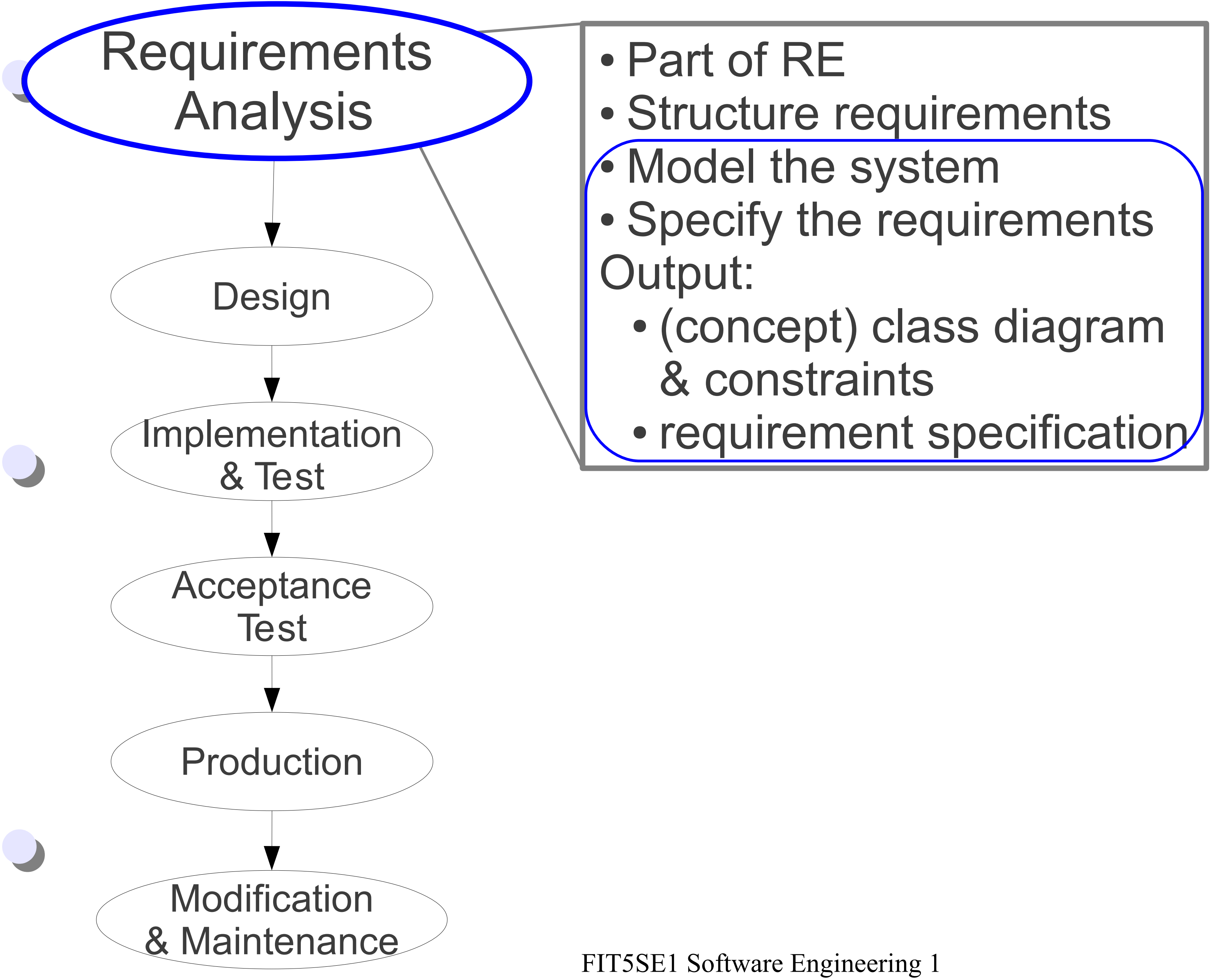
FIT5SE1 Software Engineering 1

Lecture 8:
Requirement modelling and specification

Outline

- ◊ Requirement modelling
  - UML class & use case diagrams
- ◊ Requirement specification
- 🛠 Case study: KEngine

Development process



- Part of RE
- Structure requirements
- Model the system
- Specify the requirements

Output:

