Implementing Web Intelligence: An Analyzer Framework

NetDB

CS, NTHU,

Fall, 2013

Now you know how a recommendation engine work

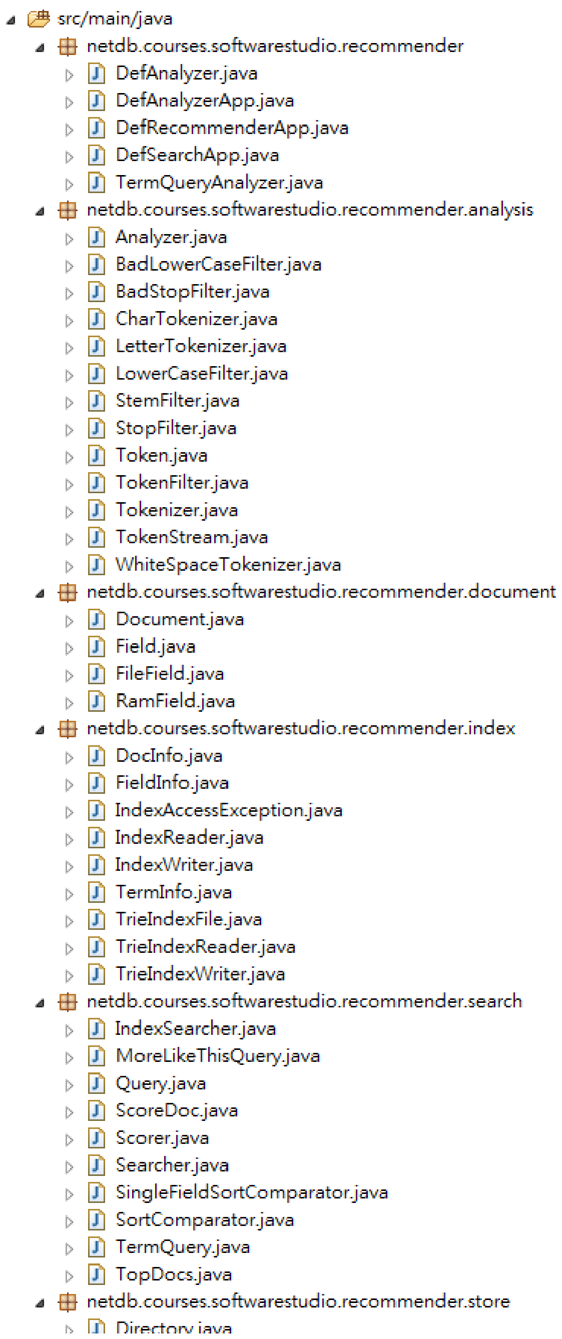
- Strategy
  - User-based
  - Content-based
- How about building up a content-based recommendation engine by yourself!

Our Tiny Recommendation Engine

- To do recommendation, we need to
  1. Analyze(Parse) a set of documents based on "terms"
  2. Store the analyzed statistics about the "terms"
  3. Given a document and an integer *k*, searches and returns the *k* most similar documents in the store
- You can reuse this engine in
  - Your final project
  - Or even future projects

