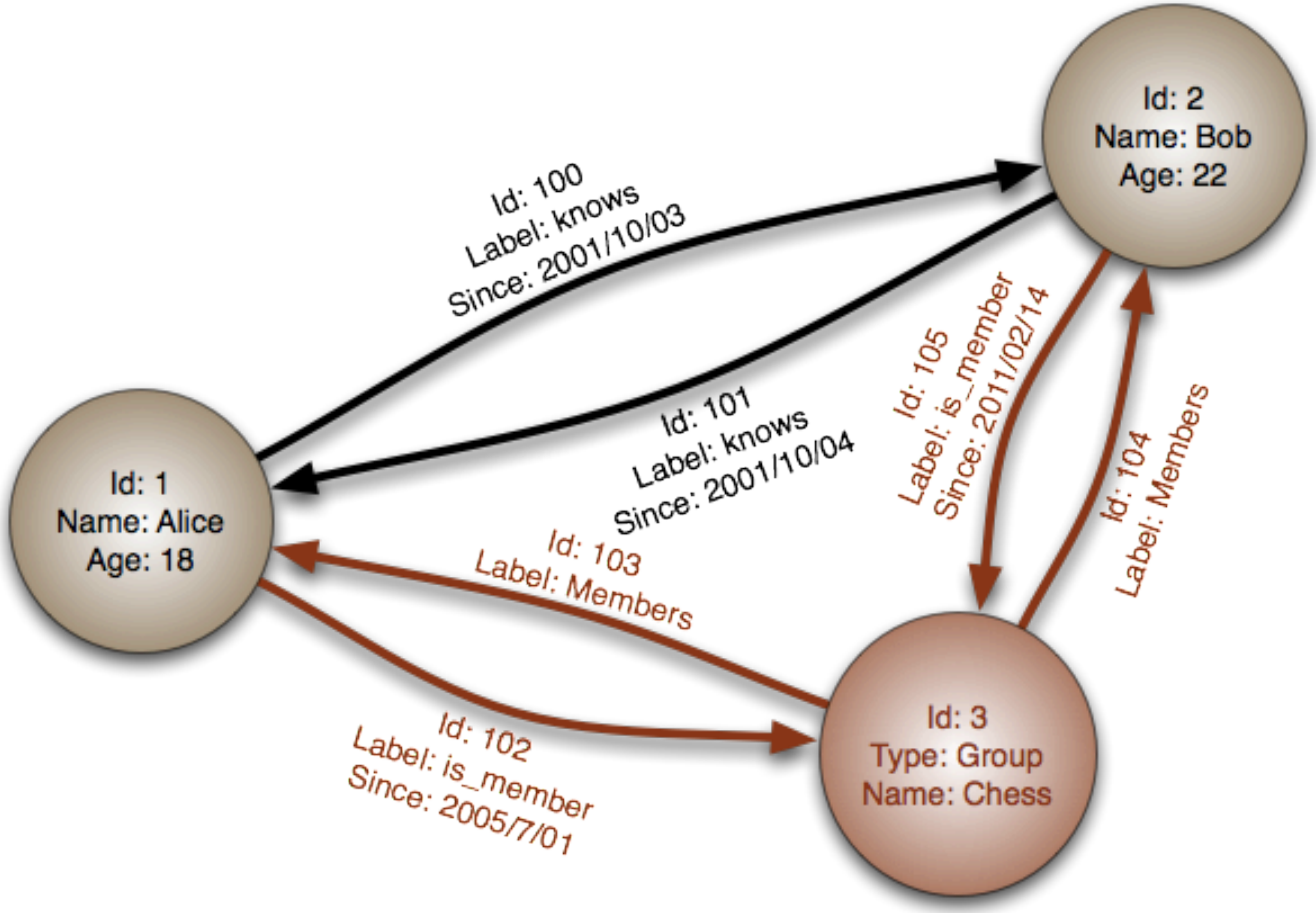
Schema “extracted” from graph



Edge properties

- If required and appropriate, add properties to edges
- Depicted as association classes





[Graph example from https://commons.wikimedia.org/wiki/File:GraphDatabase_PropertyGraph.png]

What we have learned…

Persistence (Part II)