- (concept) class diagram & constraints
- requirement specification

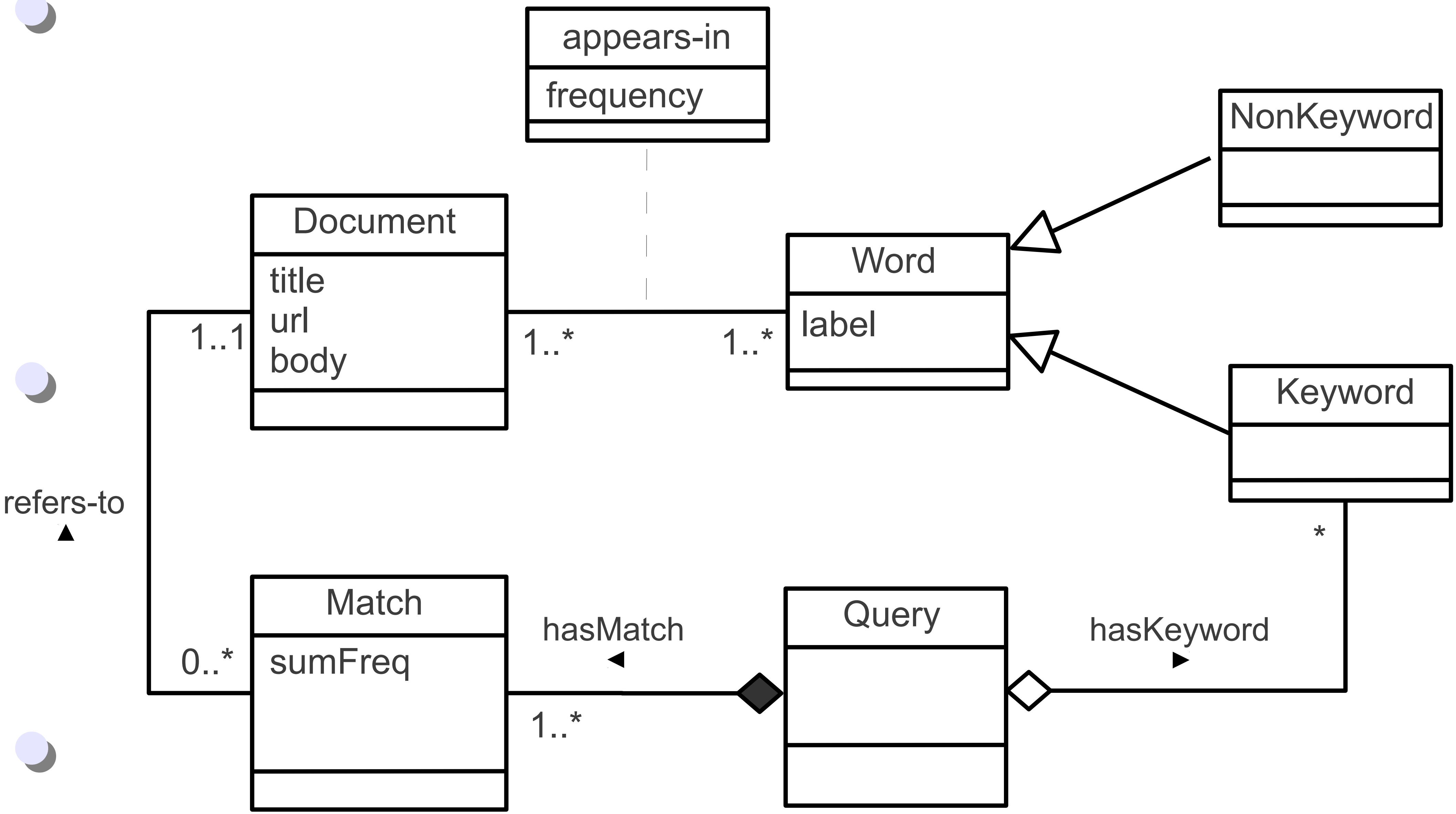
Requirement modelling

- To build conceptual models of the software
- Models exist for functional, data and non-functional requirements
- Models are expressed in a modelling language
- Unified Modelling Language (UML)
  - an object-oriented modelling language
- Selected UML models:
  - for static aspect: class diagram
  - for dynamic aspect: use case diagram

Class diagram

- Models the classes and their associations
- Developed in analysis and refined in design
- Analysis class diagram models the domain entities:
    - e.g. Query, Match, Keyword
- Design class diagram models:
    - entities in fine detail (operations & more attributes)
    - additional software entities

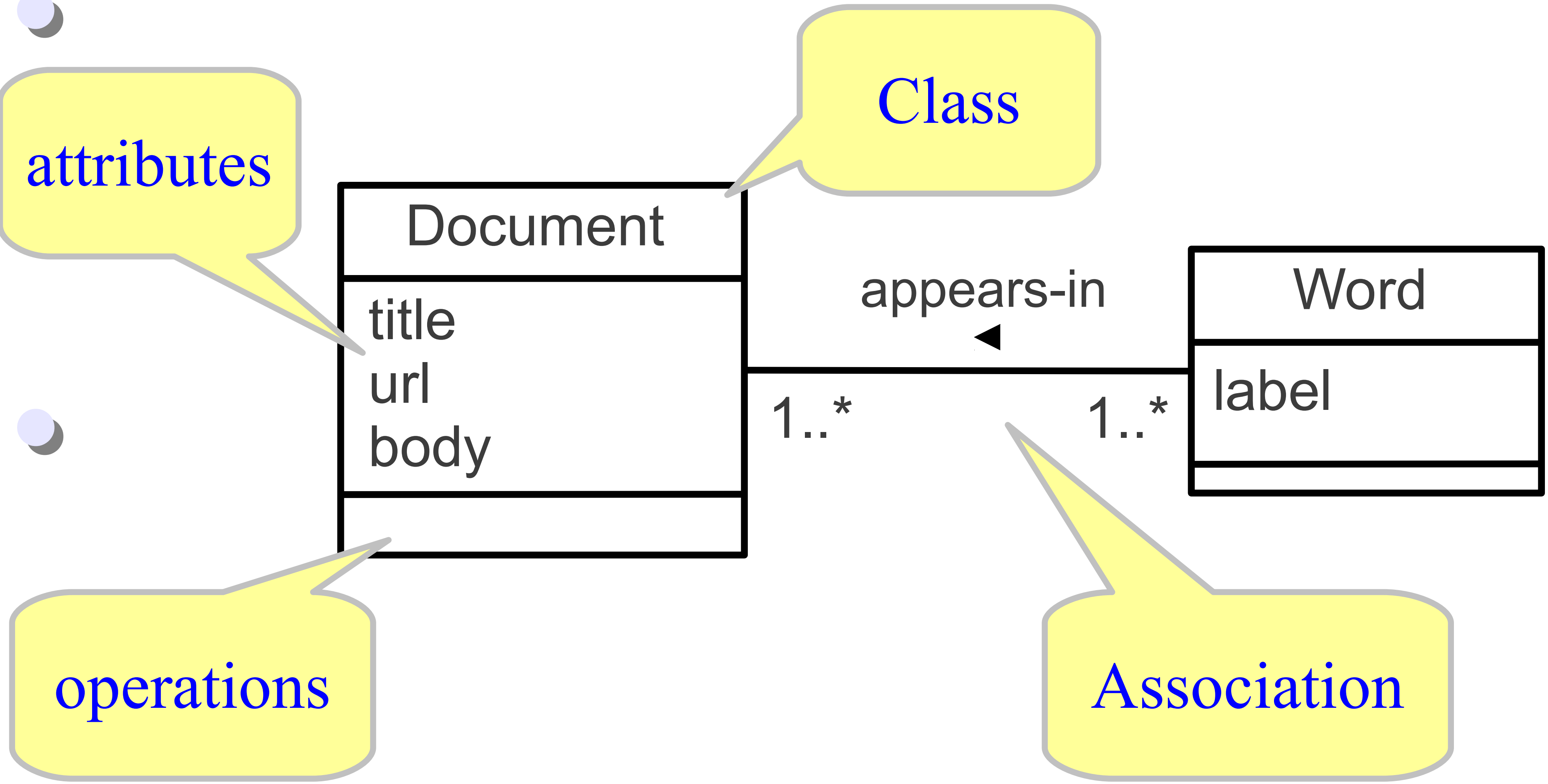
Example: KEngine (details later)