More Detailed Steps

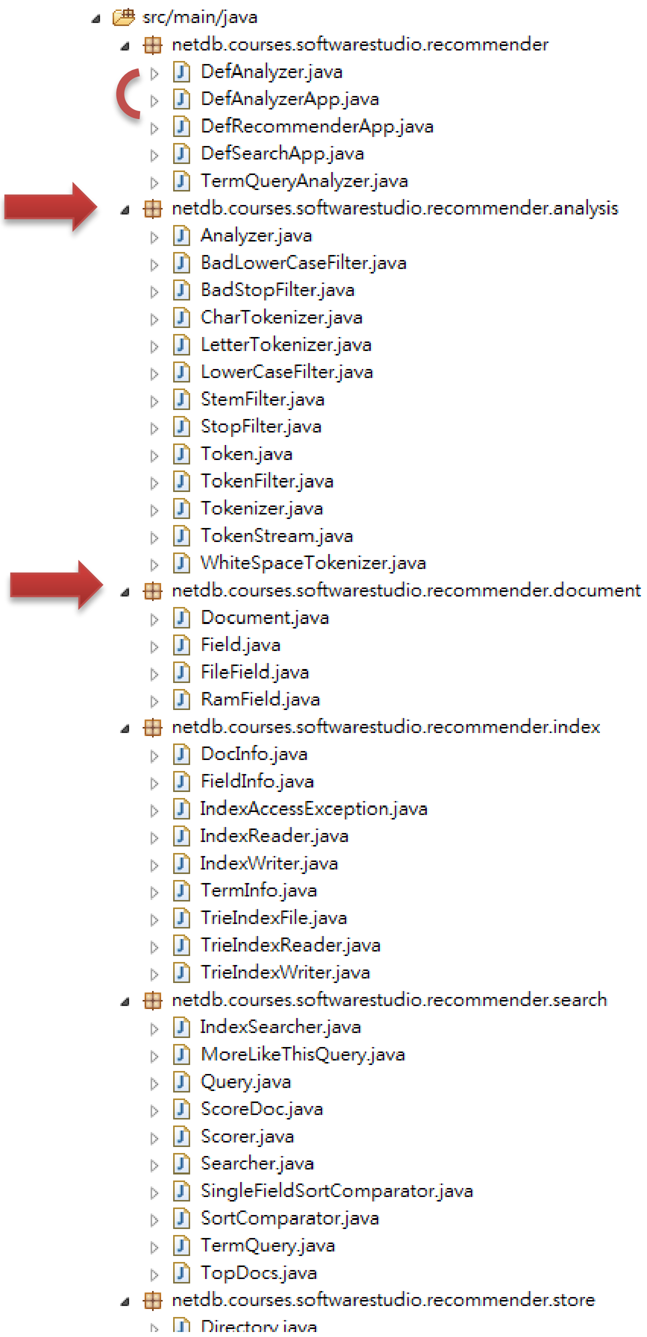
1. Create a **Document** class that is general enough to contain the searchable content
   - E.g., the content of a Definition object, a WiKi page, etc.
2. Implement an **Analyzer** that extracts the “terms” from a Document object (doc for short)
3. An **IndexWriter** that calculates the statistics about the terms/docs (e.g., term frequencies normalized by doc norms) and stores them into an **Index**
   - Basically, an index is a map from the terms to document statistics and IDs
4. An **IndexSearcher** that to find the IDs of the most similar docs to a given doc from index
   - Calculate cosine similarity by looking up the index term-by-term



```
◢ src/main/java
  ◢ netdb.courses.softwarestudio.recommender
    ▷ DefAnalyzer.java
    ▷ DefAnalyzerApp.java
    ▷ DefRecommenderApp.java
    ▷ DefSearchApp.java
    ▷ TermQueryAnalyzer.java
  ◢ netdb.courses.softwarestudio.recommender.analysis
    ▷ Analyzer.java
    ▷ BadLowerCaseFilter.java
    ▷ BadStopFilter.java
    ▷ CharTokenizer.java
    ▷ LetterTokenizer.java
    ▷ LowerCaseFilter.java
    ▷ StemFilter.java
    ▷ StopFilter.java
    ▷ Token.java
    ▷ TokenFilter.java
    ▷ Tokenizer.java
    ▷ TokenStream.java
    ▷ WhiteSpaceTokenizer.java
  ◢ netdb.courses.softwarestudio.recommender.document
    ▷ Document.java
    ▷ Field.java
    ▷ FileField.java
    ▷ RamField.java
  ◢ netdb.courses.softwarestudio.recommender.index
    ▷ DocInfo.java
    ▷ FieldInfo.java
    ▷ IndexAccessException.java
    ▷ IndexReader.java
    ▷ IndexWriter.java
    ▷ TermInfo.java
    ▷ TrieIndexFile.java
    ▷ TrieIndexReader.java
    ▷ TrieIndexWriter.java
  ◢ netdb.courses.softwarestudio.recommender.search
    ▷ IndexSearcher.java
    ▷ MoreLikeThisQuery.java
    ▷ Query.java
    ▷ ScoreDoc.java
    ▷ Scorer.java
    ▷ Searcher.java
    ▷ SingleFieldSortComparator.java
    ▷ SortComparator.java
    ▷ TermQuery.java
    ▷ TopDocs.java
  ◢ netdb.courses.softwarestudio.recommender.store
    ▷ Directory.java
```

