FIT5SE1 Software Engineering 1

Lectures 9-10:
Object oriented software design

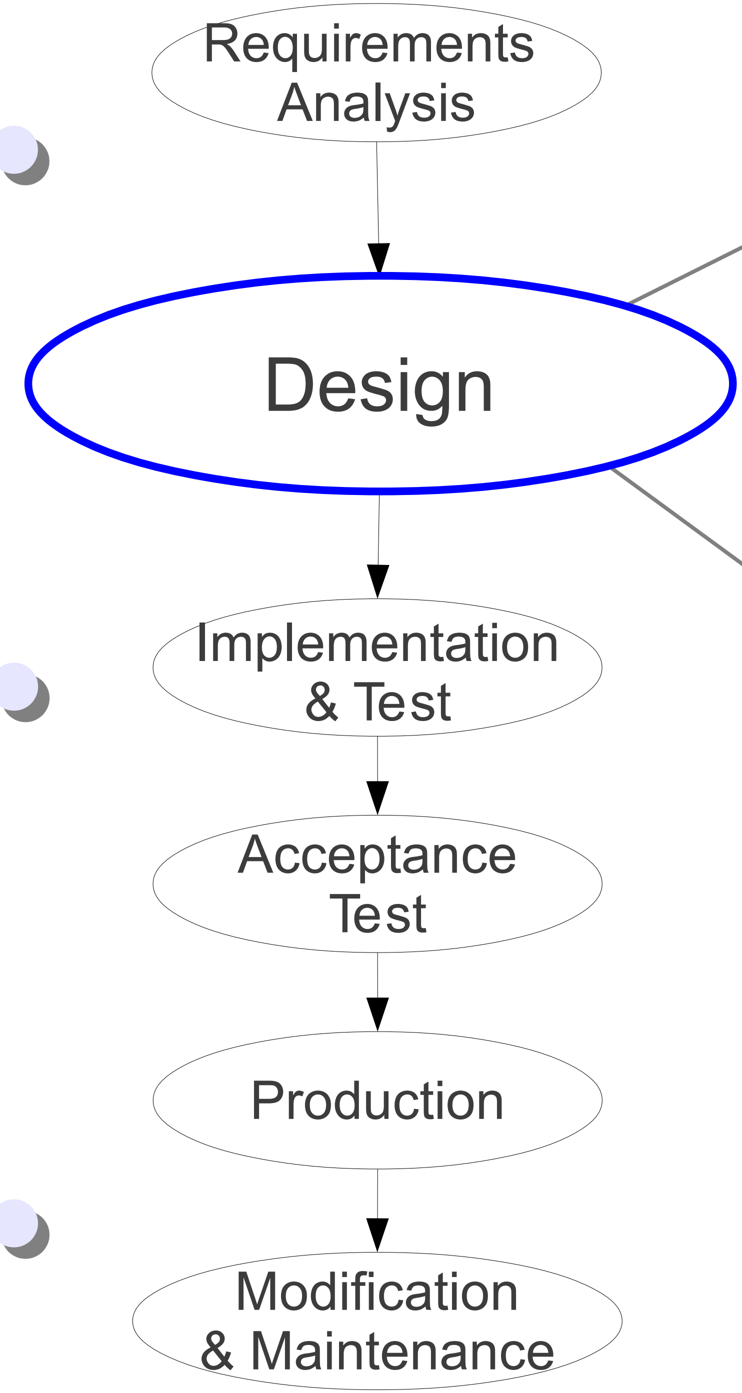
Outline

- Design overview
- Design process
- Design notebook
- Case study: KEngine design
  - Iteration 0
  - Iteration 1
  - Iteration 2
- Design process review

Lecture 9

Lecture 10

Development process



- Decomposition by abstraction
- Iterative
- Output: design notebook:
  - design class diagram
  - sequence diagram
  - design specification

Design overview

- *Input*: requirement specification
- *Output*: a modular program structure
  - components are all good abstractions
  - easy to implement and modify
- *Goal*: to develop detailed specifications

Design process

- Two principles:
  - decomposition by abstraction
  - iterative refinement
- Decomposition by abstraction:
  - decompose functions
  - invent or use abstractions to accommodate the sub-functions
- Iterative refinement (top-down):
  - divide design activities into iterations
  - start high-level, incrementally refine

Iterative steps

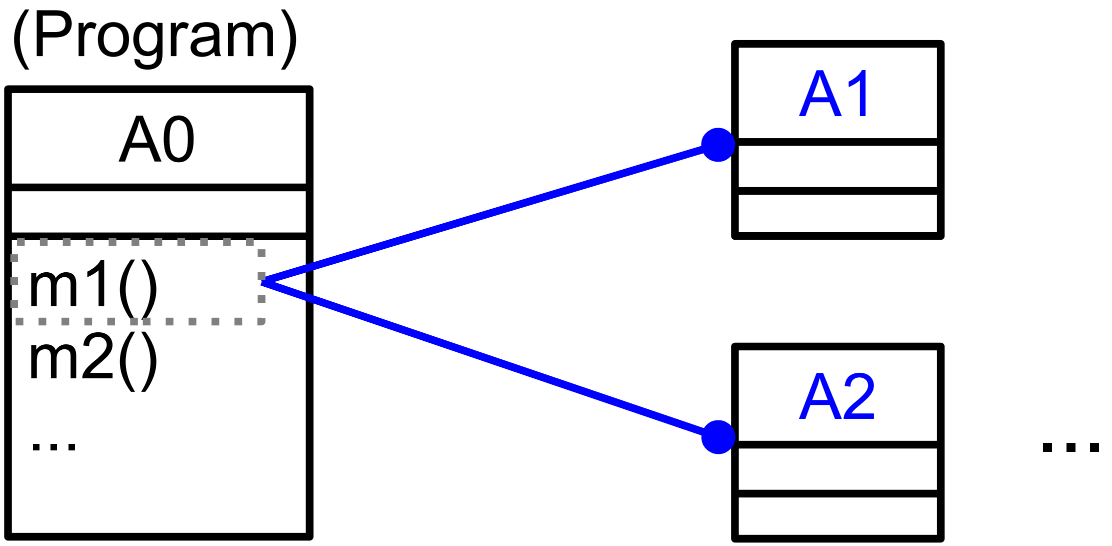
- ◊ In each iteration:
  - select an abstraction (A)
  - identify helper abstractions needed to:
    - implement A and
    - facilitate decomposition
  - write/update design specification for A
  - stop if design specifications of all abstractions have been determined

Design process overview

- **Iteration 0: initial abstractions**
  - identify some initial abstractions, including the software and other *obvious* concepts
  - these concepts can be identified from initial design spec. of the software's operations
- **Iteration 1: top-level abstractions**
  - Start analysing the design spec. of each initial abstraction to identify new abstractions (if any)
- **Subsequent abstractions:**
  - Repeat the analysis for each new abstraction until no further abstractions are identified

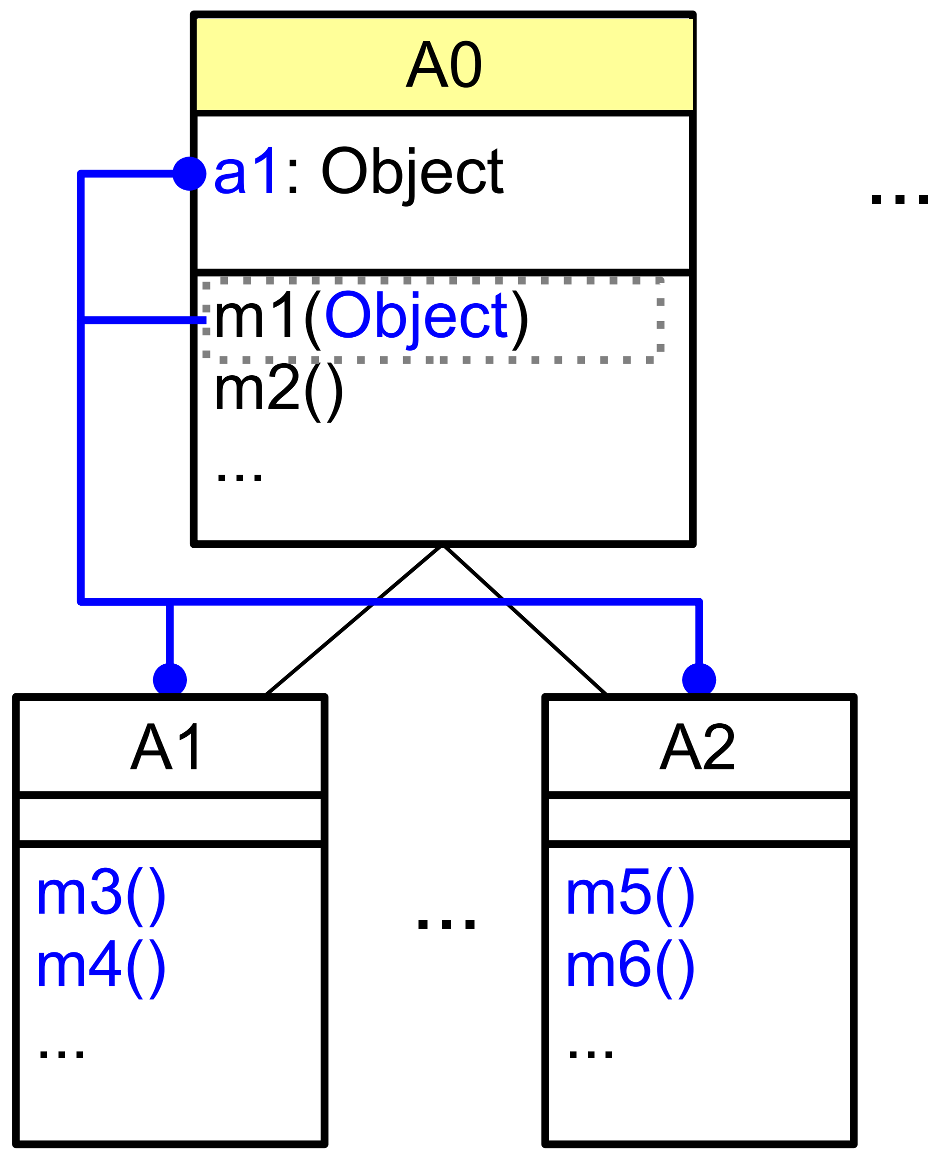
Illustration: Initial abstractions

Iteration 0



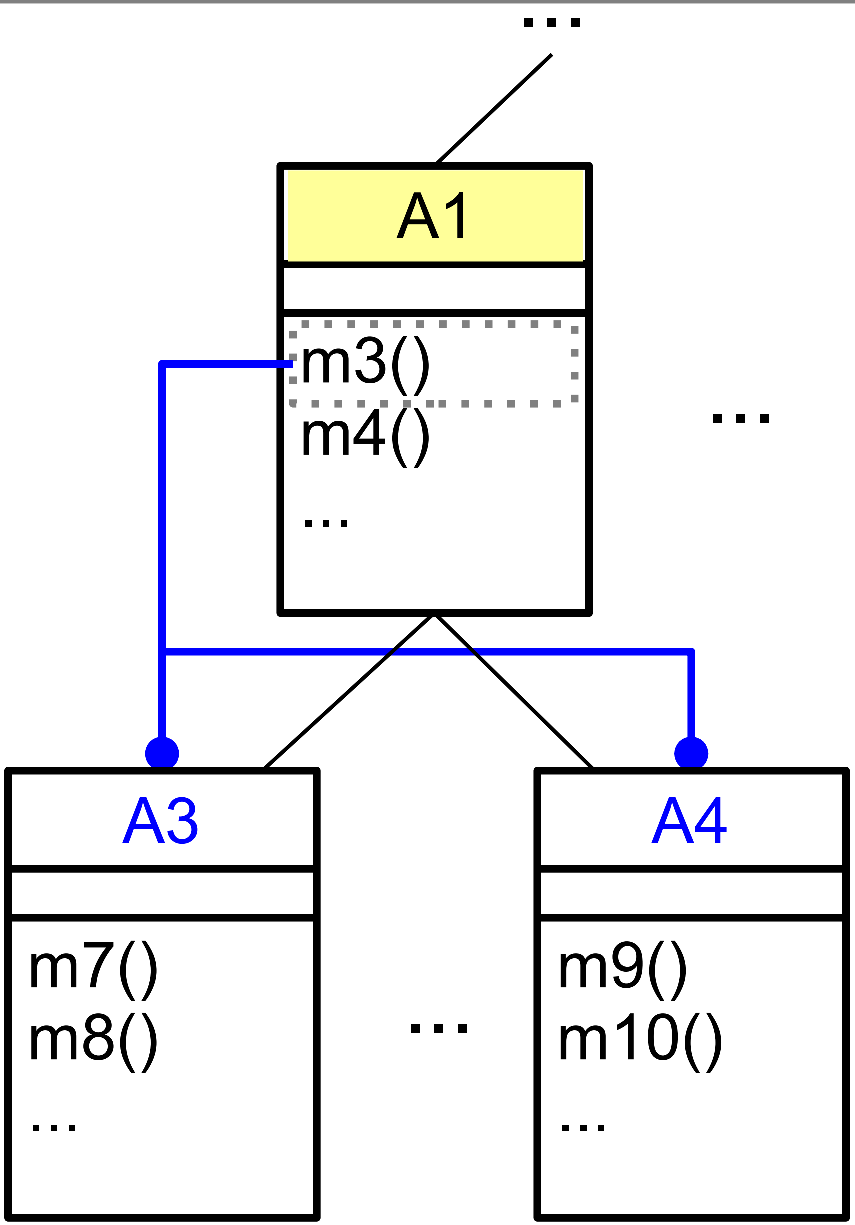
Top-level abstractions

Iteration 1



Subsequent abstractions

Iteration 2,...