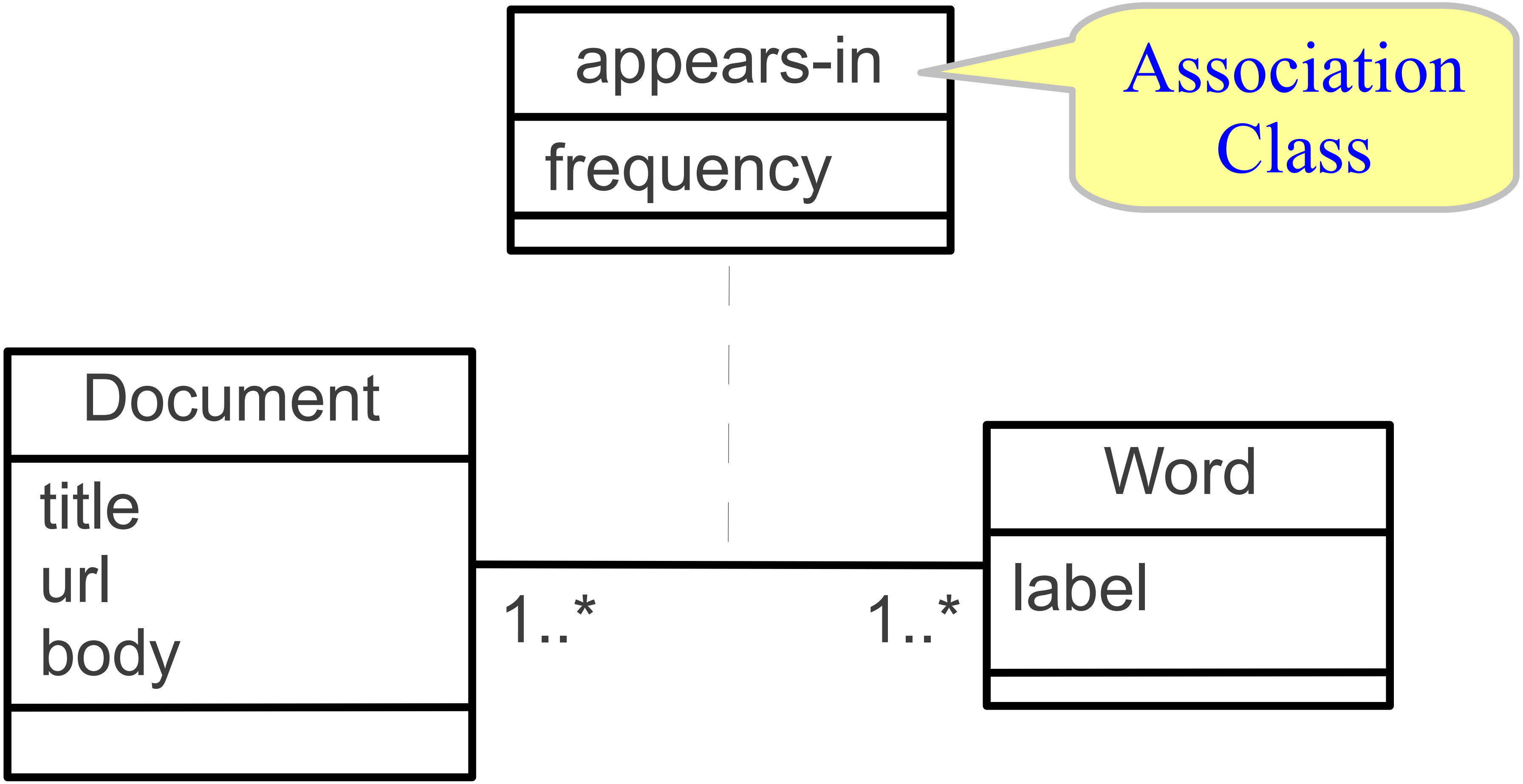
Class diagram elements

UML	ERD equivalences
◊ Class:	Entity
• attributes	
• operations (methods)	
◊ Association	Relationship
• cardinality	
◊ Association class	Associative Entity
◊ Constraint	Domain constraint, …

Graphical UML notation (1)



Graphical UML notation (2)