Today's Topic

1. Create a **Document** class that is general enough to contain the searchable content
   - E.g., the content of a Definition object, a WiKi page, etc.
2. Implement an **Analyzer** that extracts the “terms” from a Document object (doc for short)
3. An IndexWriter that calculates the statistics about the terms/docs (e.g., term frequencies normalized by doc norms) and stores them into an Index
   - Basically, an index is a map from the terms to document statistics and IDs
4. An IndexSearcher that to find the IDs of the most similar docs to a given doc from index
   - Calculate cosine similarity by looking up the index term-by-term



```
src/main/java
  netdb.courses.softwarestudio.recommender
    DefAnalyzer.java
    DefAnalyzerApp.java
    DefRecommenderApp.java
    DefSearchApp.java
    TermQueryAnalyzer.java
  netdb.courses.softwarestudio.recommender.analysis
    Analyzer.java
    BadLowerCaseFilter.java
    BadStopFilter.java
    CharTokenizer.java
    LetterTokenizer.java
    LowerCaseFilter.java
    StemFilter.java
    StopFilter.java
    Token.java
    TokenFilter.java
    Tokenizer.java
    TokenStream.java
    WhiteSpaceTokenizer.java
  netdb.courses.softwarestudio.recommender.document
    Document.java
    Field.java
    FileField.java
    RamField.java
  netdb.courses.softwarestudio.recommender.index
    DocInfo.java
    FieldInfo.java
    IndexAccessException.java
    IndexReader.java
    IndexWriter.java
    TermInfo.java
    TrieIndexFile.java
    TrieIndexReader.java
    TrieIndexWriter.java
  netdb.courses.softwarestudio.recommender.search
    IndexSearcher.java
    MoreLikeThisQuery.java
    Query.java
    ScoreDoc.java
    Scorer.java
    Searcher.java
    SingleFieldSortComparator.java
    SortComparator.java
    TermQuery.java
    TopDocs.java
  netdb.courses.softwarestudio.recommender.store
    Directory.java
```