Design notebook

- Documents all the design decisions
- A section for each abstraction, containing:
  - design specification
  - NRFs (eg. performance, modifiability)
  - implementation sketch (if needed)
  - other information: alternatives, context of use
- Includes diagrams:
  - design
  - sequence

Design class diagram

- More refined compared to the concept class diagram:
  - all are software classes
  - some new software specific classes
  - domain classes are completed with rep and operations
  - replace certain domain classes by software ones
    - e.g. Word, Keyword, NonKeyWord → String
- Expressed in UML
  - more detailed than module dependency diagram

Relationship with concept class diagram

- Two methods of building design class diagram:
  - use concept class diagram (if available)
  - is created from scratch (without using concept class diagram)

Notebook update format

- Decompose queryFirst:
  - …
  - …
  - …
  - **For each document, determine if it is a match**
  - …
  - …
  - …

<<design note>>

design update

DESIGN ITERATION 0

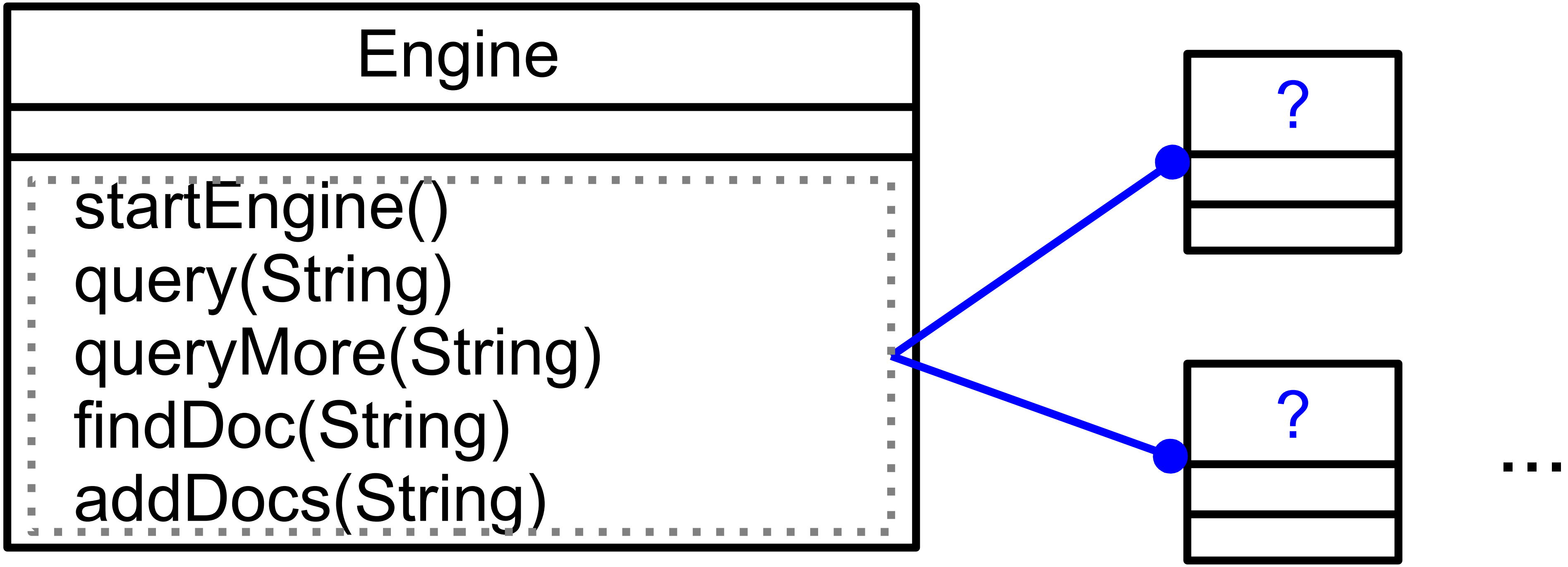
Initial abstraction(s)

Preparation

- Transform requirement specification into initial design specification:
  - types in CHECKS & EFFECTS become initial abstractions
- Write the design spec for each initial abstraction:
  - make the operations total by removing each CHECK clause by a suitable Exception
  - use initial abstractions as return types where required
- Construct initial design class diagram:
  - associations with *dependency* indicators

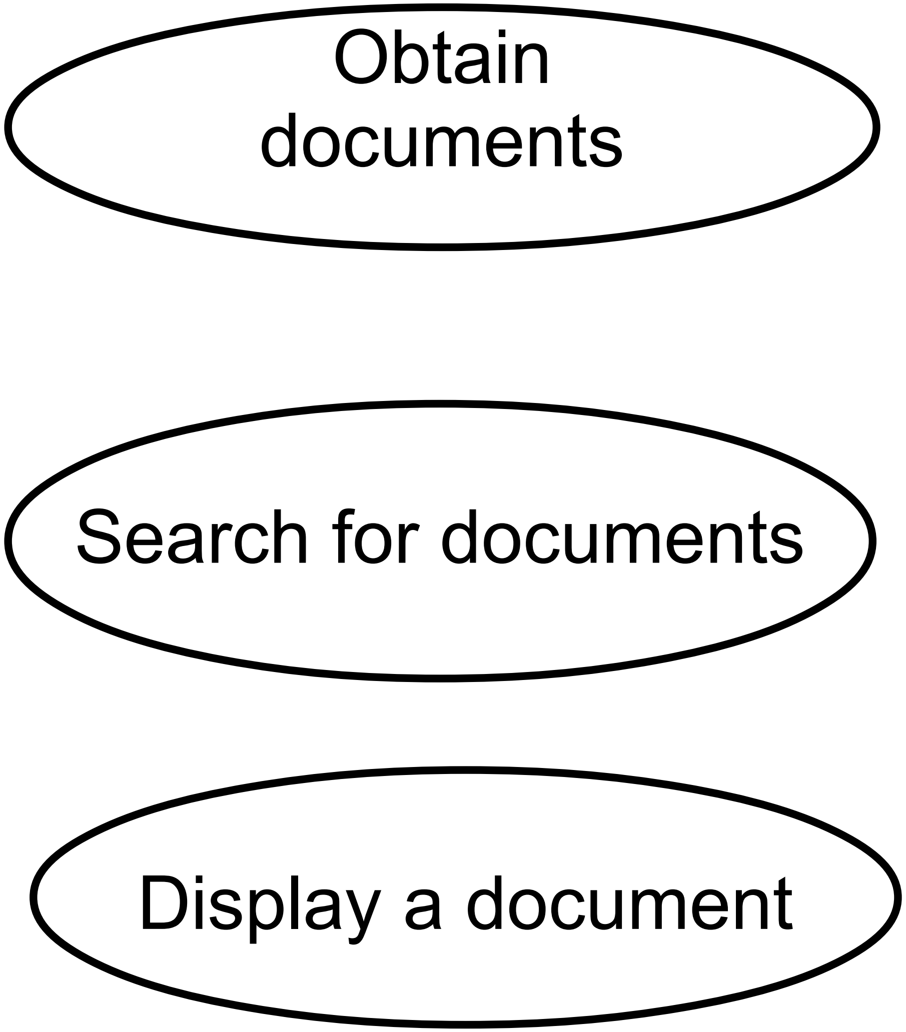
KEngine: initial design overview

- Which abstractions can we ***initially*** identify from `Engine`'s requirement spec?



Requirement specification

- startEngine
- addDocuments
- query
- queryMore
- findDoc



Obtain documents

Search for documents

Display a document

Example: initial abstractions of KEngine

```
/**
  @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 Engine {

}
```

need an abstraction to represent the engine
→ creates abstraction Engine

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)
```

- need an abstraction to represent Document
  → creates abstraction Doc
- also need for Keyword and Match (later)

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)
```

- need an abstraction to hold a keyword and to store matches

- may use String for Keyword & NonKeyword

queryMore

```
/**
  @checks Key != {} 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)
```

- need an abstraction to hold *keywords* and to store matches
  → creates abstraction `Query`
- may use `String` for Keyword & NonKeyword

findDoc

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

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

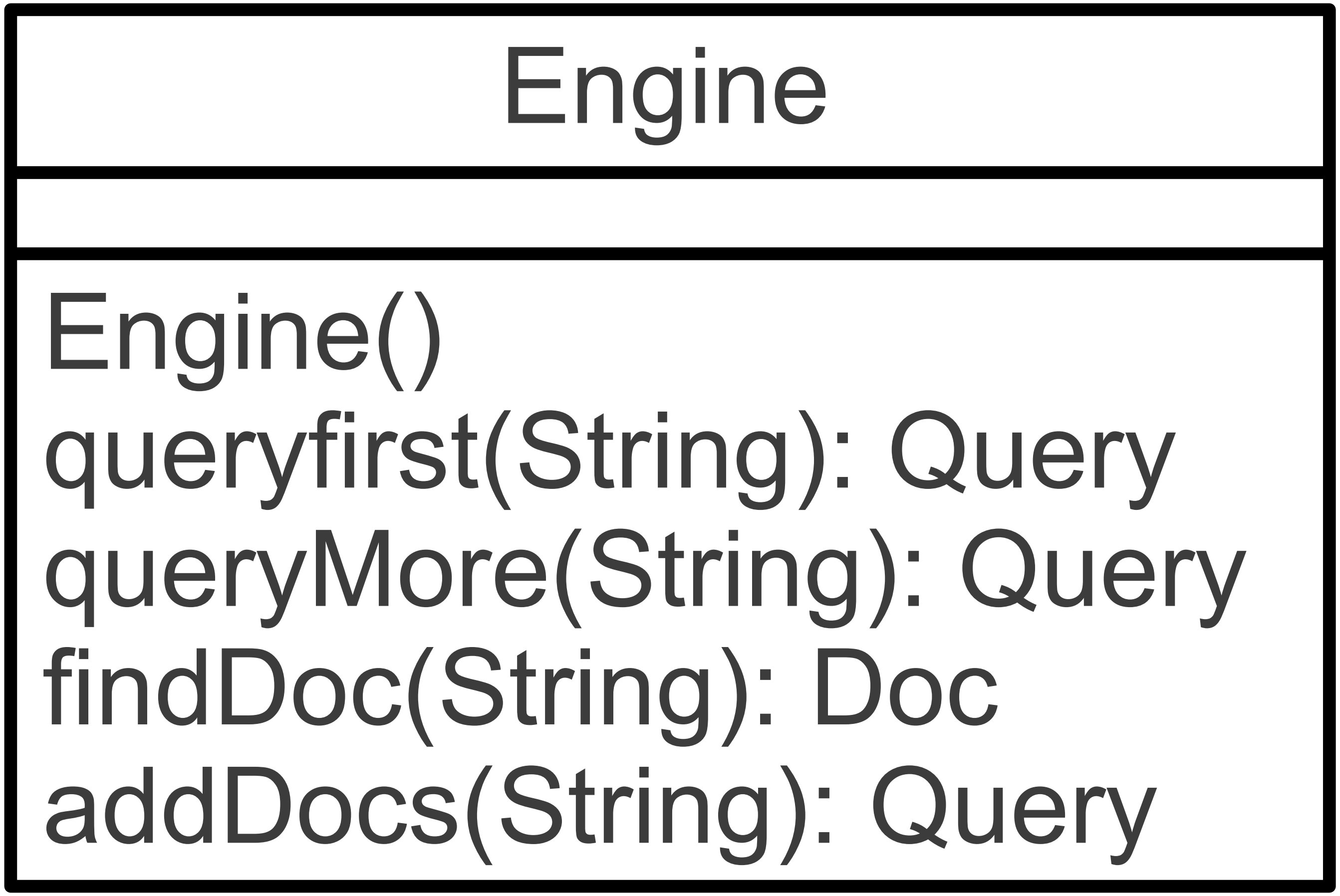
needs an abstraction to represent Document

→ *uses* abstraction Doc

Initial data abstractions

- `Engine`
- `Doc`
- `Query`

Engine



Engine

Engine()
queryfirst(String): Query
queryMore(String): Query
findDoc(String): Doc
addDocs(String): Query

Initial design spec (1)

```
/**
 * @overview ...(omitted)...
 */
class Engine {
  /**
   * @effects
   *  If uninteresting words not retrievable
   *     throws NotPossibleException
   *  else
   *     creates NonKeyword and initialises app. state
   *     appropriately
   */
  Engine() throws NotPossibleException
```

Initial design spec (2)

```
/**
 * @effects
 *  If WORD(w) = false or w in NonKeyword
 *    throws NotPossibleException
 *  else
 *    sets Keyword = {w}, performs the new query, and returns the result
 */
 Query queryFirst(String w) throws NotPossibleException
```

```
/**
 * @effects
 *  If WORD(w) = false or w in NonKeyword or Key = {} or w in Keyword
 *    throws NotPossibleException
 *  else
 *    add w to Keyword and returns the query result
 */
 Query queryMore(String w) throws NotPossibleException
```

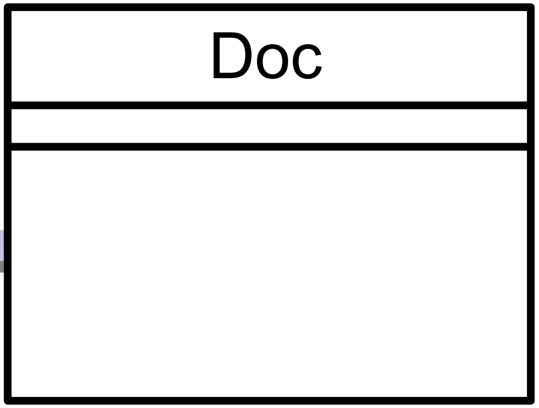
Initial design spec (3)

```
  /**
   * @effects
   *  If t not in Title throws NotPossibleException
   *  else returns the document with title t
   */
  Doc findDoc (String t) throws NotPossibleException

  /**
   * @effects
   *  If u is not a URL for a site containing documents or u in URL
   *    throws NotPossibleException
   *  else adds the new documents to Doc.
   *    If no query was in progress
   *      returns the empty query result
   *    else
   *      returns query result that includes any new matching documents
   */
  Query addDocs(String u) throws NotPossibleException

} // end Engine
```

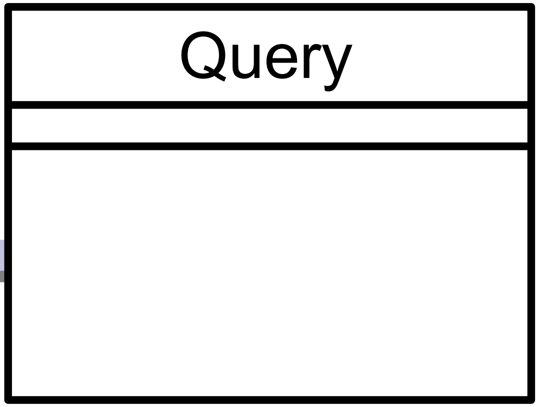
Doc



```
/**
 * @overview
 *  A textual document contains a title and some text content.
 */
class Doc {

} // end Doc
```

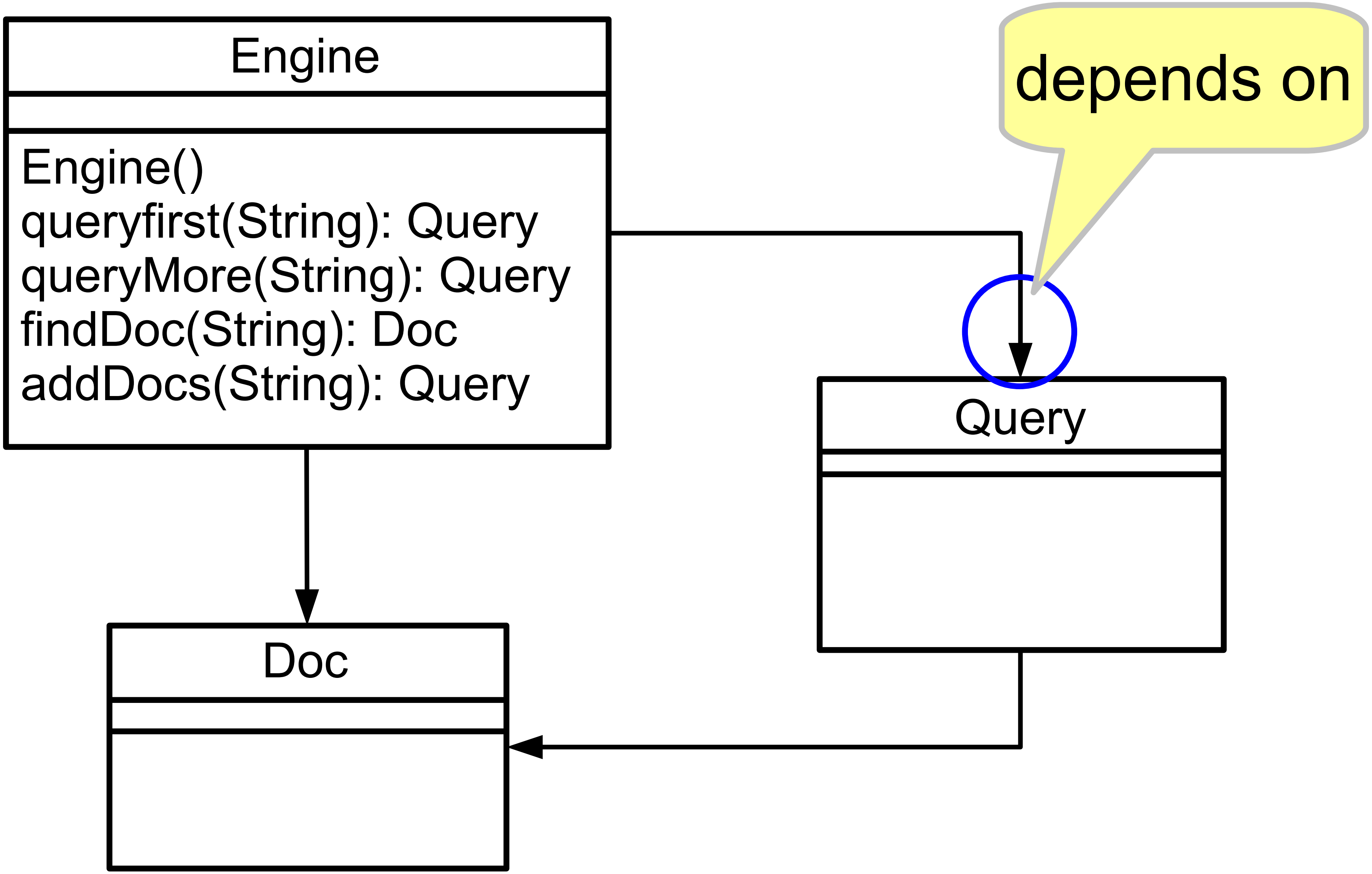
Query



```
/**
 * @overview
 *   A query consists of keywords that are of interest.
 */
class Query {

}
```