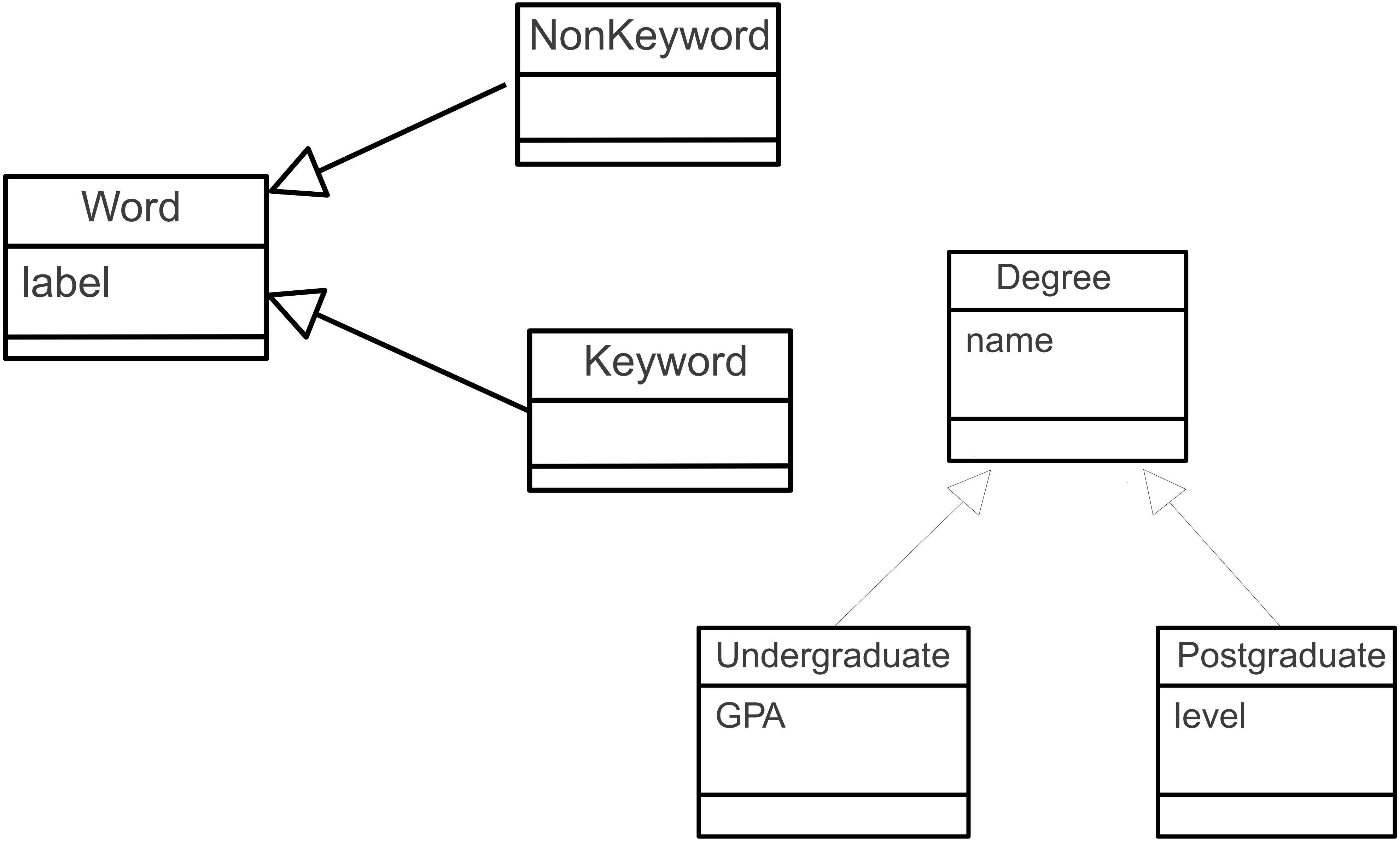
Enhanced associations

- Generalisation
- Aggregation

Generalisation association

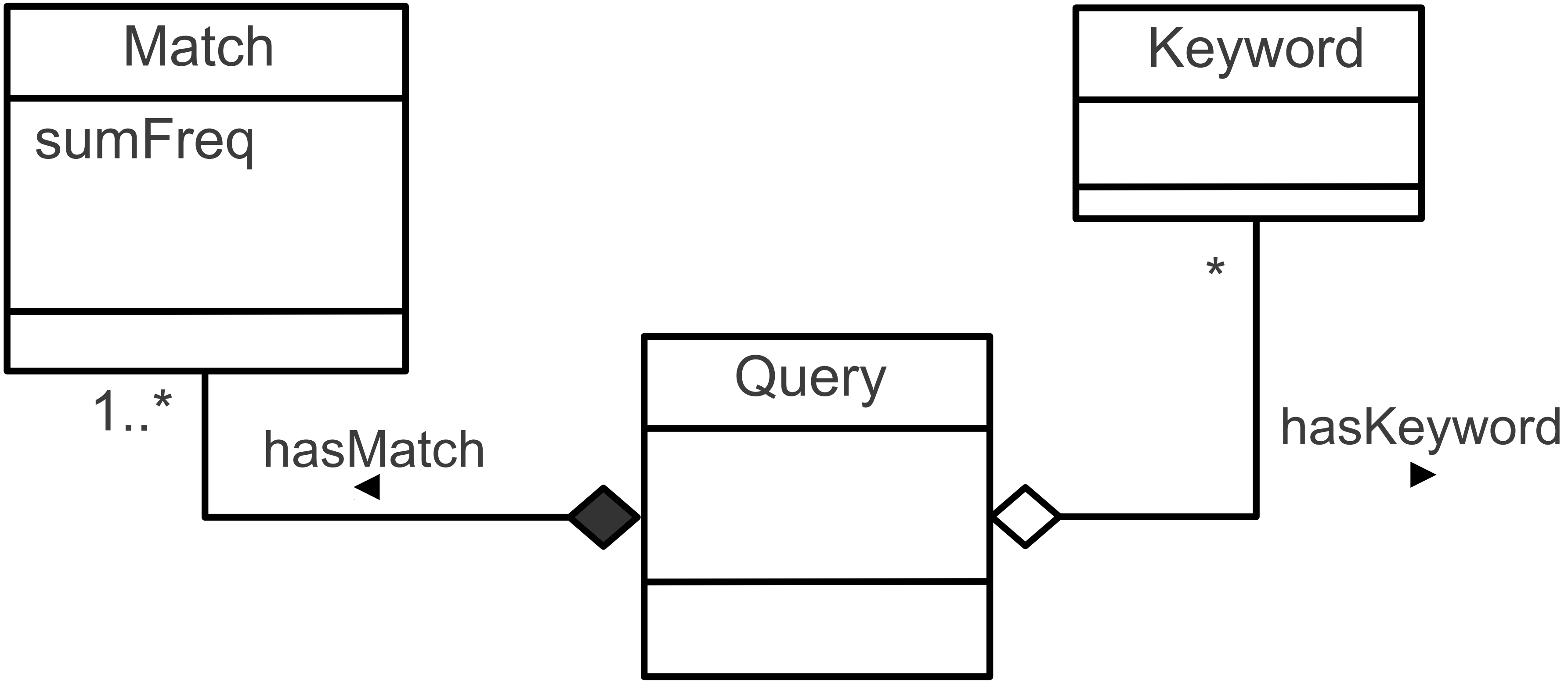
- Model type hierarchy
- Group classes that have common characteristics to form a more general one
- Generalised class is called super class, specialised classes are sub-classes
- Sub-classes inherit properties of super class

Examples


NonKeyword
Word
label
Keyword
Degree
name
Undergraduate
GPA
Postgraduate
level

Aggregation association

- Models a composition relationship



Constraint

- Statement not modelled in the class diagram
- Two types: attribute and association constraint
- **Attribute constraint** specifies:
  - domain constraints,
  - or derived values of an attribute
- **Association constraint** specifies:
  - composition, ordering, etc.