Code Time

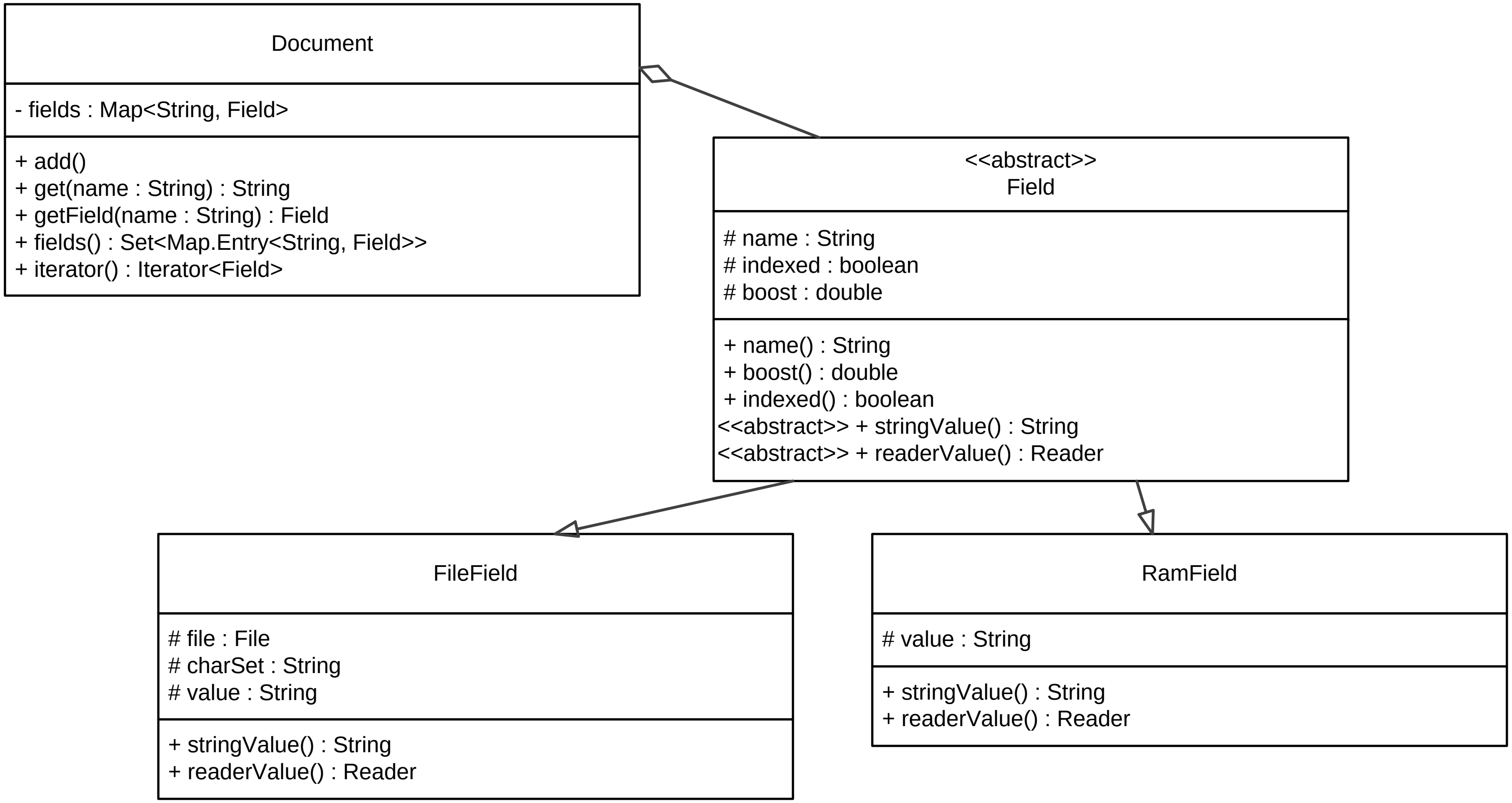
- Checkout the "recommender" project from the repository
- Packages:
  - `document` (for step 1)
  - `analysis` (for step 2)
  - `index` and `store` (for step 3)
  - `search` (for step 4)

Document (1/2)

- A class that general enough to contain the searchable content
- Can we represent a document as a String object?
- Yes, but
  - Sometimes, some fields (e.g., title) of a document is more important than others (e.g., description)
    - We needs a way to boost terms coming from such fields when calculating the cosine similarity
  - You may not be able to load the whole text into memory a time
    - E.g., when text is loaded from a (large) file, or is transmitted from a remote machine

Document (2/2)

- A more rigorous design:



Document

- fields : Map<String, Field>

+ add()
+ get(name : String) : String
+ getField(name : String) : Field
+ fields() : Set<Map.Entry<String, Field>>
+ iterator() : Iterator<Field>

<<abstract>>
Field

name : String
indexed : boolean
boost : double

+ name() : String
+ boost() : double
+ indexed() : boolean
<<abstract>> + stringValue() : String
<<abstract>> + readerValue() : Reader