Initial design class diagram



DESIGN ITERATION 1

Top-level abstractions

Top-level data abstractions (1)

- Find *all* the top-level abstractions
- Start decomposition with Engine:
  - decompose each function
  - analyse the sub-tasks (most significant ones first) to identify other data abstractions
  - identify operations of each data abstraction

Top-level data abstractions (2)

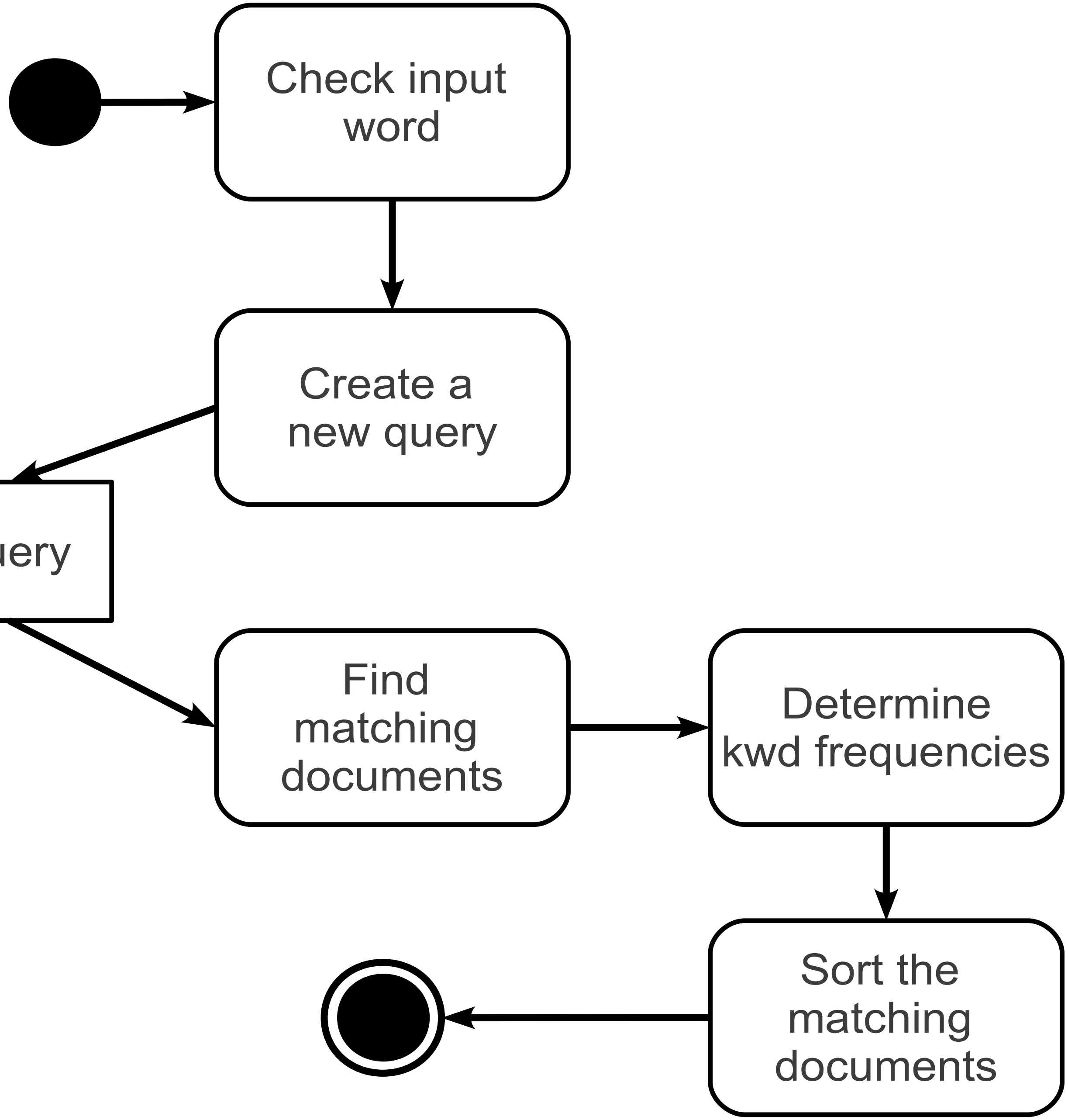
- Validate using *sequence diagram*
- Update the design class diagram
- Write/update the representation (rep) of each data abstraction
- Write/update the specification of each abstraction:
  - data &
  - procedural

KEngine: top-level design overview

- Which **abstractions** can we **identify/refine** from the design spec. of previous iteration?

Engine.queryFirst

Activity diagram



D by A (1.1)

- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

D by A (1.2)

- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

<<design note>>

• same document is scanned many times (for different queries)
• need a fast look up method to find w in doc
→ *record the words of each document when it is processed*

D by A (1.3)

- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

- freqs are likely to be re-used many times (for different queries)
  → *record freqs of words when scanning documents*

D by A (1.4)

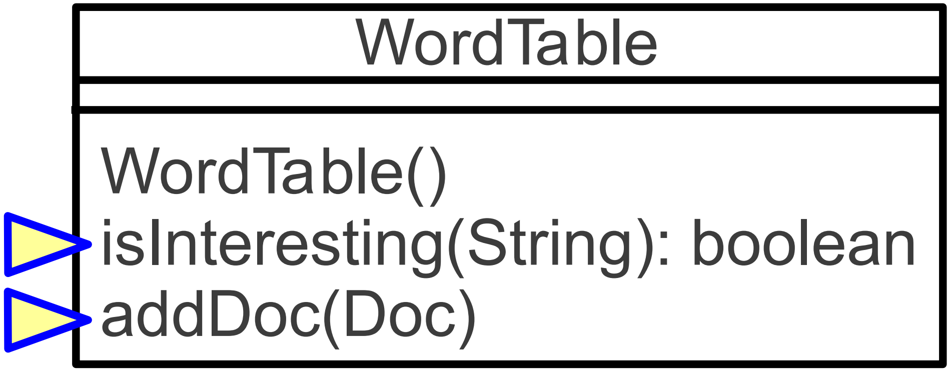
- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

- needs to know the uninteresting words
- needs to maintain both interesting and uninteresting words easily

→ *record both types of words in the same abstraction (WordTable)*

WordTable

- Iteration abstraction
- Stores words
- Has operations to check and maintain the words set



WordTable

WordTable()
isInteresting(String): boolean
addDoc(Doc)

D by A (1.5)

- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

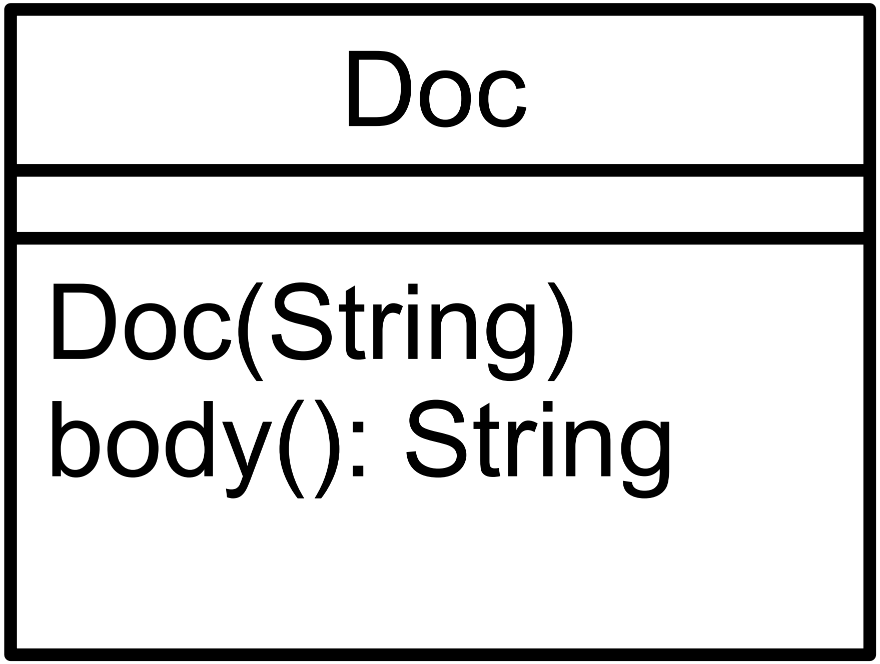
- needs to record first keyword in Query

D by A (1.6)

- Check that the input string w is a word
- Check that w is an interesting word
- Start a new query with w as the keyword
- For each document, determine if it is a match
- For each document, determine the freq of w
- Sort the matches by freq of w
- Return the query and matches

- need to know Doc’s body → *create body() in Doc*
- need to record the query matches and their freqs: → *records matches and their freqs in Query*
- need a simple way of retrieving each match: → *create methods size() and fetch(int) in Query*

Doc



Doc

Doc(String)
body(): String

Engine.queryMore

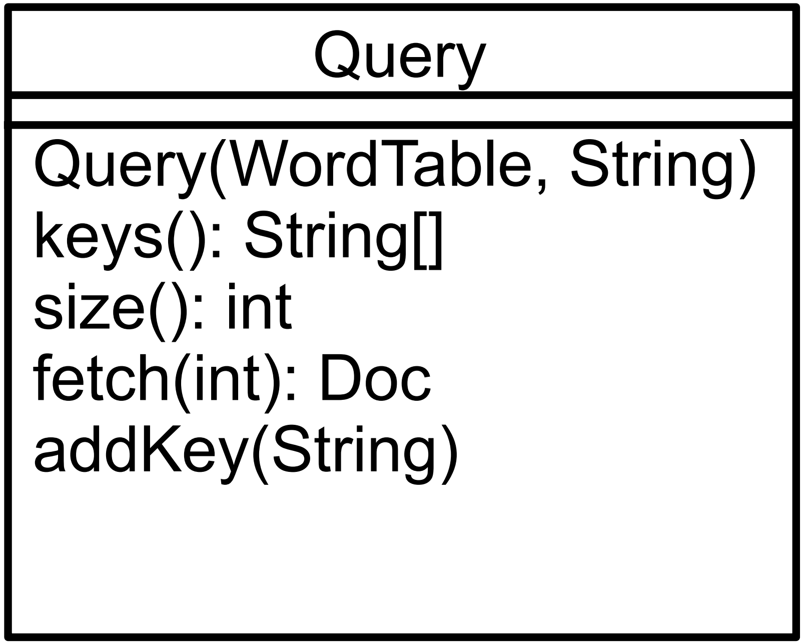
D & A (2)

- adds a new keyword to an existing query
- repeats the check for the new keyword to filter the existing matches (if any)

• to record subsequent keywords in Query:
 → *create addKey() method to add a new keyword to Query*
• *(together with 1.5) → create keys() in Query to observe the keywords*

Query

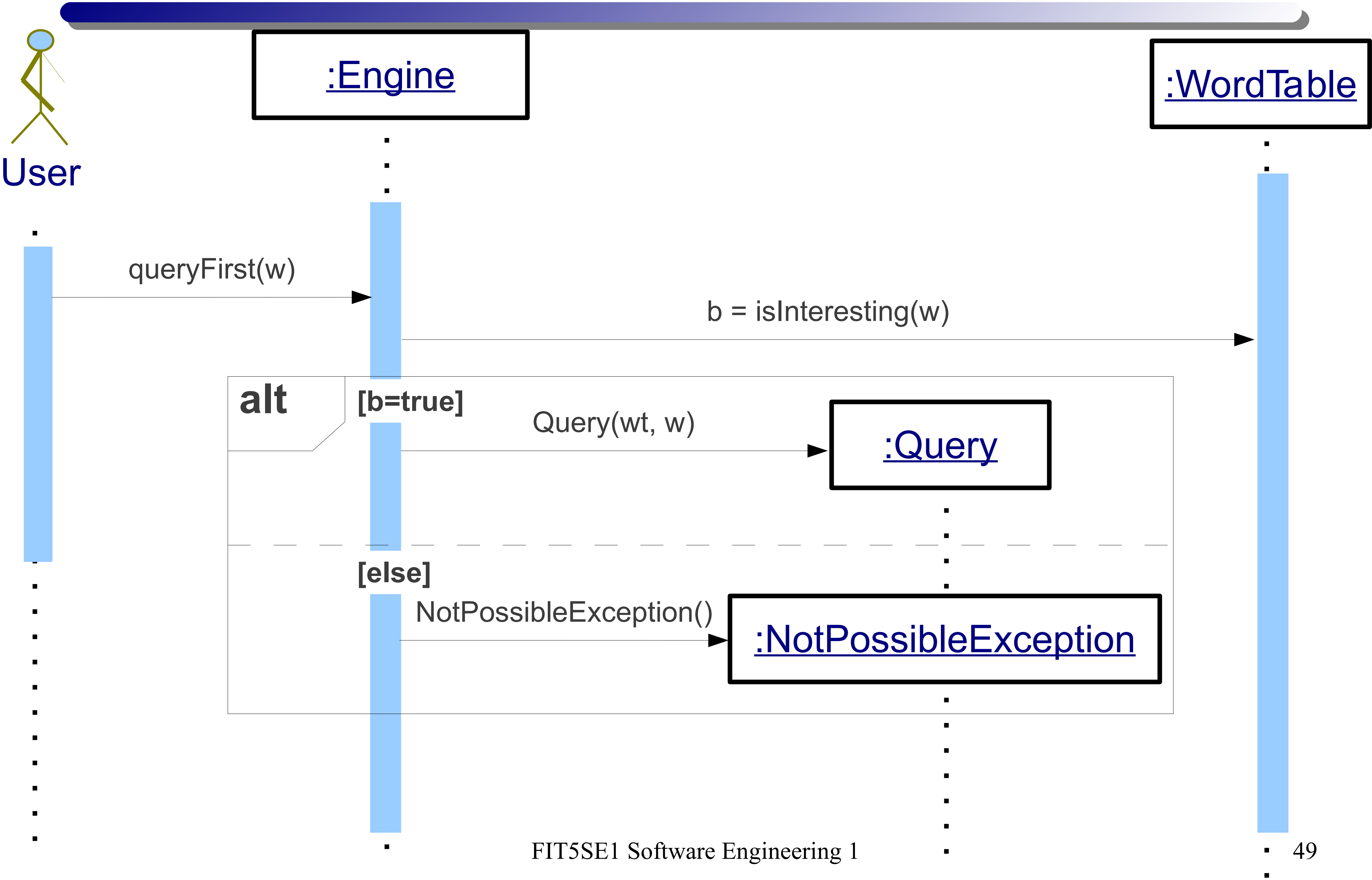
- Updated with constructor and the new methods