Constraint language

- A formal or informal language (similar to specification's)
- We adopt Liskov’s constraint language but apply to UML model
- Consists of two parts:
  - Natural lang. description (English)
  - A logic statement expressing the constraint over the concerned model elements
- Natural language description is required

Example

Natural lang desc.

- `appears-in:frequency` *is the count of occurrences of a word in a given document*

```
for all d: Document, w: Word [
  appears-in(w,d) =>
  appears-in(w,d):frequency =
    | {k | k in d.body, k=w } |
]
```

Logic statement

How to construct a class diagram

- Map entities to domain classes
- Map relationships to associations
  - cardinality constraints to class cardinalities
- Map associative entities to association classes
- Write constraint statements (if any)

KEngine entities

Document: title, url, body

Word: label

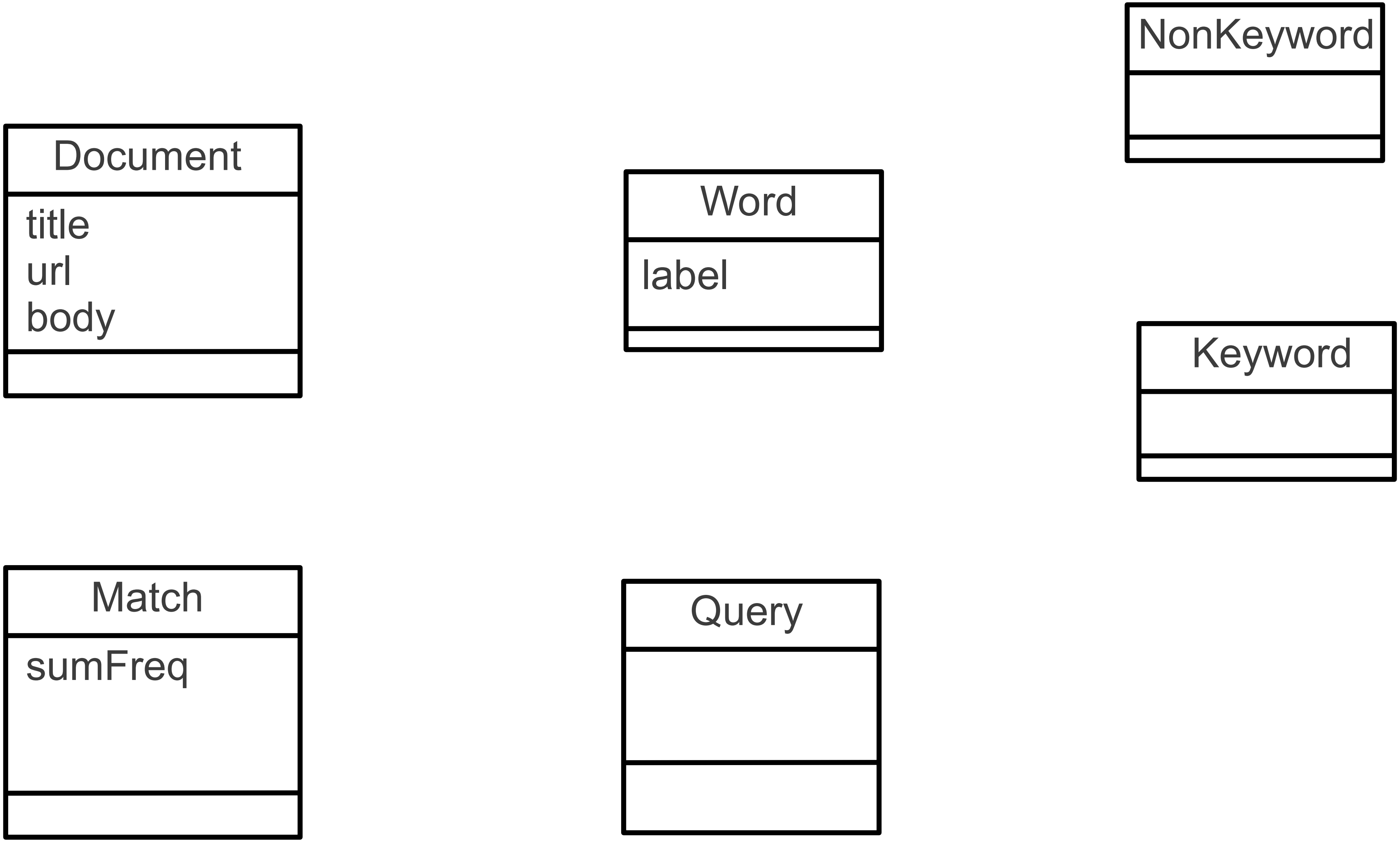
Keyword

NonKeyword

Query

Match: document, sum-freq

Class diagram (a)