FileField

file : File
charSet : String
value : String

+ stringValue() : String
+ readerValue() : Reader

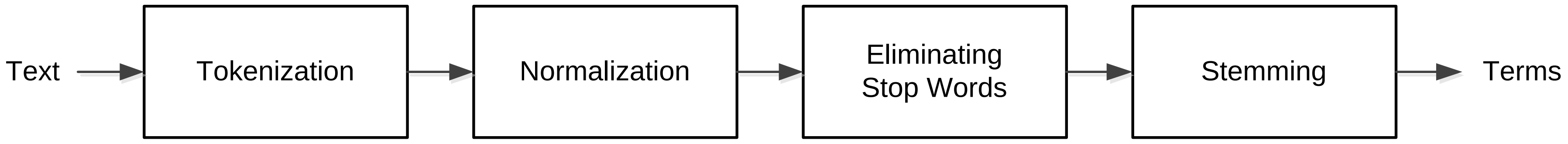
RamField

value : String

+ stringValue() : String
+ readerValue() : Reader

The Text Analyzer Framework

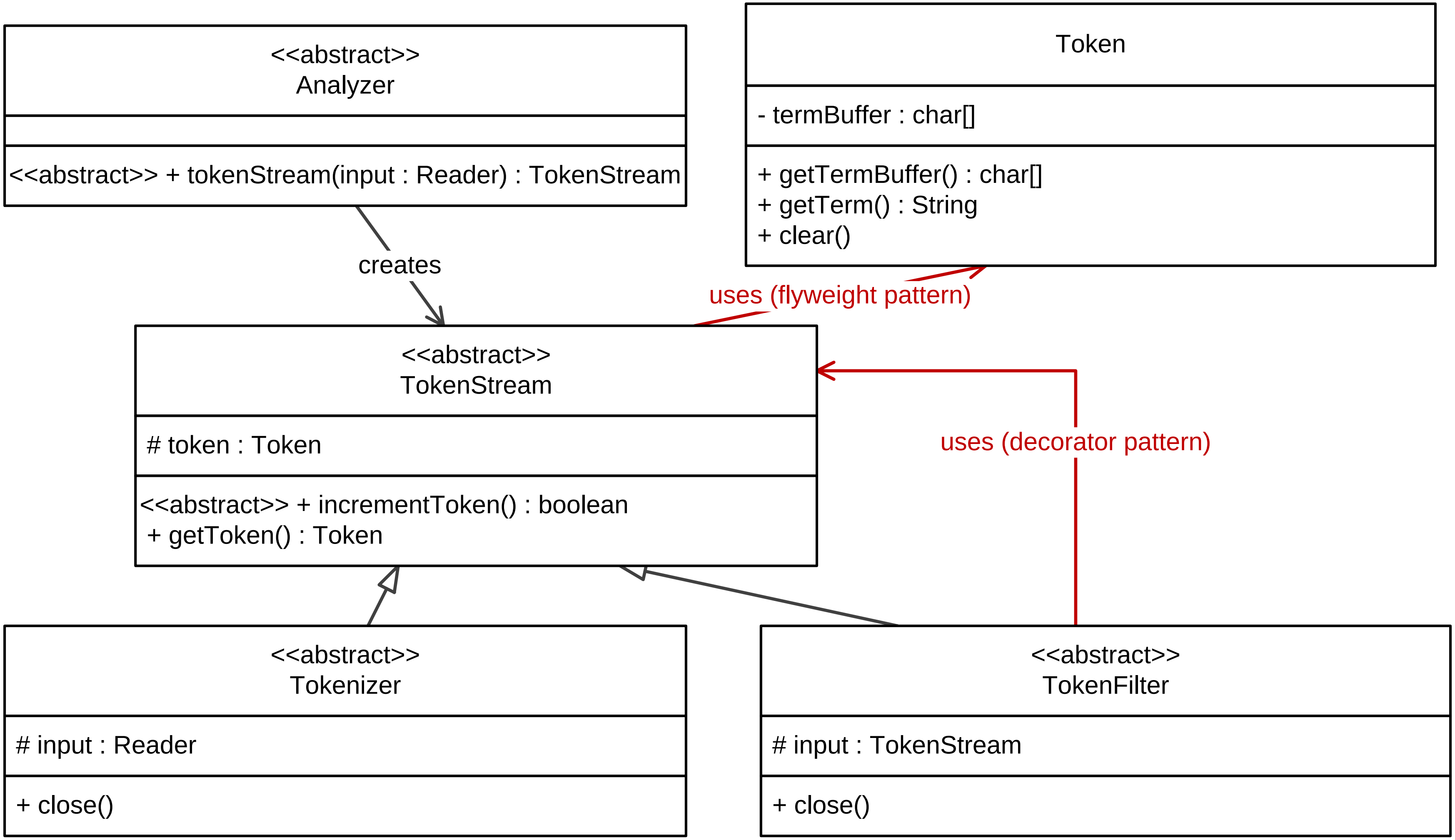
- Input: A text
- Output: terms
- Steps:



Why not String?