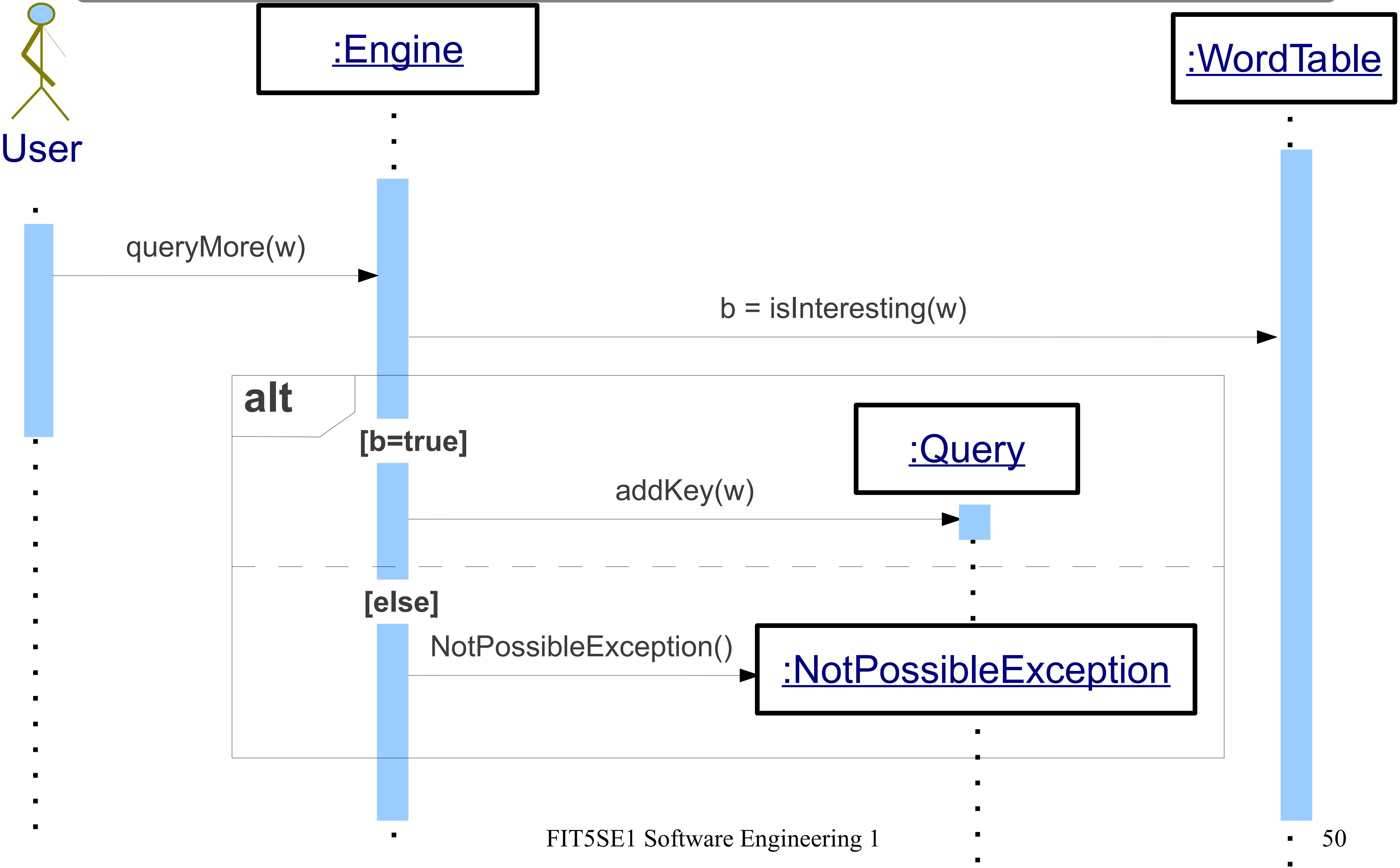
Query

Query(WordTable, String)
keys(): String[]
size(): int
fetch(int): Doc
addKey(String)

sd.queryFirst



sd.queryMore



`Engine.findDoc`

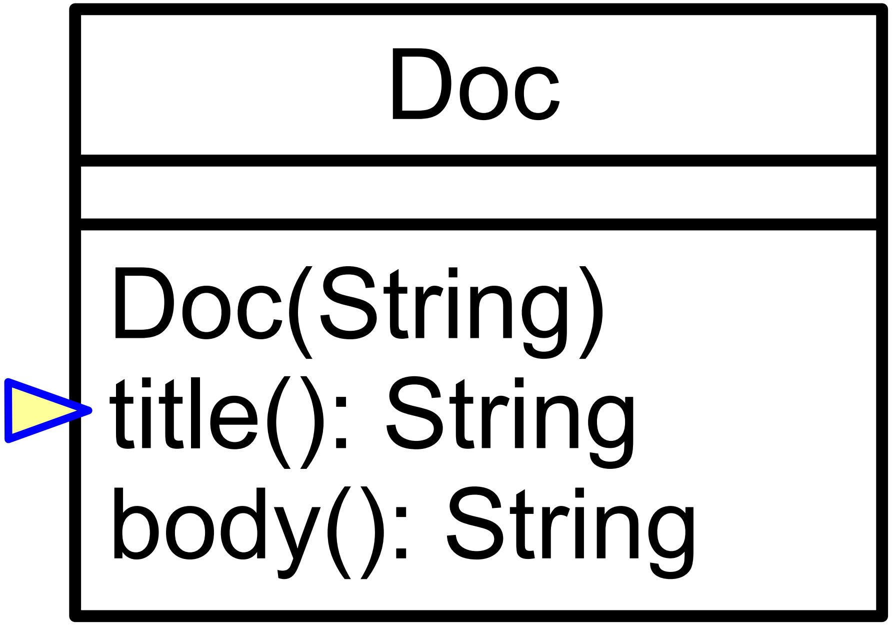
D & A (3)

- For each document, determine if its title matches the given title
- Return the first matching document

- document has title → *create method title() in Doc*
- document titles are re-used many times to determine matches
- needs a fast method to look up document matching a title

→ *records documents and their titles in TitleTable*

Doc

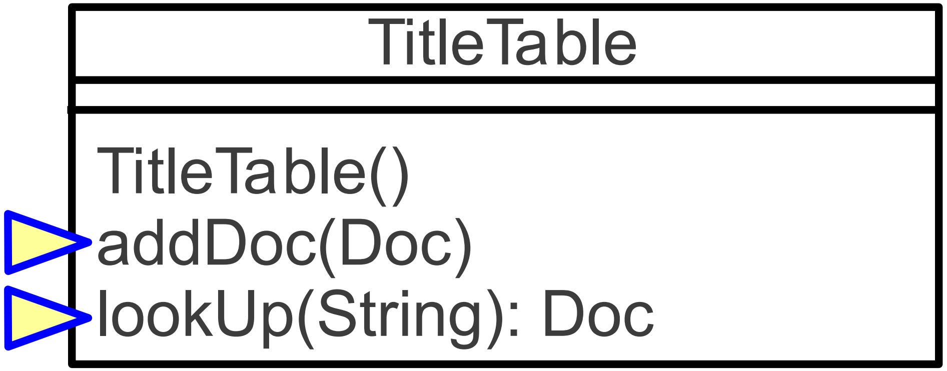


Doc

Doc(String)
title(): String
body(): String

TitleTable

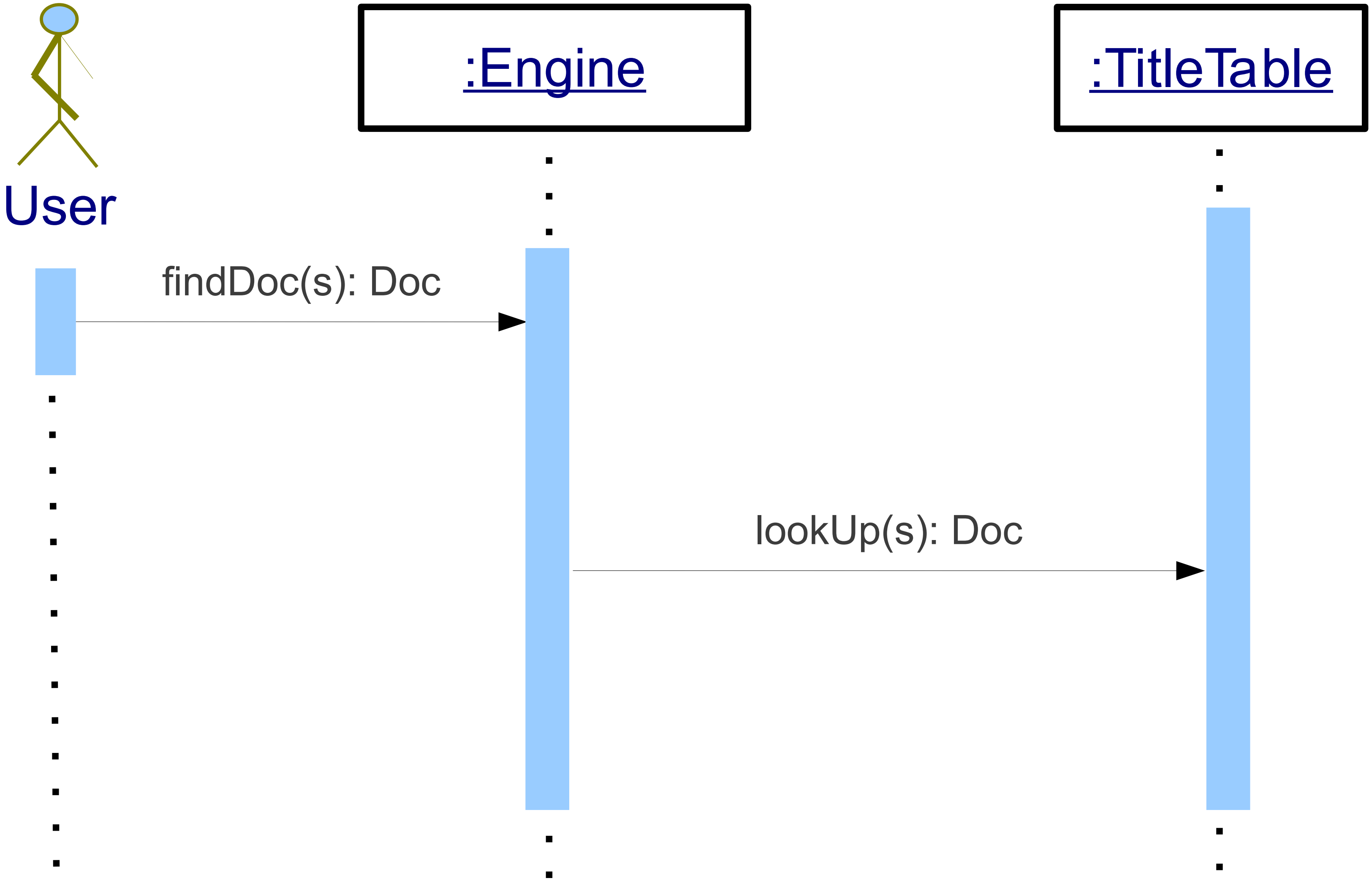
- Iteration abstraction
- Stores documents
- Has operations to add and look up documents



TitleTable

TitleTable()
addDoc(Doc)
lookUp(String): Doc

sd.findDoc



`Engine.addDocs`

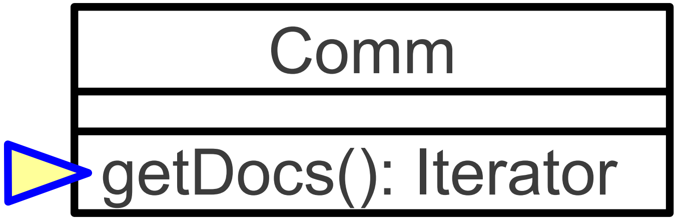
D & A (4.1)

- Contact the site with the given URL
- Retrieve documents from the site
- Add documents to the collection
- Update an existing query (if one is in progress) or creates an empty query object
- Returns the query object

• needs to get documents from a remote web site
→ *create a getDocs() method that returns an Iterator object for the documents*

Comm.getDocs

- A new abstraction Comm
- Added Comm.getDocs



D & A (4.2)

- Contact the site with the given URL
- Retrieve documents from the site
- Add documents to the collection
- Update an existing query (if one is in progress) or creates an empty query object
- Returns the query object

• need to add each document to TitleTable & WordTable
→ *use method TitleTable.addDoc*
→ *use method WordTable.addDoc*

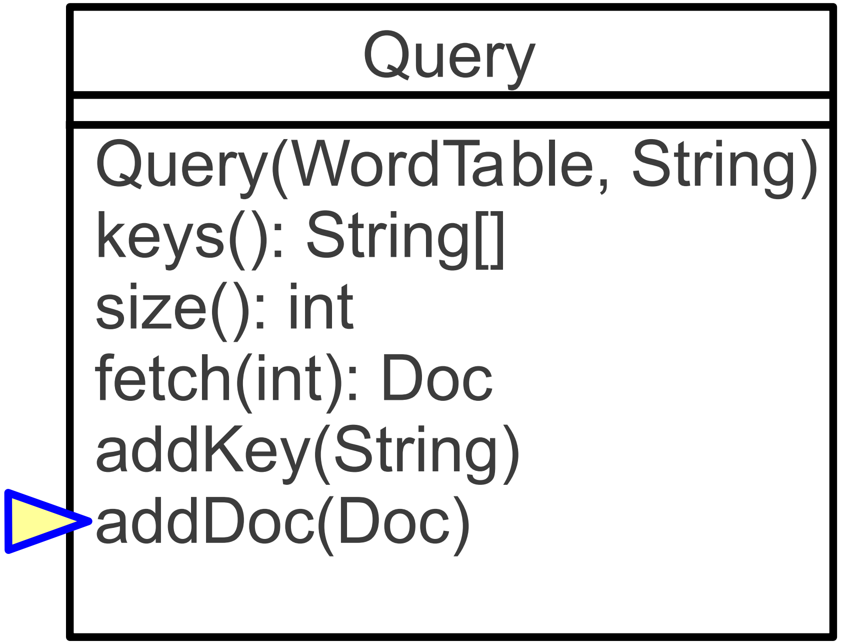
D & A (4.3)

- Contact the site with the given URL
- Retrieve documents from the site
- Add documents to the collection
- Update an existing query (if one is in progress) or creates an empty query object
- Returns the query object

→ *creates addDoc() method to add a new document to Query*

Query

- Updated with addDoc method

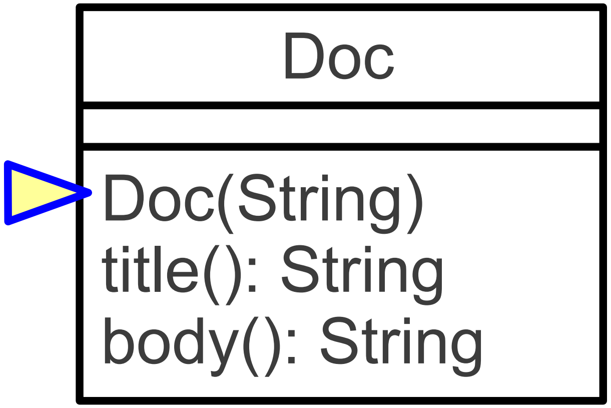


Query

Query(WordTable, String)
keys(): String[]
size(): int
fetch(int): Doc
addKey(String)
addDoc(Doc)