NonKeyword

Document

title
url
body

Word

label

Keyword

Match

sumFreq

Query

KEngine relationships

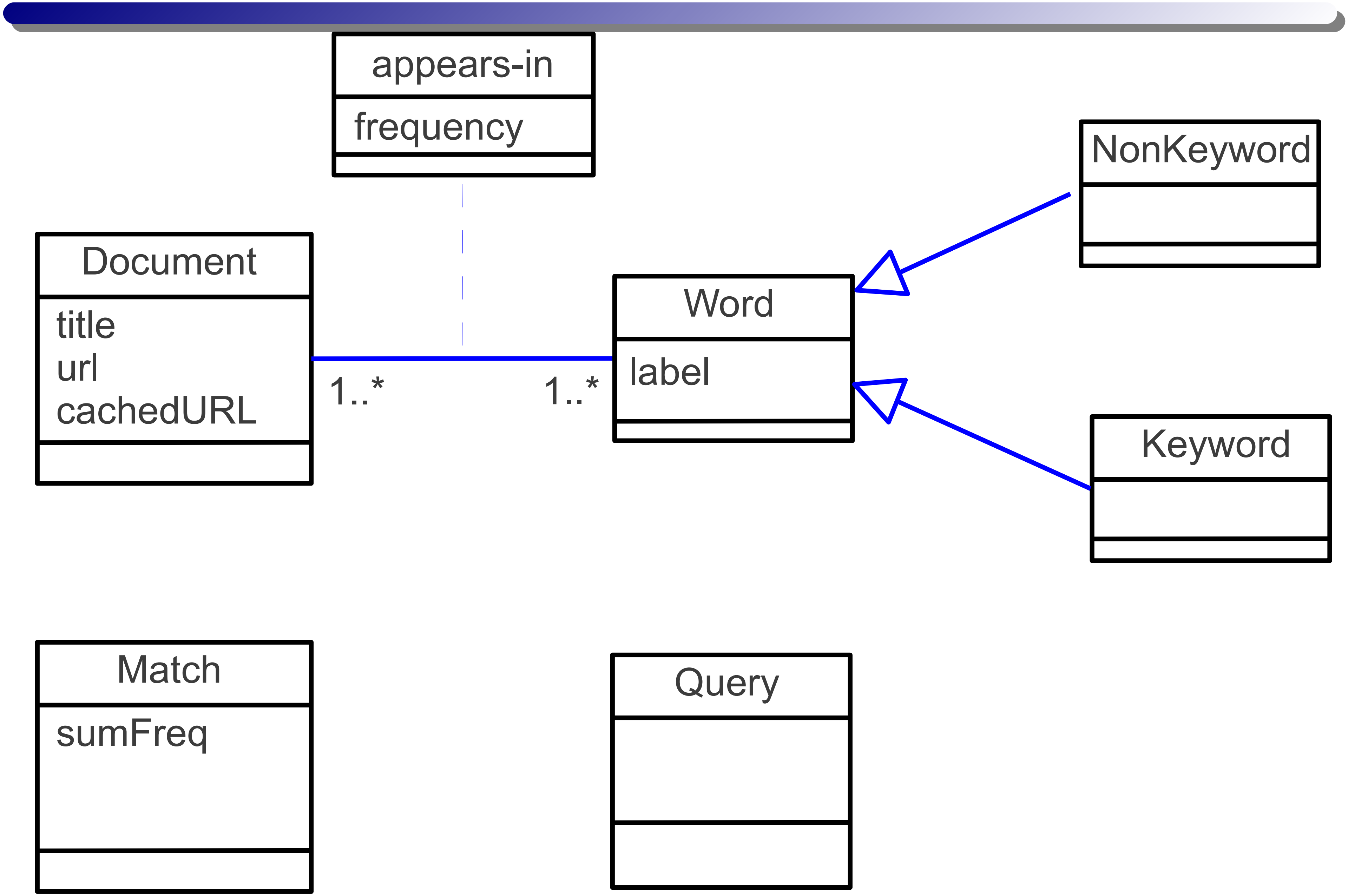
appears-in(Keyword,Document): frequency

hasKeyword(Query,Keyword)

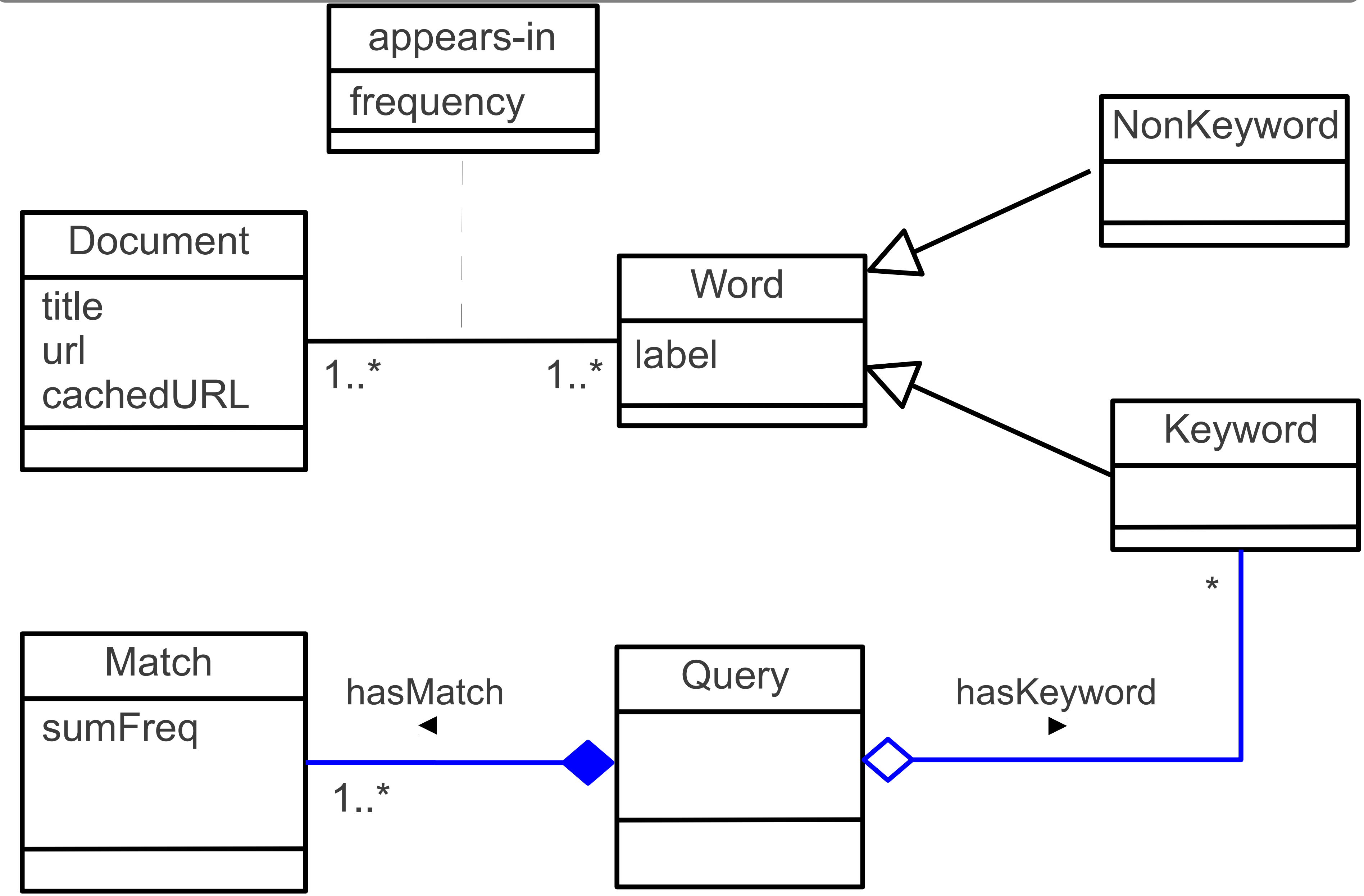
hasMatch(Query, Match)

refers-to(Match, Document)