- ✓ Graph Databases
- ✓ Graph Types
  Labeled Property Graphs
- ✓ Graph Traversal
- ✓ Graph Queries
  Introduction to Neo4J
- ✓ Graph Schemas,
  Object - Graph Mapping

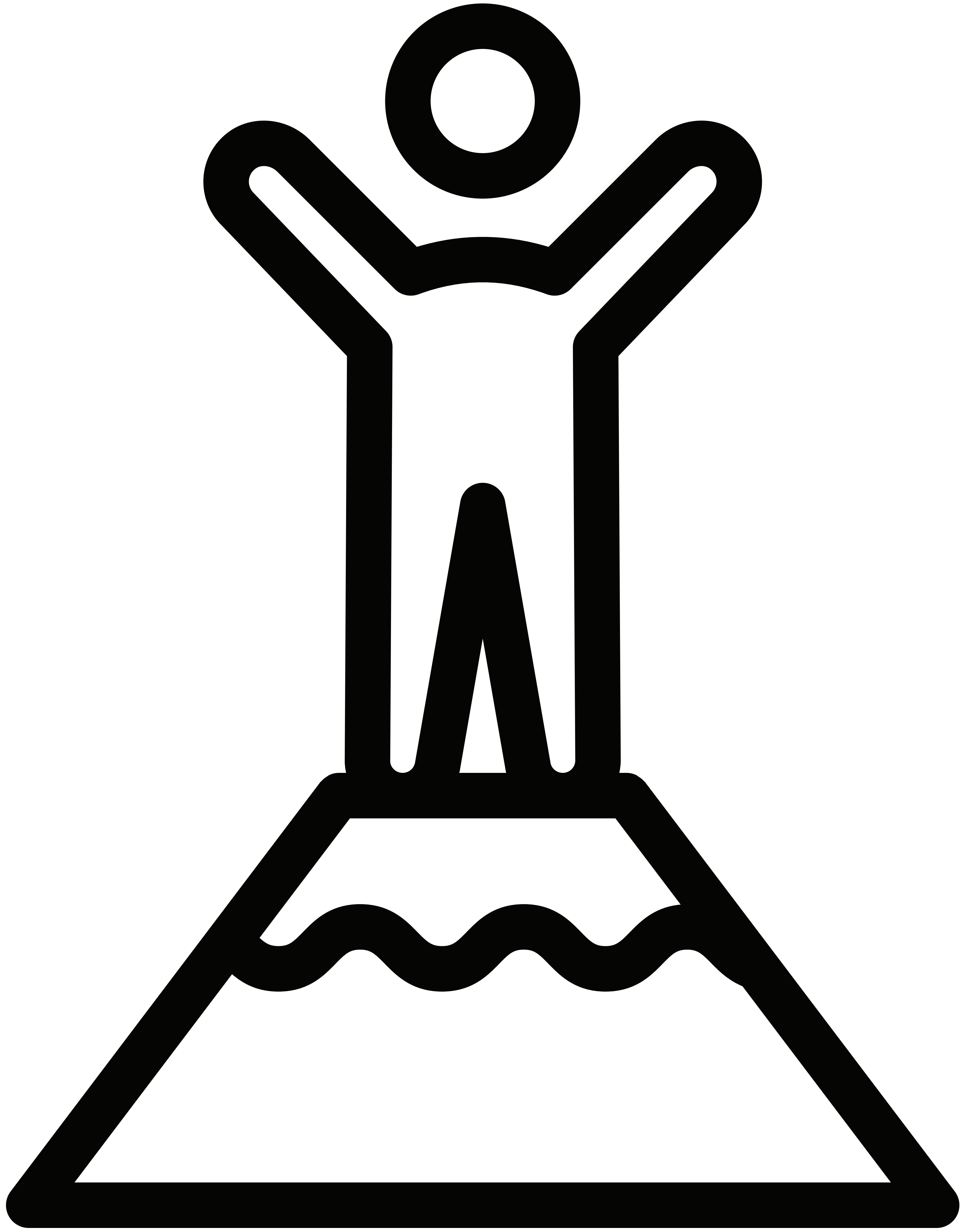


Image: colourbox.de