Doc

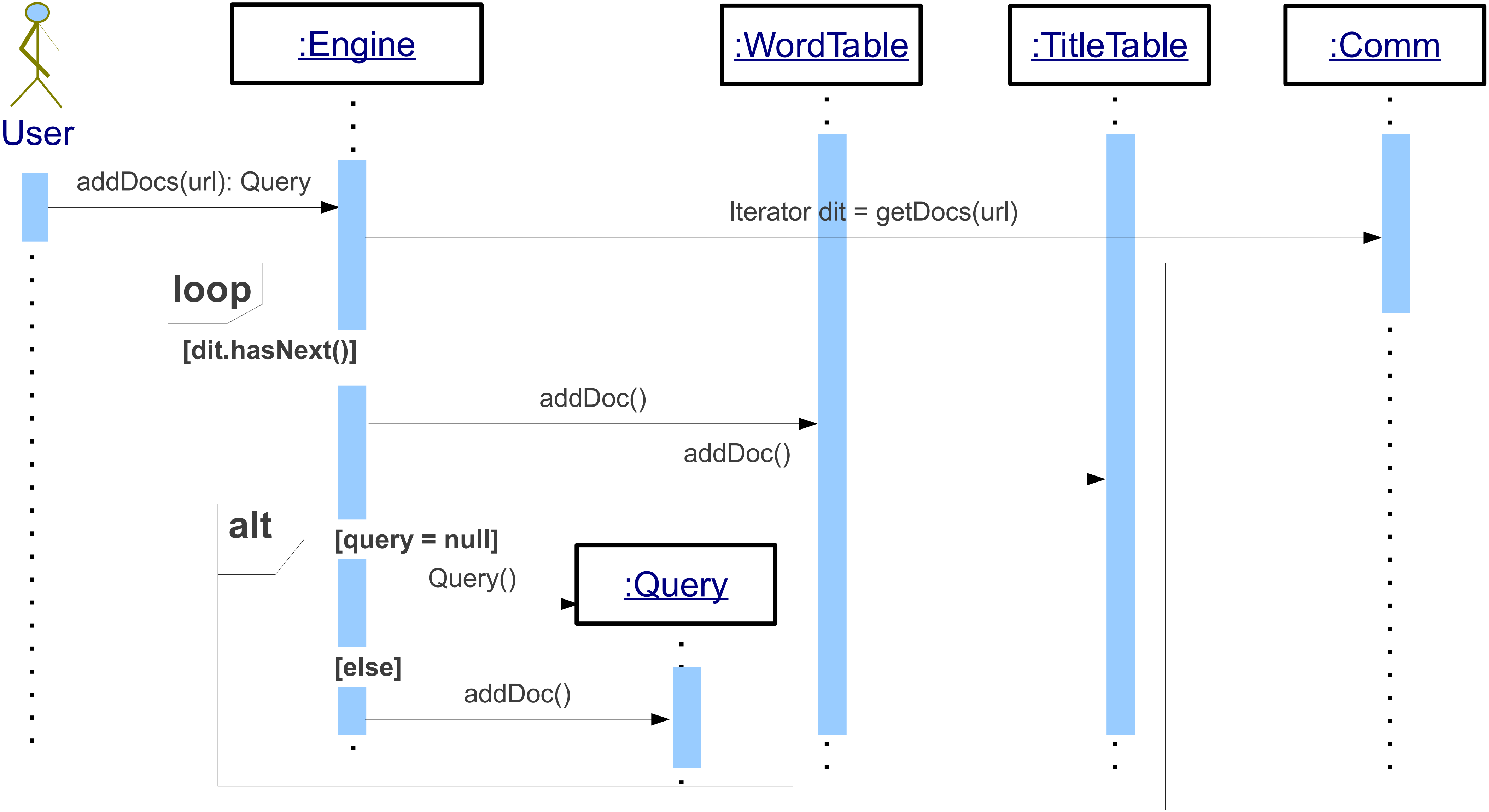
- Updated with a constructor to create Doc object from a string



Doc

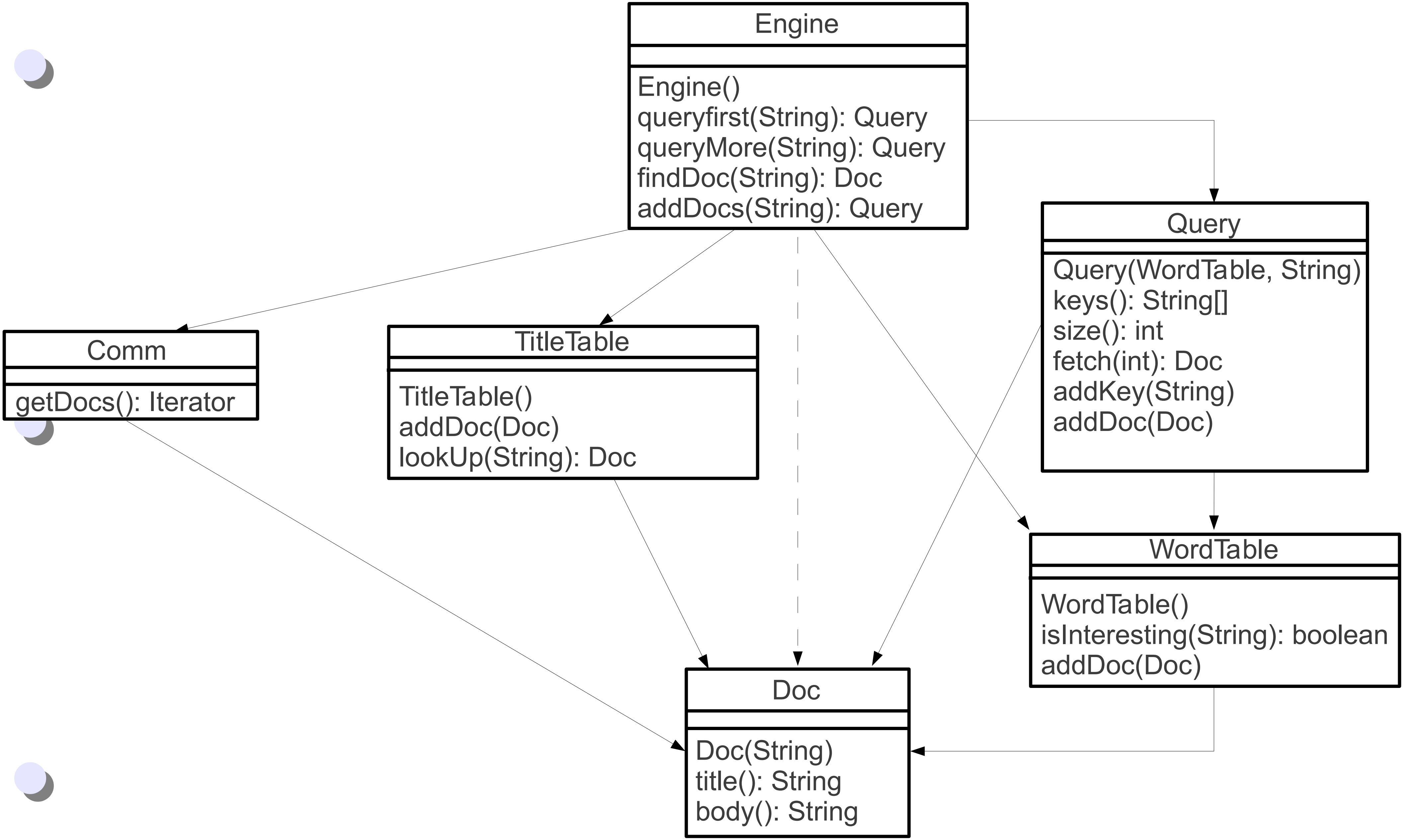
Doc(String)
title(): String
body(): String

sd.addDocs


:Engine
:WordTable
:TitleTable
:Comm
User
addDocs(url): Query
Iterator dit = getDocs(url)
loop
[dit.hasNext()]
addDoc()
addDoc()
alt
[query = null]
Query()
:Query
[else]
addDoc()

Design class diagram & specification

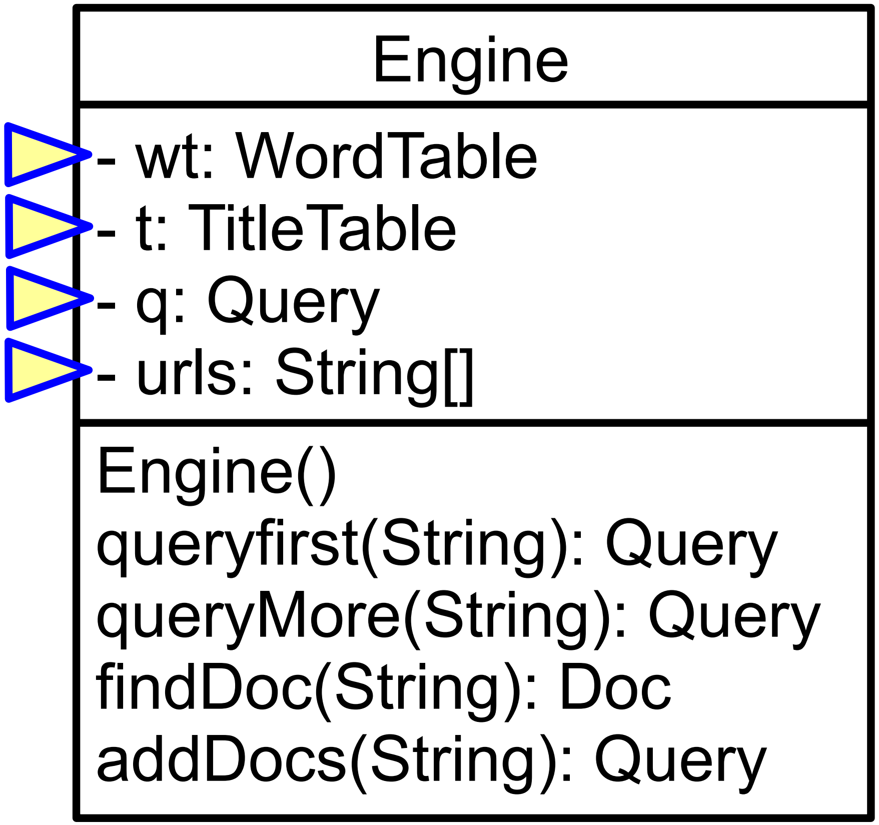
Design class diagram



Engine rep

- ◊ Determined from:
  - specifications of the methods
  - associations with other abstractions

Engine rep



Engine

- wt: WordTable
- t: TitleTable
- q: Query
- urls: String[]

Engine()
queryfirst(String): Query
queryMore(String): Query
findDoc(String): Doc
addDocs(String): Query

Engine specification

```
/**
 * @overview ...(omitted)...
 * @version (iteration) 1.0
 */
class Engine {
  @DomainConstraint(type="WordTable",optional=false)
  private WordTable wt;
  @DomainConstraint(type="TitleTable",optional=false)
  private TitleTable tt;
  @DomainConstraint(type="Query")
  private Query q;

  private String[] urls;
  ///// END version 1.0
} // end Engine
```

WordTable (1)

```
/**
 * @overview Keeps track of interesting and uninteresting words.
 *  Uninteresting words are obtained from a private file.
 *  Records number of times each interesting word occurs in a document.
 * @version (iteration) 1.0
 */
class WordTable {

  /**
   * @effects
   *  If uninteresting-word file cannot be read
   *    throws NotPossibleException
   *  else initialises this to contain all words in the file
   */
  WordTable() throws NotPossibleException
```

WordTable (2)

```
  /**
   * @effects
   *  If w is null or a nonword or an uninteresting word
   *    returns false
   *  else returns true
   */
  boolean isInteresting(String w)

  /**
   * @requires d is not null
   * @modifies this
   * @effects adds to this interesting words of d
   *    with their numbers of occurrences
   */
  void addDoc(Doc d)
} // end WordTable
```

Query (1)

```
/**
 * @overview
 * Provides information about the keywords of a query and
 *    the documents that match those keywords.
 *  Documents are accessed using indexes between 0 and size.
 *  Documents are ordered by the number of matches they
 *   contain, with document 0th containing the most matches.
 * @version (iteration) 1.0
 */
class Query {
  /**
   * @effects returns an empty query
   */
  Query()
```

Query (2)

```
/**
 * @effects returns a count of the documents that match query
 */
int size()

/**
 * @effects
 *  If 0 <= i < size
 *    returns the ith matching document
 *  else
 *    throws IndexOutOfBoundException
 */
Doc fetch (int i) throws IndexOutOfBoundException

/**
 * @effects returns the keywords of this
 */
String[] keys()
```

Query (3)

```
  /**
   * @requires w is not null
   * @modifies this
   * @effects
   *  If this is empty or w is already a keyword in this
   *    throws NotPossibleException
   *  else modifies this to contain w and all keywords already in this
   */
  void addKey(String w) throws NotPossibleException

  /**
   * @requires d is not null
   * @modifies this
   * @effects
   *  If this is not empty and d contains all the keywords of this
   *    adds d to this as a query result
   *  else do nothing
   */
  void addDoc(Doc d)
} // end Query
```

TitleTable (1)

```
/**
 * @overview
 *  Keeps track of documents and their titles.
 *
 * @author dmle
 *
 * @version (iteration) 1.0
 */
class TitleTable {

  /**
   * @effects Initialises this to be empty
   */
  TitleTable()
```

TitleTable (2)

```
  /**
   * @requires d is not null
   * @modifies this
   * @effects
   *  If a document with d's title already in this
   *    throws DuplicateException
   *  else adds d with its title to this
   */
  void addDoc(Doc d) throws DuplicateException

  /**
   * @effects
   *  If t is null or there is no document with title t in this
   *    throws NotPossibleException
   *  else returns the document with title t
   */
  Doc lookUp(String t) throws NotPossibleException
} // end TitleTable
```

Comm.getDocs

```
/**
 * @overview
 *  Represents the communication module responsible for obtaining
 *  documents from remote sites.
 * @version (iteration) 1.0
 */
public class Comm {
  /**
   * @effects
   *  If u isn't a valid URL or the site it names fails to respond
   *    throws NotPossibleException
   *  else returns a generator for documents from site u
   *    (as strings)
   */
  static Iterator getDocs (String u) throws NotPossibleException
} // end Comm
```

DESIGN ITERATION 2

Refinement

Abstraction selection criteria

- Specification is complete but not yet refined
- Has uncertainty
- Increase insight into the design
- Help finish up a part of a design

Which abstraction?

- Three candidates:
  - Comm.getDocs
  - TitleTable
  - Query
- Which one to start first?
  - Comm is considered part of the library, i.e. given
  - TitleTable and Query are both likely
  - choose TitleTable (helps gain further insight into Doc)

KEngine: refinement overview

TitleTable

D & A (5.1)

- addDoc:
  - extracts title from document
- lookup:
  - finds a document given its title

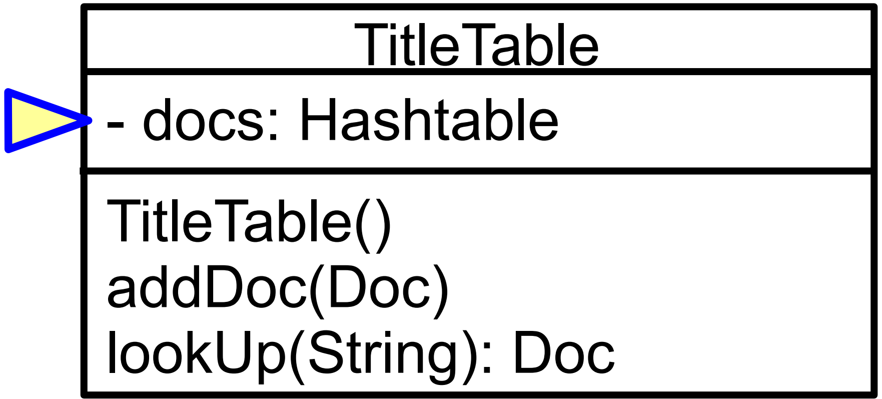
- uses Doc.title() method

D & A (5.2)

- addDoc:
  - extracts title from document
- lookup:
  - finds a document given its title

- document titles are re-used many times
- requires a data structure that maps Docs to strings

→ *uses java.util.Hashtable as the rep of TitleTable*



TitleTable

- docs: Hashtable

TitleTable()
addDoc(Doc)
lookUp(String): Doc

Query

D & A (6.1)

- Query(WordTable, String):
  - find all the documents that contain the keyword with its count
  - keep track of the keyword
  - sort the documents based on the number of occurrences of keywords