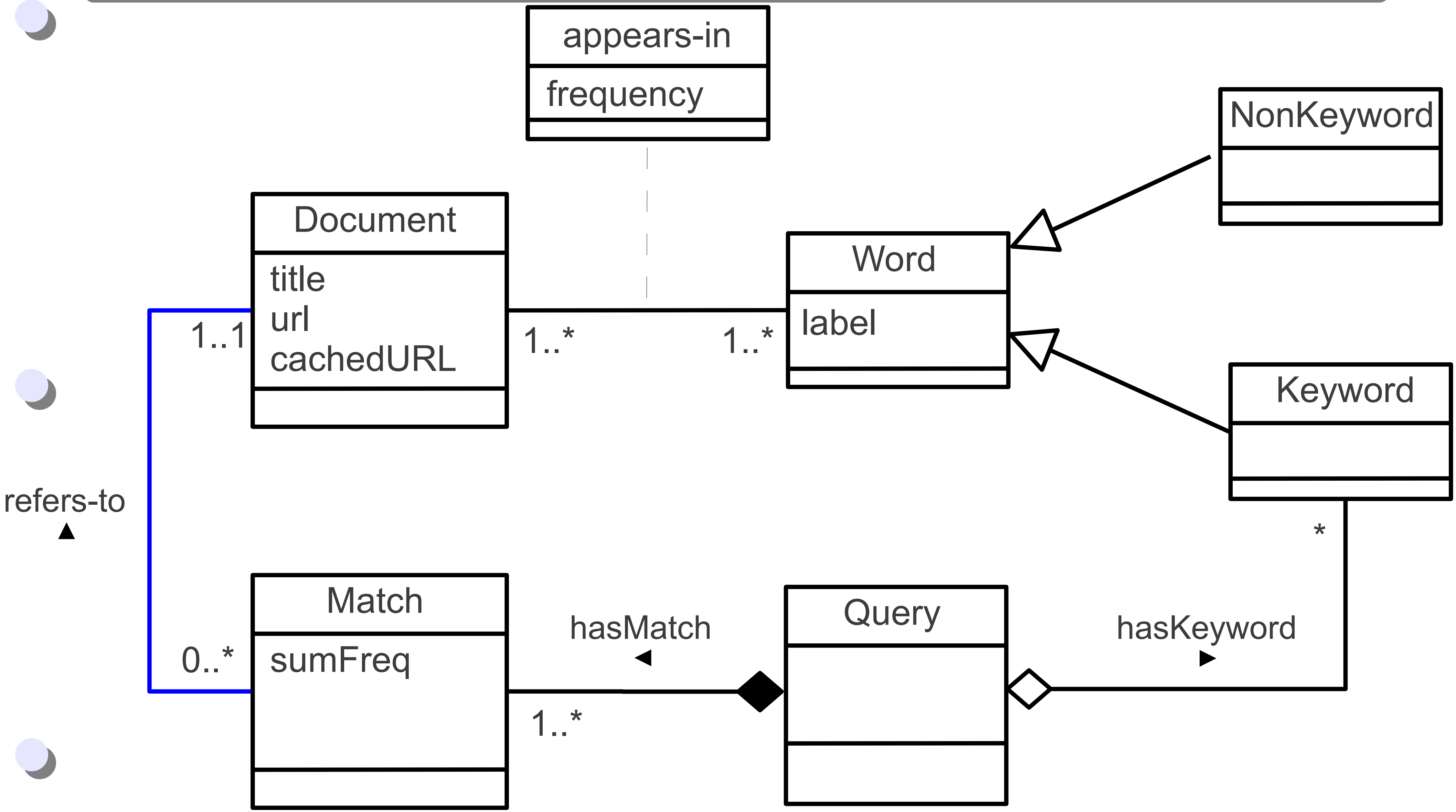
Class diagram (b)



Class diagram (c)



Class diagram (d)



Attribute constraints

- appears-in:**frequency**

  Match:**sumFreq**

appears-in.frequency constraint

- given earlier

Match.sumFreq constraint

- Match:sumFreq *is the total count of occurrences of all keywords in that document*

```
for all q: Query, m: Match, d:
Document [
   hasMatch(q,m) /\ refers-to(m,d) =>
   m.sumFreq =
   sum(appears-in(w,d):frequency),
       for all w in q
]
```

Association constraints

- `Document` matches `Query`

  `Matches`' ordering

-

-

Document matches Query

- *A document matches a query if it contains all the query keywords*

```
for all q: Query, m: Match, d:
Document [
   hasMatch(q,m) /\ refers-to(m,d) =>
   for all w in q (w in d.body)
]
```

Matches ordering

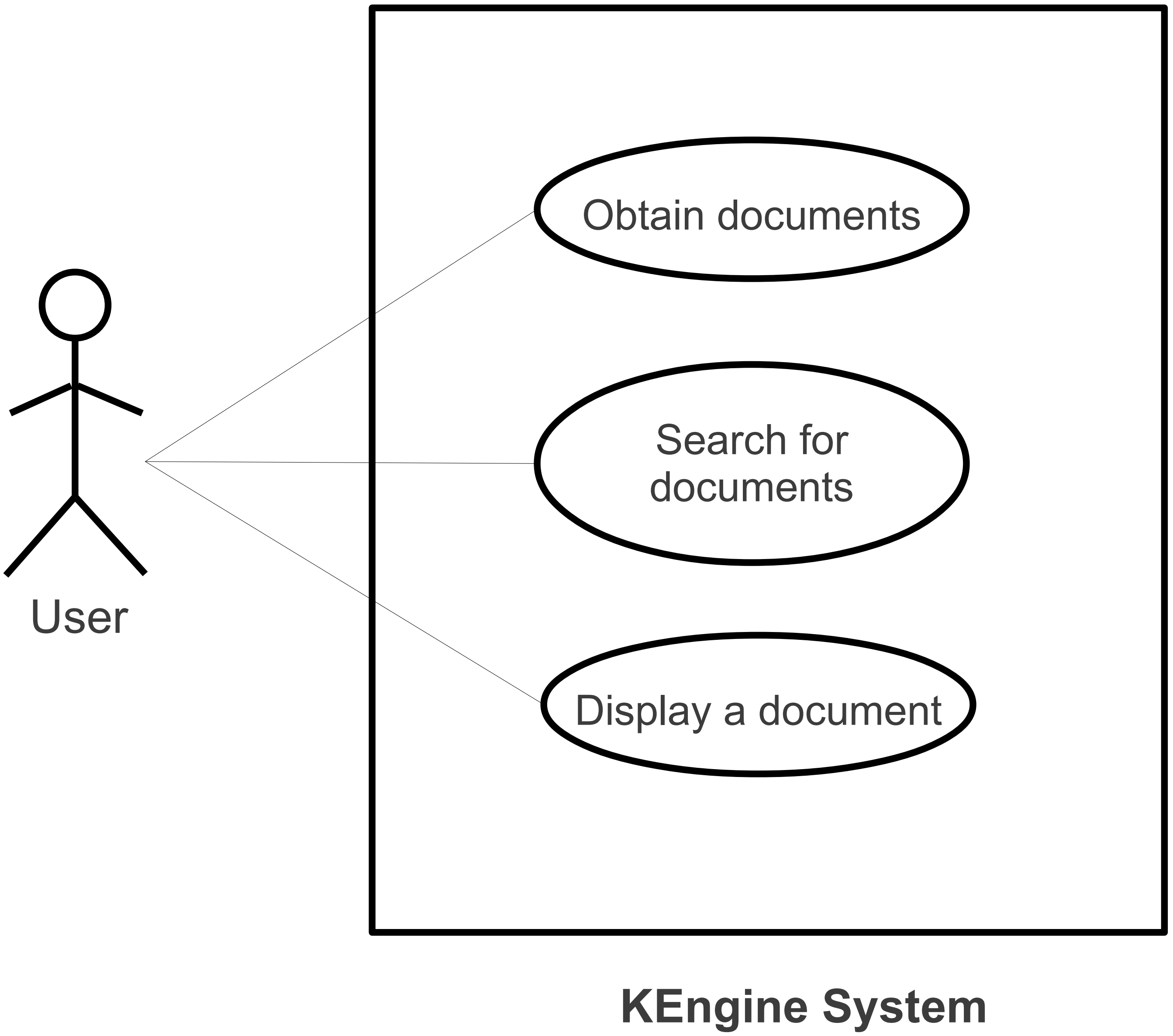
- *Matches are ordered by sum of keyword counts*

```
for all q: Query, m1, m2: Match [
  hasMatch(q,m1) /\ hasMatch(q,m2) /\
  m1.sumFreq ≥ m2.sumFreq =>
    hasMatch(q,m1).index <
    hasMatch(q,m2).index
]
```

Use case diagram

- Shows actor interactions via use cases
- Many-to-many interactions:
  - an actor may interact with many use cases
  - a use case may involve more than one actors
- System is a high-level abstraction
  - only functionality description, no further detail

Graphical notation



Requirement specification

- A high-level specification of the system:
  - system as a high-level abstraction
- Combines both data and function models
- Specifies succintly *what* the system provides
- Used as input in design to generate the design specification

Requirement specification language

- A simplified form of the (design) specification language
- Replace REQUIRES clause by CHECKS
- CHECKS clause:
  - lists the input and model constraints
- No MODIFIES clause
  - operations always modifies the system state
- Refers to the model elements

System specification

- Considers the system as an abstraction
- Use cases become system operations

Example: Engine

- `startEngine`
- `addDocuments`
- `query`
- `queryMore`
- `findDoc`

Obtain documents

Search for documents

Display a document

Engine specification

```
/**
  @overview
   Represents keyword search engines. An engine holds a mutable
   collection of documents, which are obtained from some given URLs.
   The engine is able to pocess a keyword query to search for
   documents that contain the keywords.

   The matching documents are ranked based on the frequencies of the
   keywords found in them.

   The engine has a private file that contains the list of
   uninteresting words.
 */
class KEngine {

}
```

Procedural specification

- No return types or exceptions
- Total
- Preserve model constraints

startEngine

```
/**
  @overview ...(omitted)...
 */
class Engine {
   /**
     @effects
      Starts the engine running with NonKeyword
        containing the words in the private file.
      All other sets are empty.
    */
   static startEngine()
```

addDocuments

```
/**
  @checks u does not name a site in URL and
    u names a site that provides documents

  @effects
   Adds u to URL and
   adds documents at site u with new titles to Document.
   If Keyword is non-empty adds any documents that match
      the keywords to Match.
 */
addDocuments(String u)
```

query

```
/**
  @checks: w is not in NonKeyword

  @effects
   Sets Keyword = {w} and
   makes Match contain the documents that match w,
     ordered as required.
 */
query(String w)
```

queryMore

```
/**
  @checks Keyword != {} and
    w not in NonKeyword and w not in Keyword

  @effects
   Adds w to Keyword and
   makes Match be the documents already
     in Match that additionally match w.
   Orders Match properly.
 */
queryMore(String w)
```

findDoc

```
  /**
    @checks t is in titles

    @effects
     return d in Document s.t. d's title = t
   */
  findDoc(String t)
} // end Engine
```

Summary

- A model is expressed in a modelling language
- UML is an object-oriented modelling language that supports requirement modelling
- Data and functional modelling are helped by UML class and use case diagrams
- Requirement specification is written in a simplifed version of the specification language, using the models

Questions?