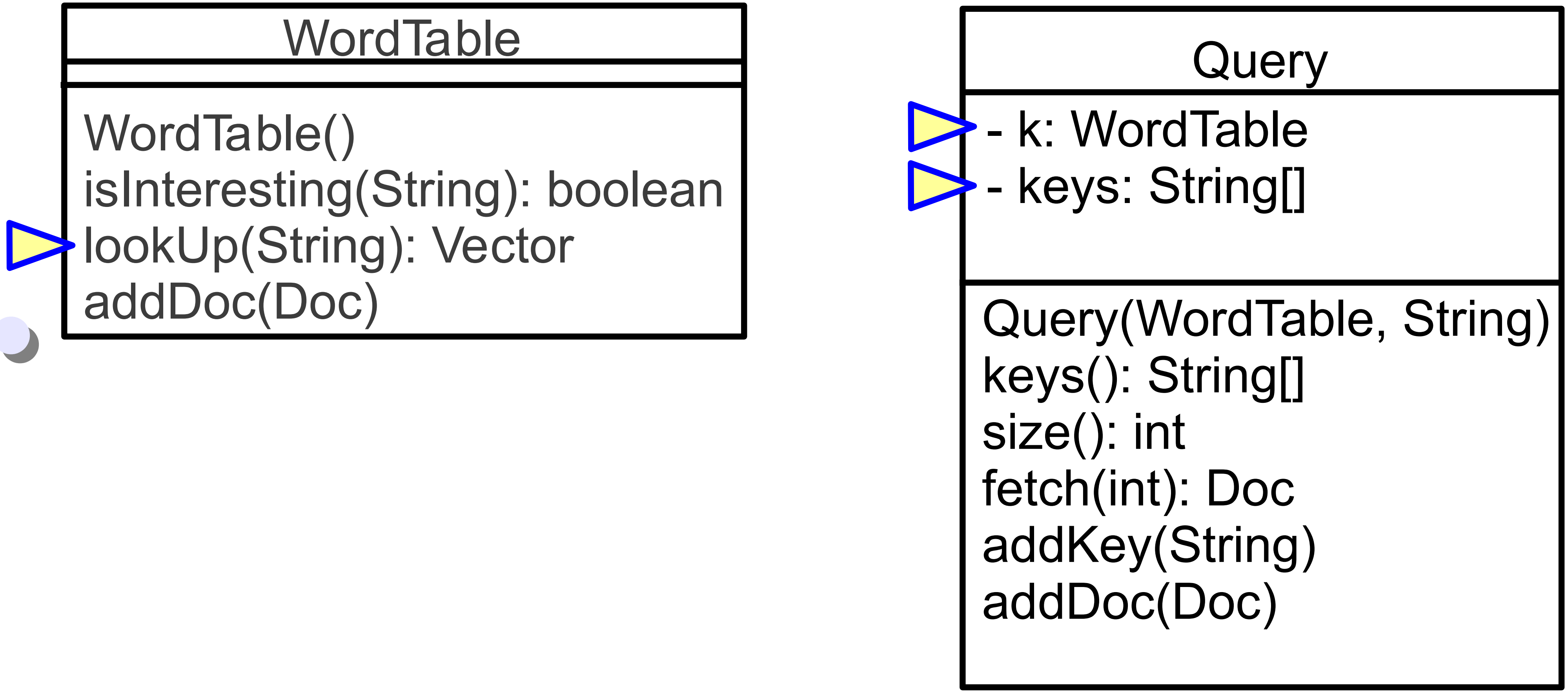
→ *creates WordTable.lookUp method*

D & A (6.2)

- Query(WordTable, String):
  - find all the documents that contain the keyword with its count
  - keep track of the keyword
  - sort the documents based on the number of occurrences of keywords

→ *creates String[] keys in Query to store keywords*

Query & WordTable



WordTable

WordTable()
isInteresting(String): boolean
lookUp(String): Vector
addDoc(Doc)

Query

- k: WordTable
- keys: String[]

Query(WordTable, String)
keys(): String[]
size(): int
fetch(int): Doc
addKey(String)
addDoc(Doc)

D & A (6.3)

◊ Query(WordTable, String):

- find all the documents that contain the keyword with its count
- keep track of the keyword
- sort the documents based on the number of occurrences of keywords

→ sorts documents by keyword frequencies (e.g. using sorted tree)

- *also see 6.7*

D & A (6.4)

◊ fetch(int):

- retrieves the ith document from the current matches

→ needs an index-based collection to store documents (e.g. Vector)

- *also see 6.7*

D & A (6.5)

- addKey(String):
  - check the new keyword for duplicacy
  - find documents containing the new keyword
  - find the new documents that are in the query
  - sort the matches by the sums of the frequencies

→ *performed by checking the Query.keys array*

D & A (6.6)

- addKey(String):
  - check the new keyword for duplicacy
  - find documents containing the new keyword
  - find the new documents that are in the query
  - sort the matches by the sums of the frequencies

→ *uses WordTable.lookUp method*

D & A (6.7)

◊ addKey(String):

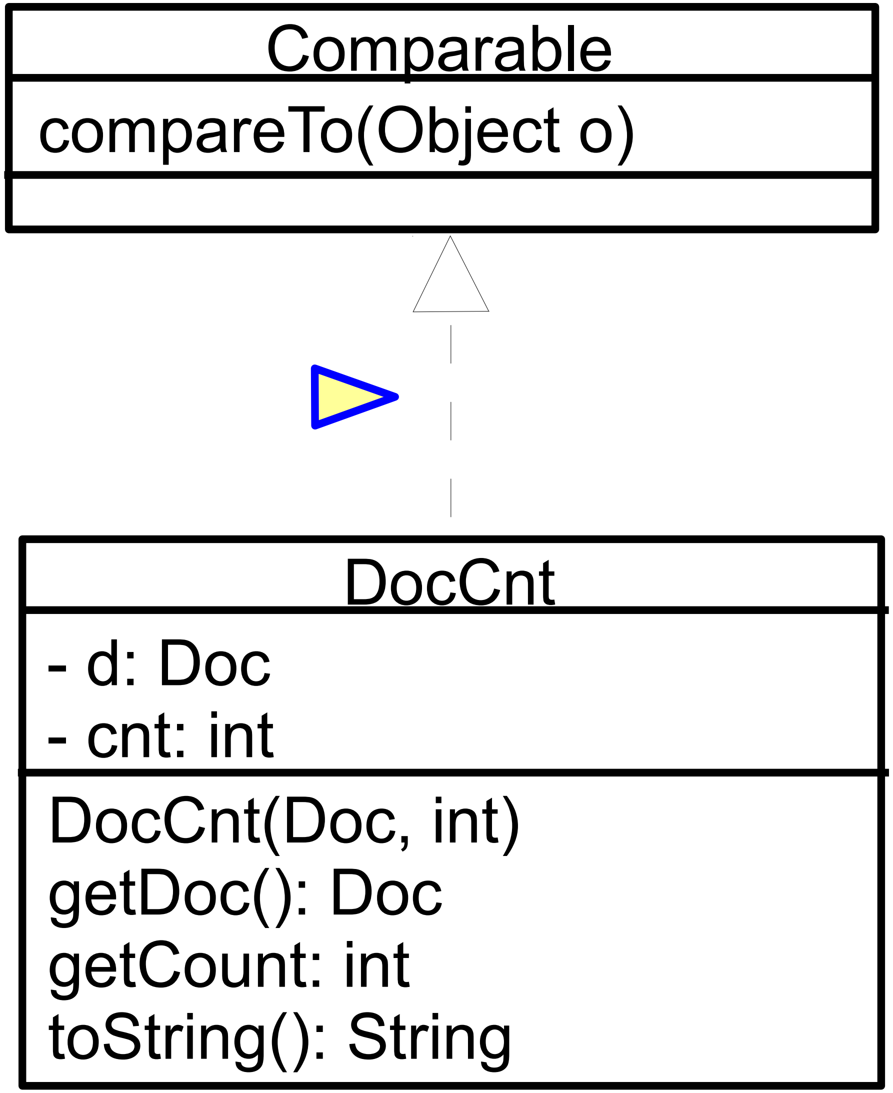
- check the new keyword for duplicacy
- find documents containing the new keyword
- find the new documents that are in the query
- sort the matches by the sums of the frequencies

- needs a fast way to look up document
- also see 6.4

- needs to maintain the sum of frequencies for each match
- needs to sort matches by this sum

→ *creates DocCnt<Document,Count> abstraction for matches*
→ *uses Vector to store matches (DocCnt objects)*
→ *uses quick-sort to sort this vector*

DocCnt



Comparable

compareTo(Object o)

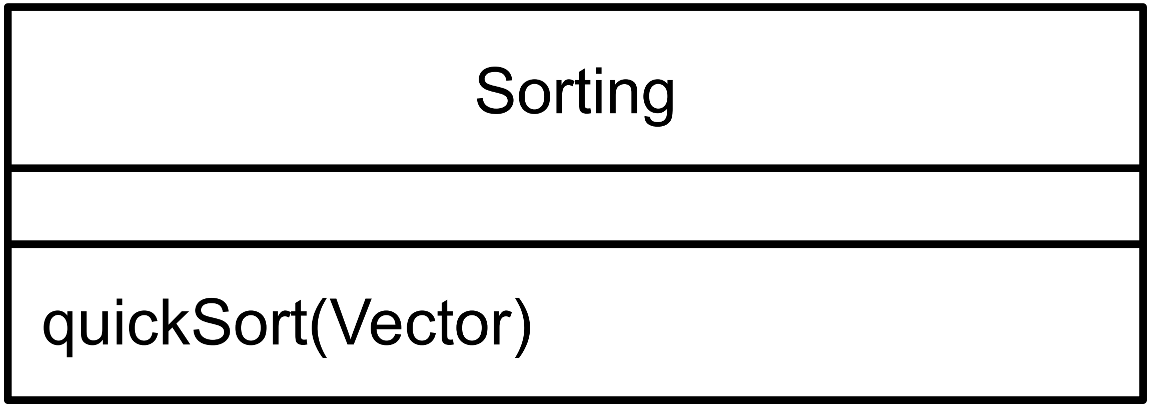
DocCnt

- d: Doc
- cnt: int

DocCnt(Doc, int)
getDoc(): Doc
getCount: int
toString(): String

Sorting

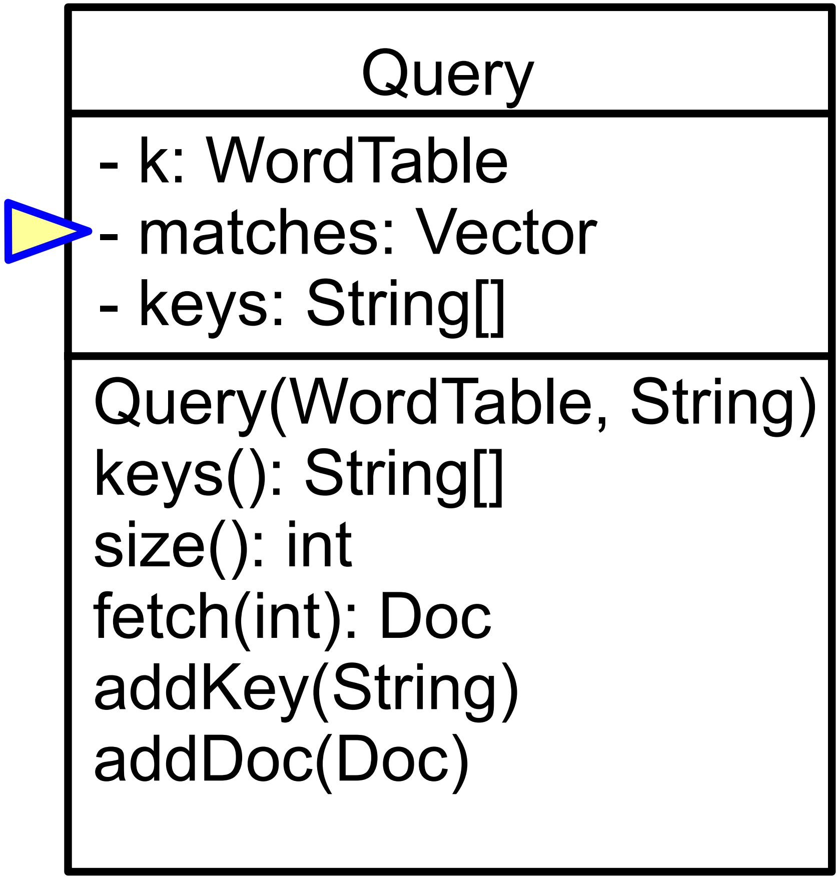
- Sorting.quickSort(Vector)
  - adapts quick-sort for Comparable objects



Sorting

quickSort(Vector)

Query



Query

- k: WordTable
- matches: Vector
- keys: String[]

Query(WordTable, String)
keys(): String[]
size(): int
fetch(int): Doc
addKey(String)
addDoc(Doc)

D & A (6.8)

◊ addDoc(Doc):

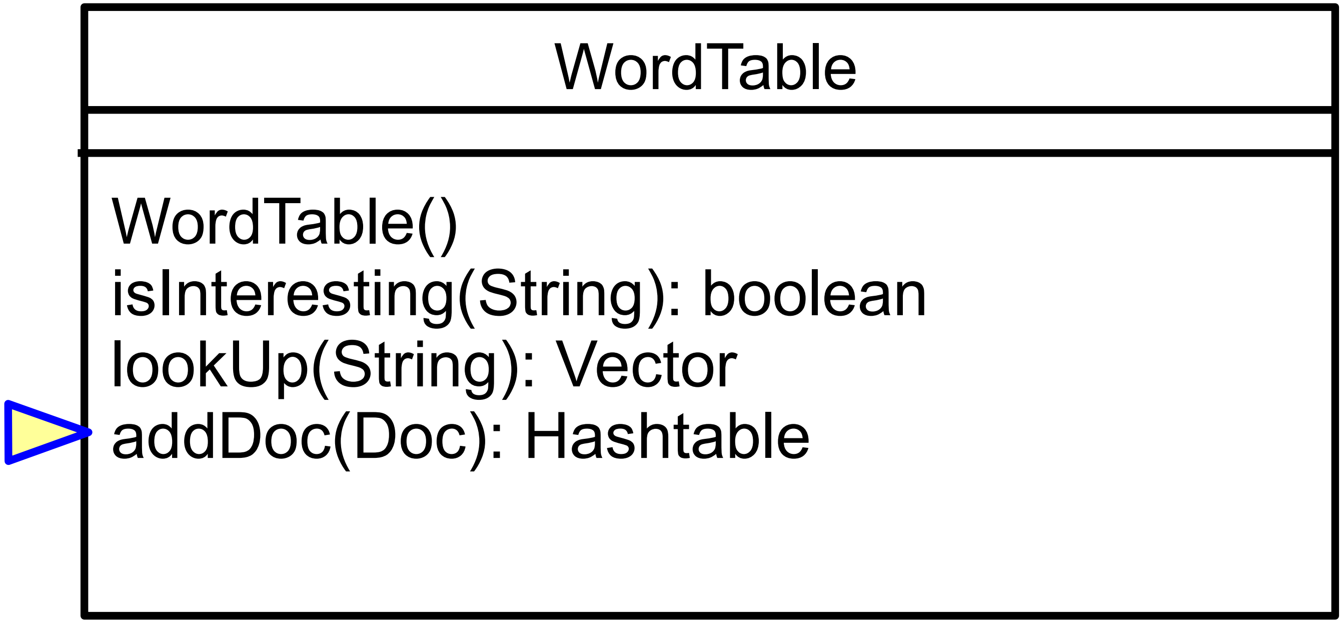
- check each current keyword in the document
- if so, add doc to matches
- update sorting of matches

- needs to know the document keywords and their frequencies, but can be provided by WordTable.addDoc (see sd.addDocs):

→ *updates WordTable.addDoc to return a Hashtable mapping keywords to their frequencies*

→ *modify Query.addDoc(Doc) to become Query.addDoc(Doc, Hashtable)*

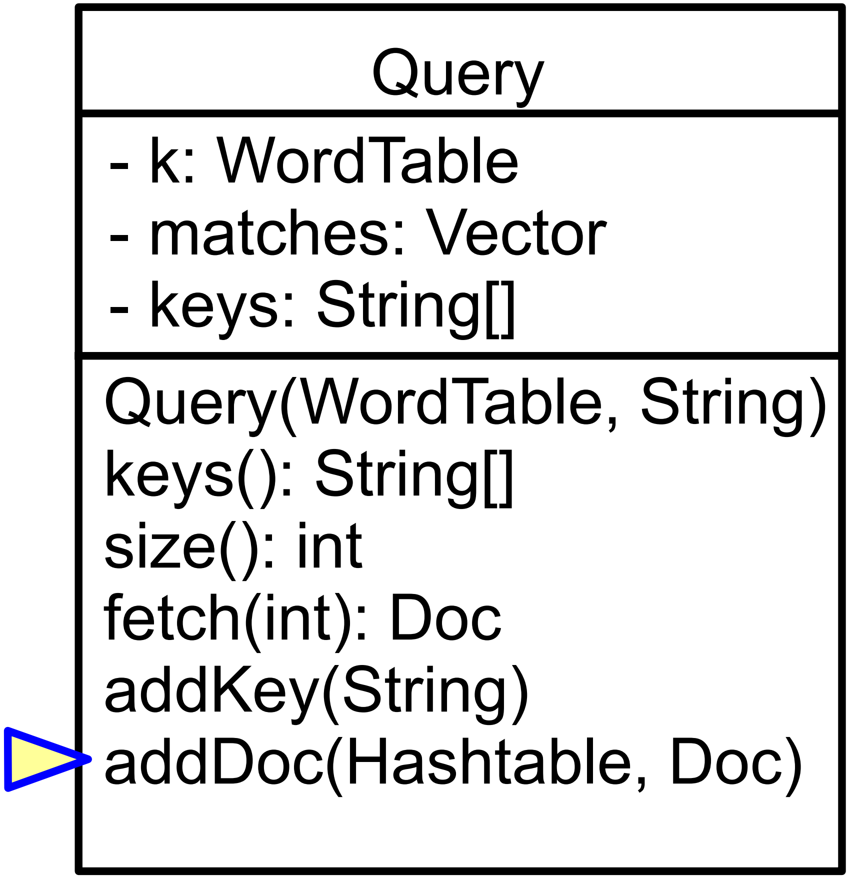
WordTable



WordTable

WordTable()
isInteresting(String): boolean
lookUp(String): Vector
addDoc(Doc): Hashtable

Query



Query

- k: WordTable
- matches: Vector
- keys: String[]

Query(WordTable, String)
keys(): String[]
size(): int
fetch(int): Doc
addKey(String)
addDoc(Hashtable, Doc)

WordTable

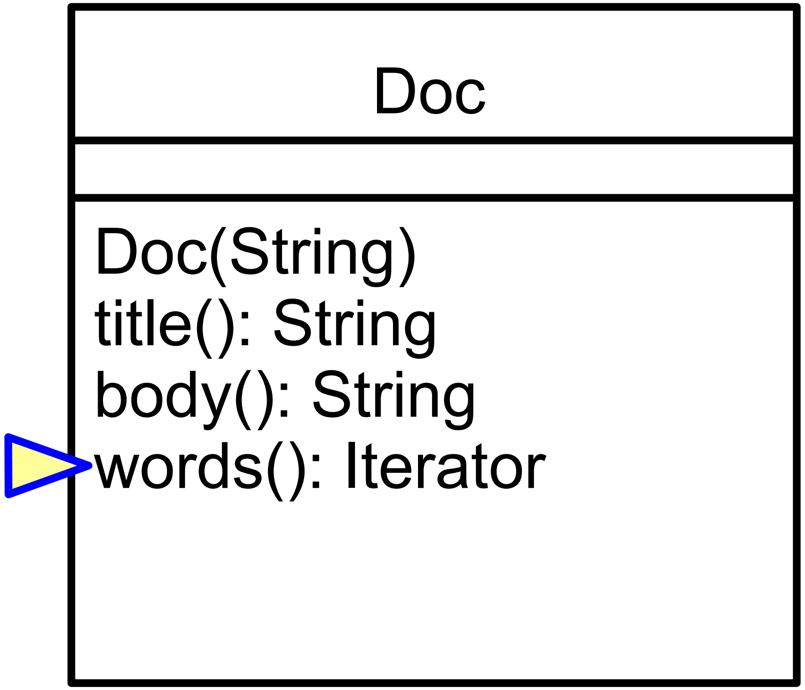
D & A (7)

◊ addDoc(Doc):

- for each word in doc, if it is interesting then creates a DocCnt object from doc and maps it to word
- also adds the mapping <word,DocCnt> to a hash table that is returned as the result

- needs access to an iterator method of Doc that iterates over all words
  → *creates Doc.words(): Iterator method*
- needs to record for each keyword a set of DocCnt objects
  → *adds WordTable.table to map keyword to Vector of DocCnts*
- needs to consider canonical word forms, e.g. student ~ Student
  → *creates Helpers.canon method to convert words to a common format (e.g. lower case)*

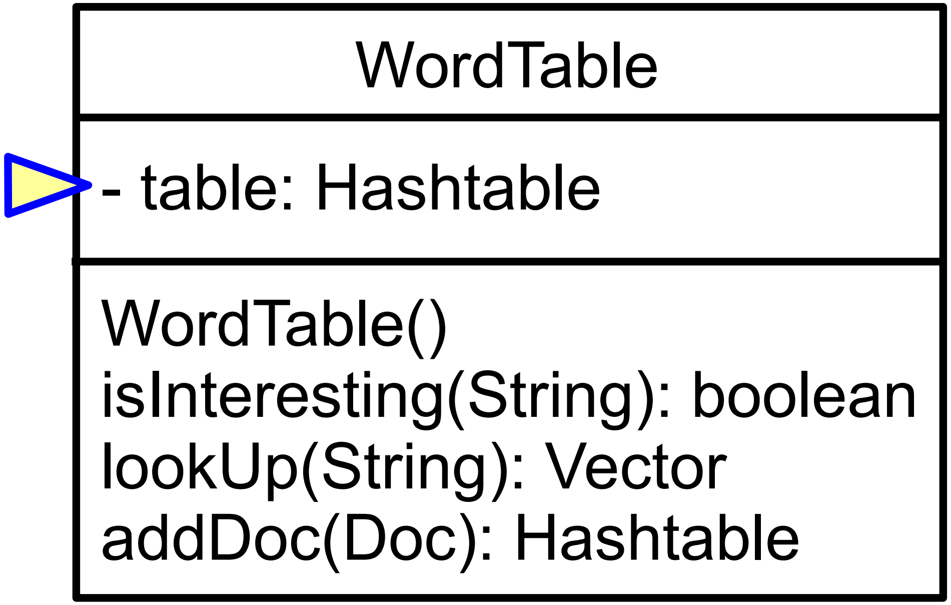
Doc



Doc

Doc(String)
title(): String
body(): String
words(): Iterator

WordTable

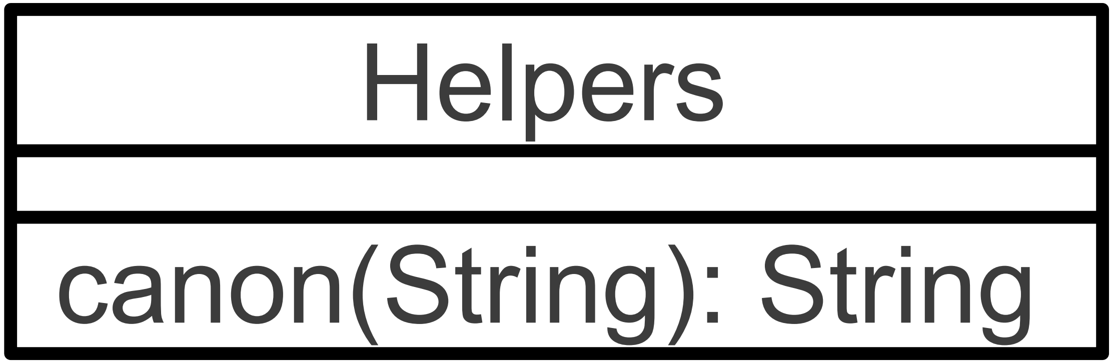


WordTable

- table: Hashtable

WordTable()
isInteresting(String): boolean
lookUp(String): Vector
addDoc(Doc): Hashtable

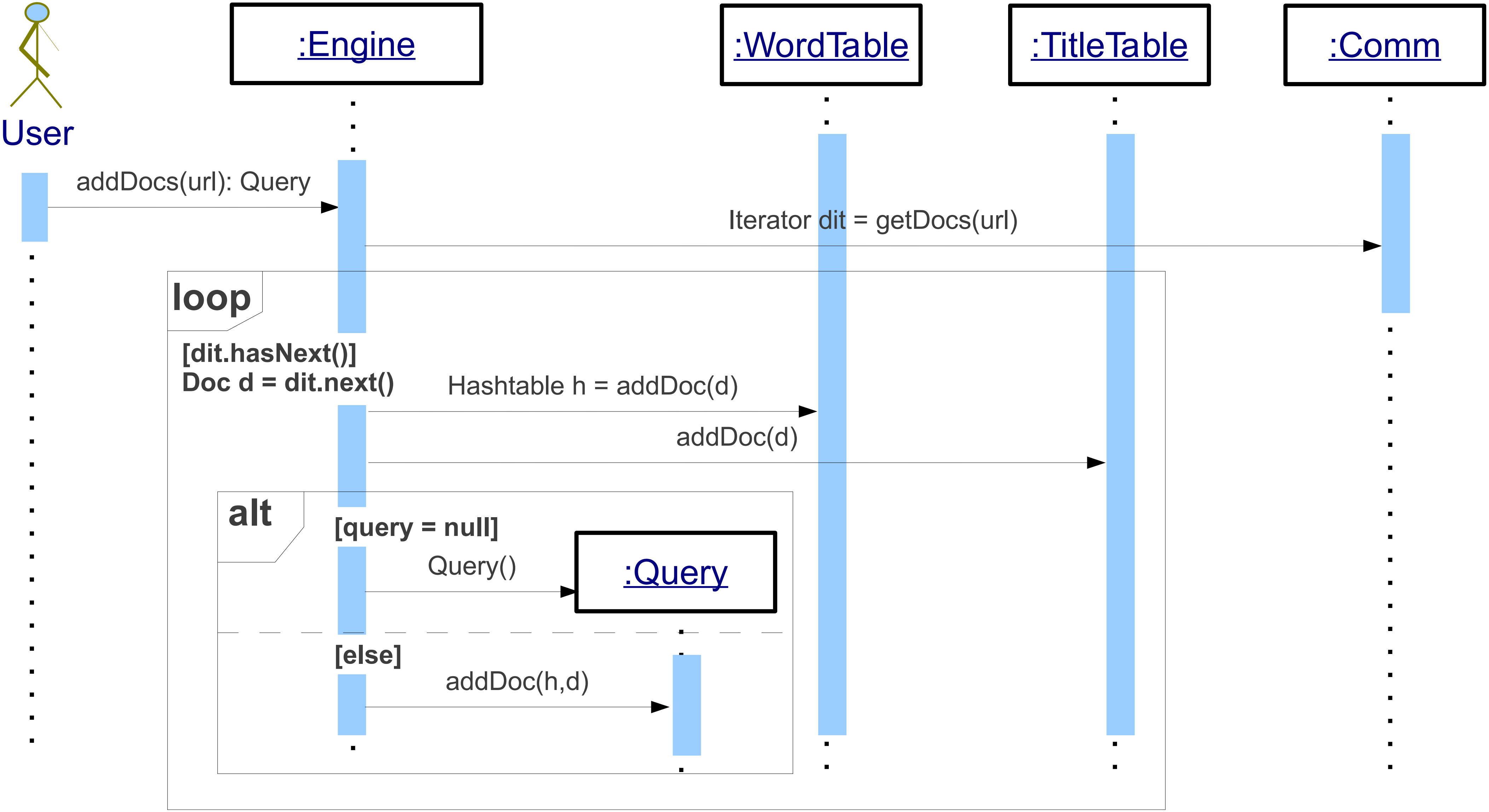
Helpers



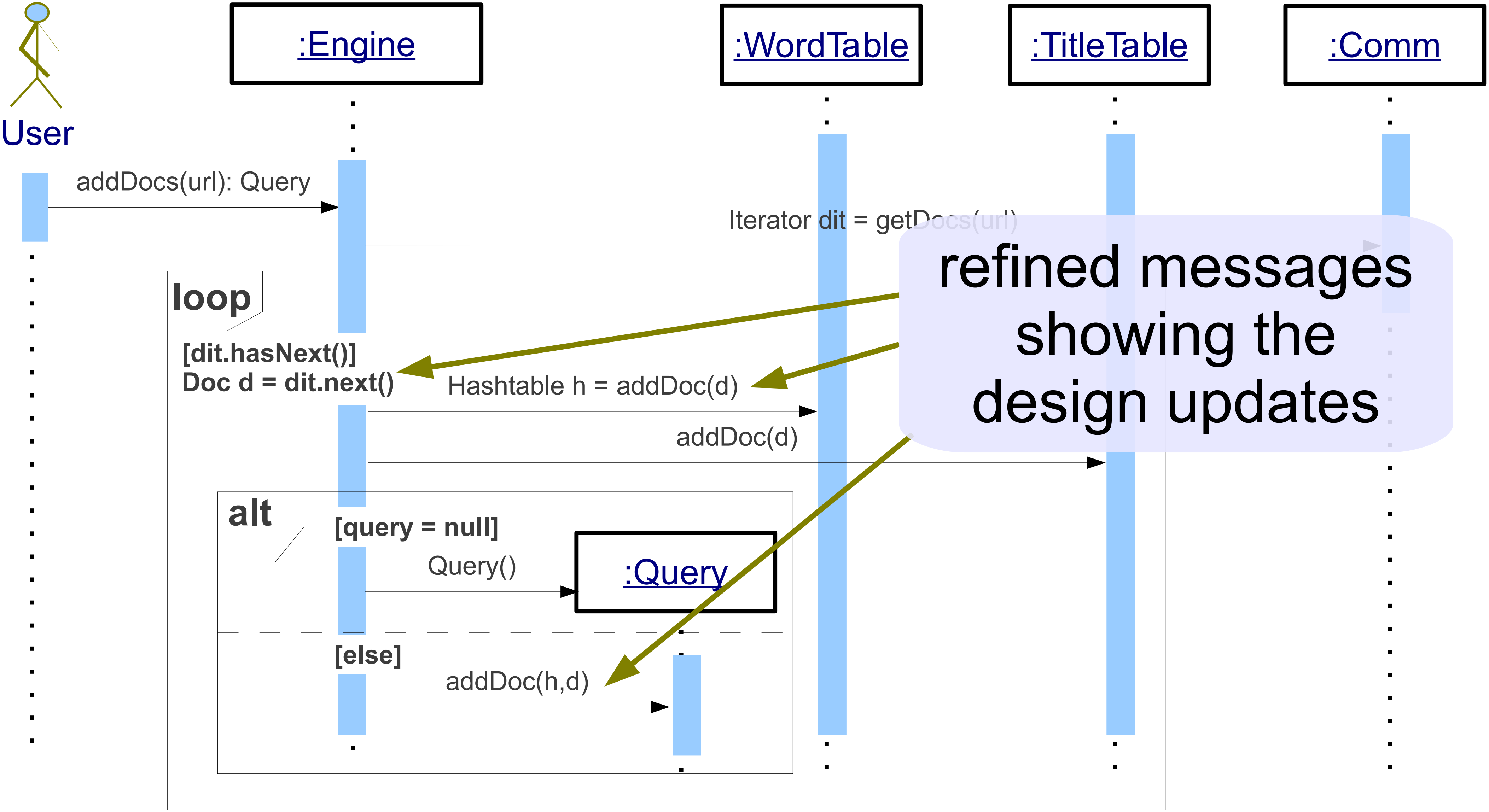
Helpers

canon(String): String

sd.addDocs

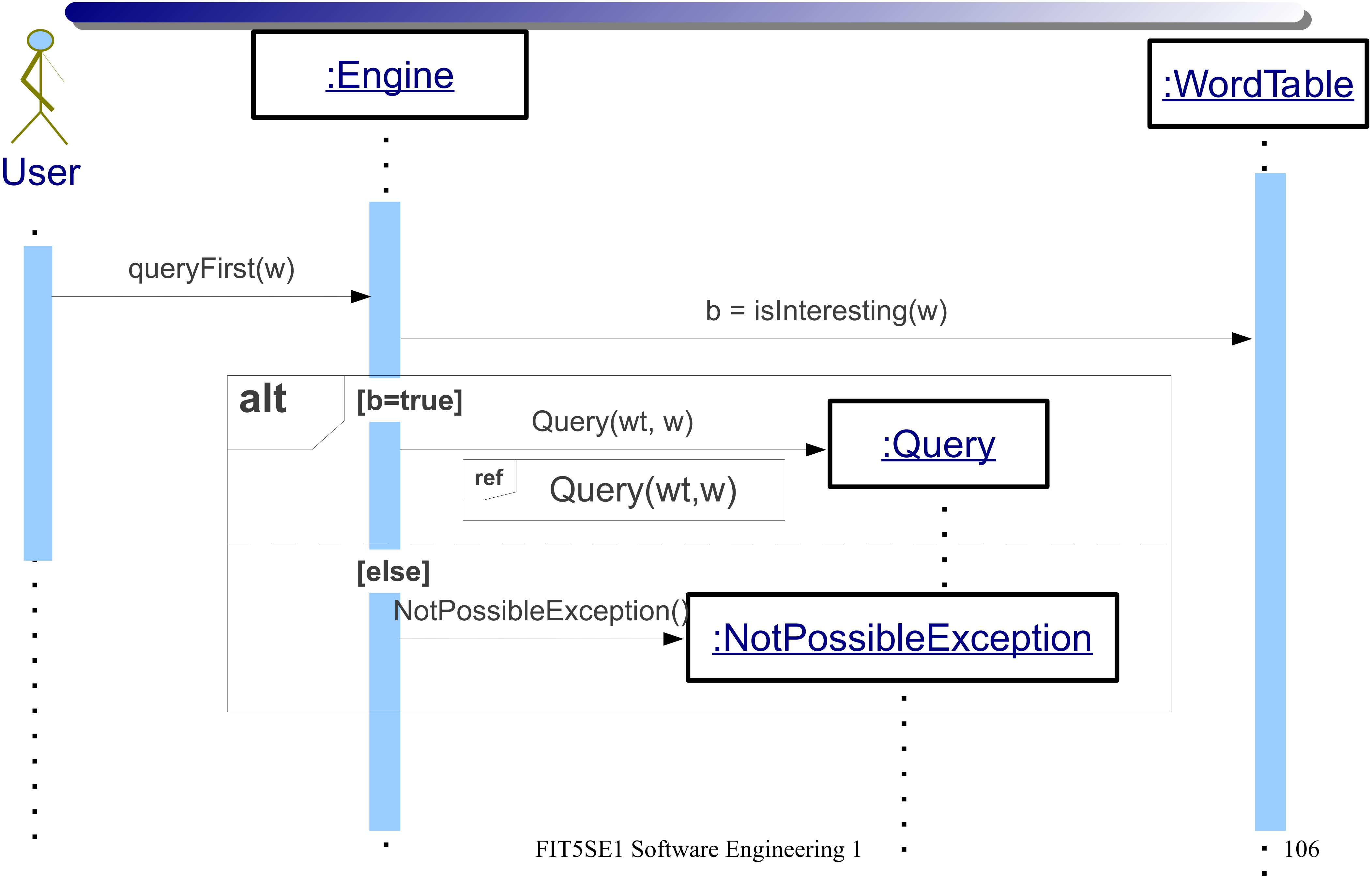

User
:Engine
:WordTable
:TitleTable
:Comm
addDocs(url): Query
Iterator dit = getDocs(url)
loop
[dit.hasNext()]
Doc d = dit.next()
Hashtable h = addDoc(d)
addDoc(d)
alt
[query = null]
Query()
:Query
[else]
addDoc(h,d)

sd.addDocs

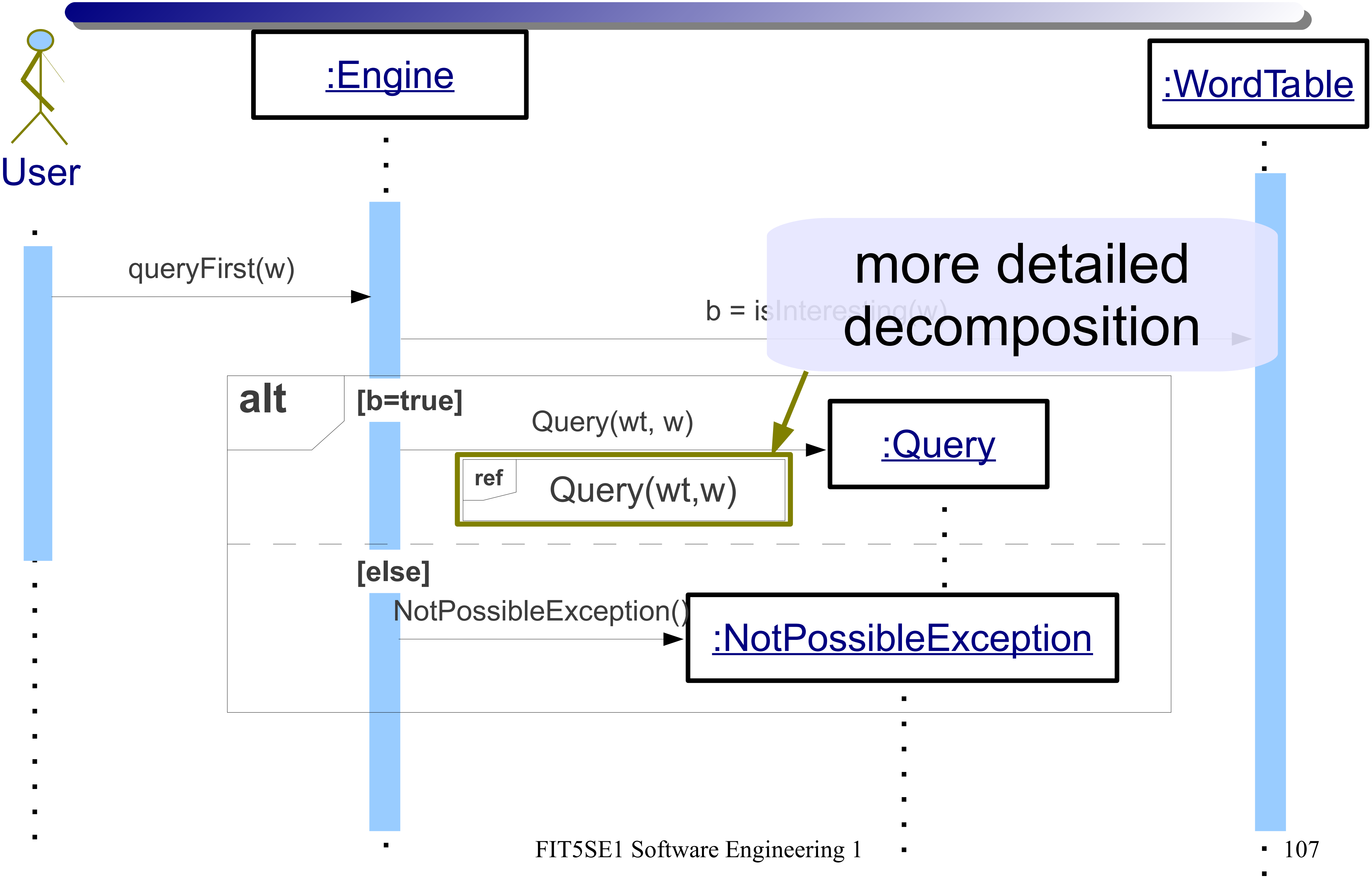


refined messages showing the design updates

sd.queryFirst

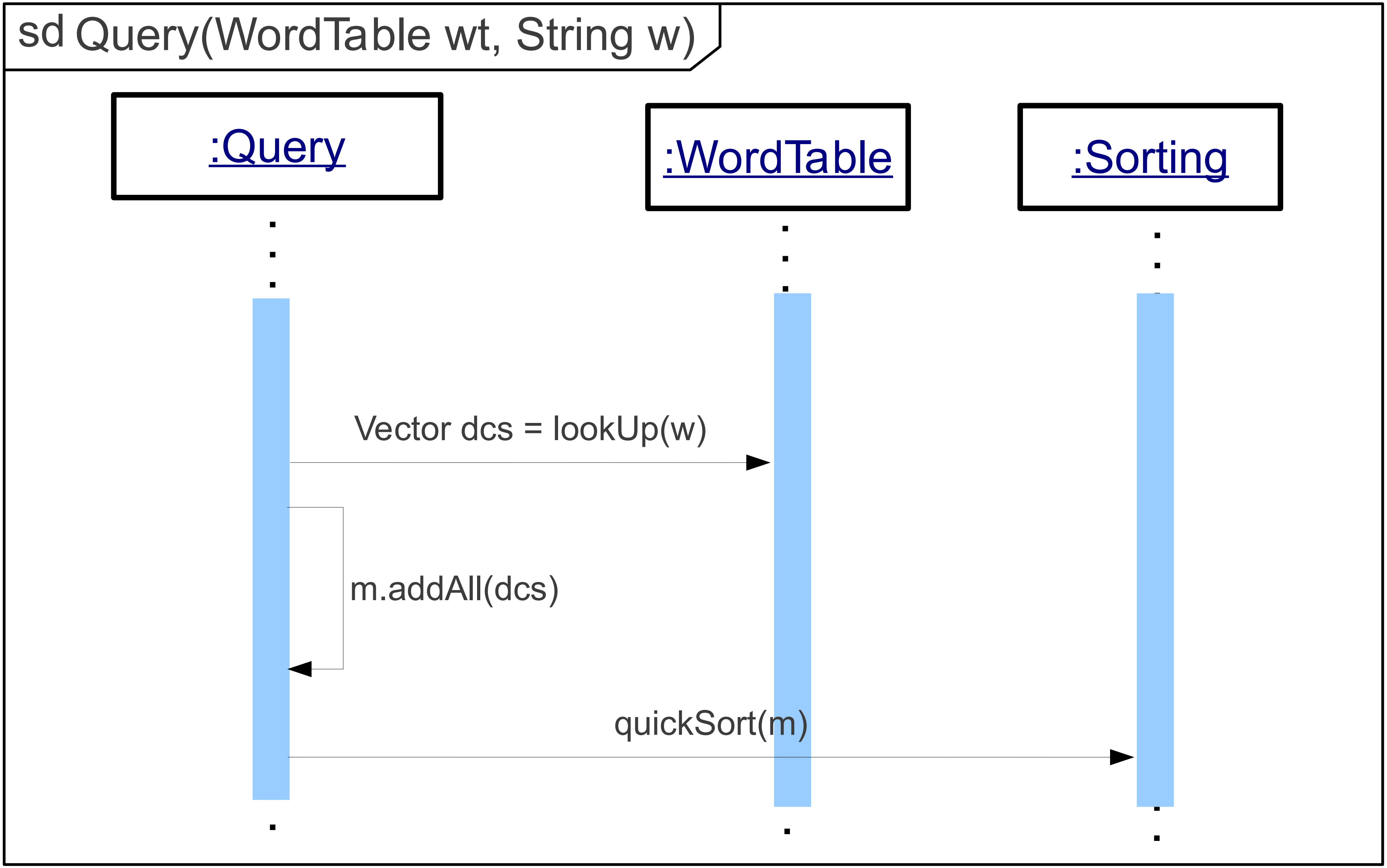


sd.queryFirst



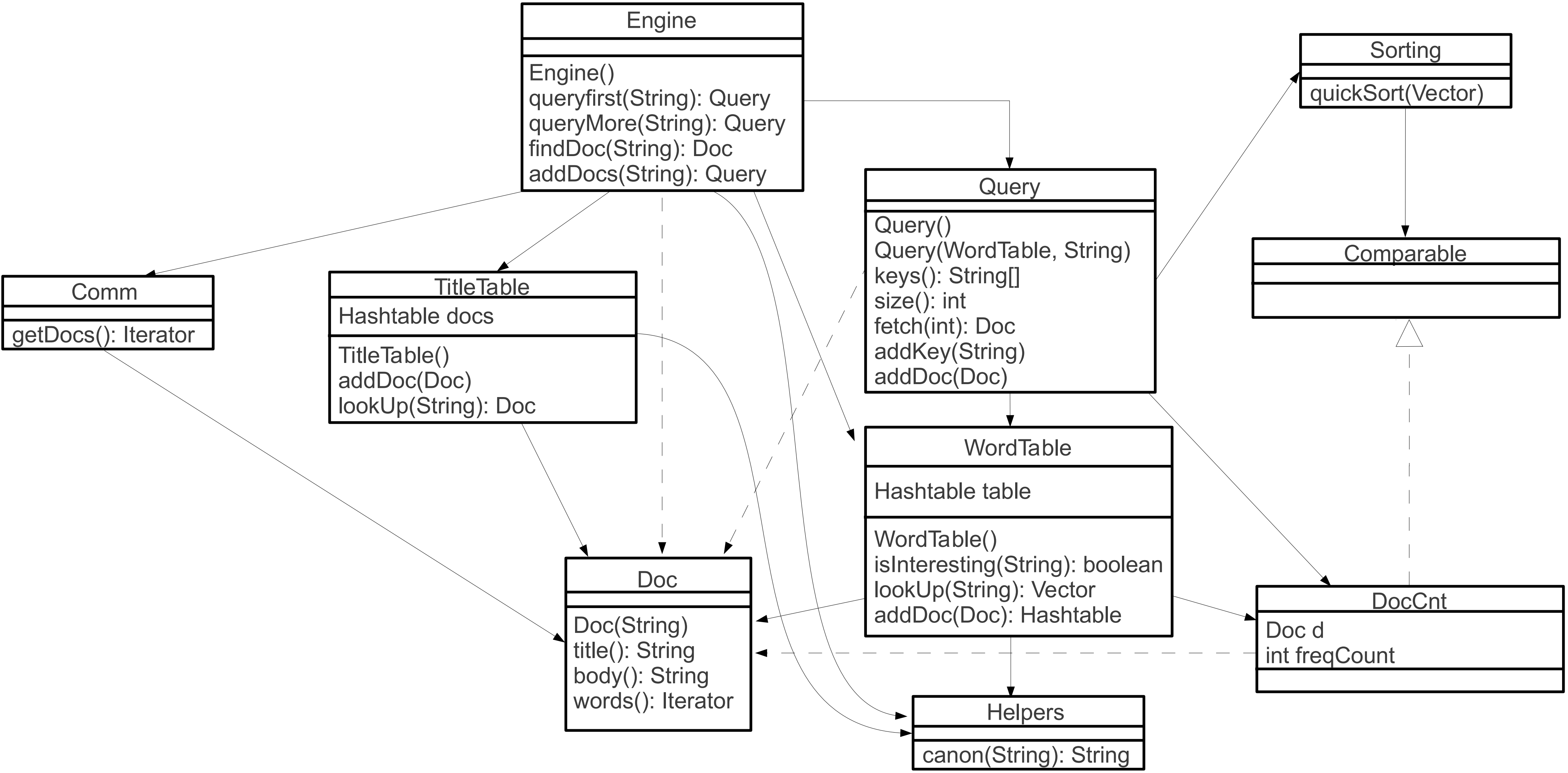
more detailed decomposition

sd.Query(wt,w)


sd Query(WordTable wt, String w)
:Query
:WordTable
:Sorting
Vector dcs = lookUp(w)
m.addAll(dcs)
quickSort(m)

Design class diagram & specification

Design class diagram



Query implementation sketches (1)

```
/**
 * @requires wt and w are not null
 * @effects initialises this to contain w
 *
 * @pseudocode <pre>--- implementation sketch -----
     lookup the key in the WordTable
     sort the matches using quickSort</pre>
 */
Query(WordTable wt, String w)
```

Query Implementation sketches (2)

```
/**
 * @requires ...
 * @modifies ...
 * @effects ...
 *
 * @pseudocode <pre>--- implementation sketch -----
   lookup the new key in the WordTable
   store information about the matches in a hash table
     for each current match, look up document in the
     hash table and if it is there, store in a vector
     sort the vector using quickSort </pre>
 */
void addKey(String w) throws NotPossibleException
```

Query Implementation sketches (3)

```
/**
 * @requires ...
 * @modifies ...
 * @effects ...
 *
 * @pseudocode <pre>--- implementation sketch -----
     use the argument table to get the number of occurrences
       of each current key
     if the document has all the keywords, compute the sum
       and insert the (doc,sum) pair in the vector of matches
     </pre>
 */
void addDoc(Doc d)
```

Reflection

Design process

- Top-down design approach:
  - decomposition by abstraction
- Abstractions are:
  - created as needed
  - refined as necessary
- Design updates make use of design/sequence diagrams

Summary

- Object oriented software design is supported by the UML modelling language
- Design aims to produce an adequate (not best) design
- Design is iterative with later iterations reveal more details about the program structure
- Design is validated using sequence diagrams

Questions?