- Recall that String provides a method `split()`?
  - We can break a string `s` in to tokens by `s.split(" ")`
  - More complex split regex can be instructed
- Why do we need to implement an analyzer framework ourselves?
- Again, you may not be able to load the whole text into memory a time
  - E.g., when text is loaded from a (large) file, or is transmitted from a remote machine
- We need a text analyzer framework reads the text (from the beginning to the end) only **once** to parse all the tokens

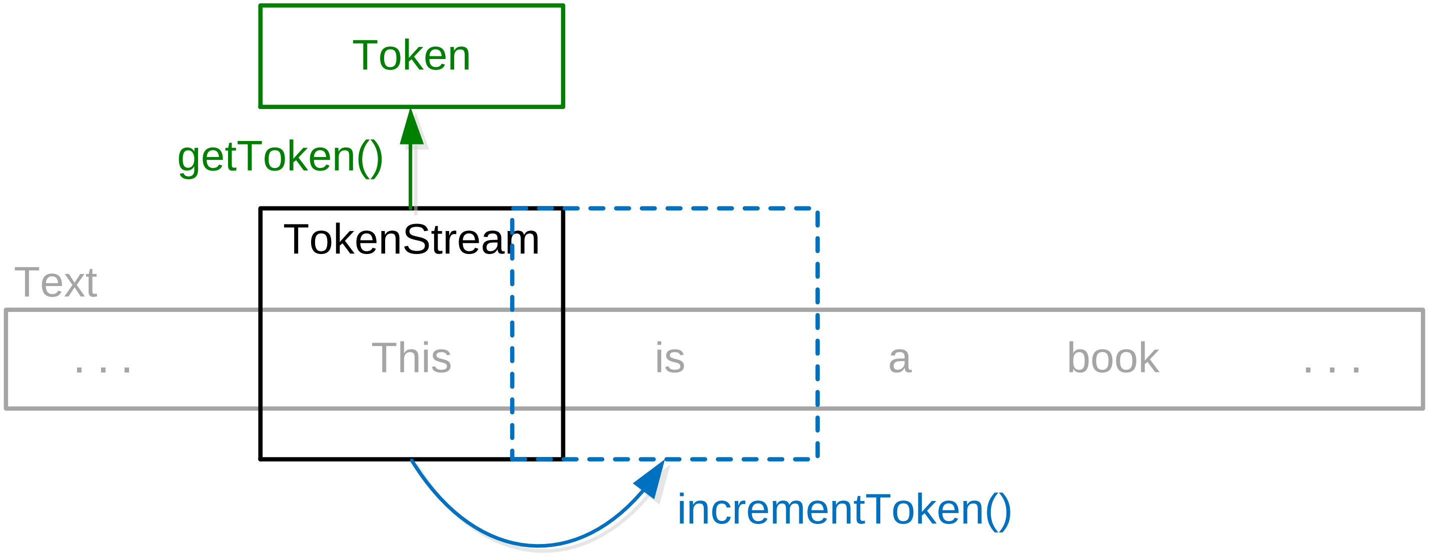
UML



<<abstract>>
Analyzer
<<abstract>> + tokenStream(input : Reader) : TokenStream
Token
- termBuffer : char[]
+ getTermBuffer() : char[]
+ getTerm() : String
+ clear()
creates
uses (flyweight pattern)
<<abstract>>
TokenStream
# token : Token
<<abstract>> + incrementToken() : boolean
+ getToken() : Token
uses (decorator pattern)
<<abstract>>
Tokenizer
# input : Reader
+ close()
<<abstract>>
TokenFilter
# input : TokenStream
+ close()

Core Classes - TokenStream

- Iterates tokens (likes a java.lang.Iterator)
- Call **incrementToken()** (like next()) to move to the next term
- When **getToken()** is called, returns an Token

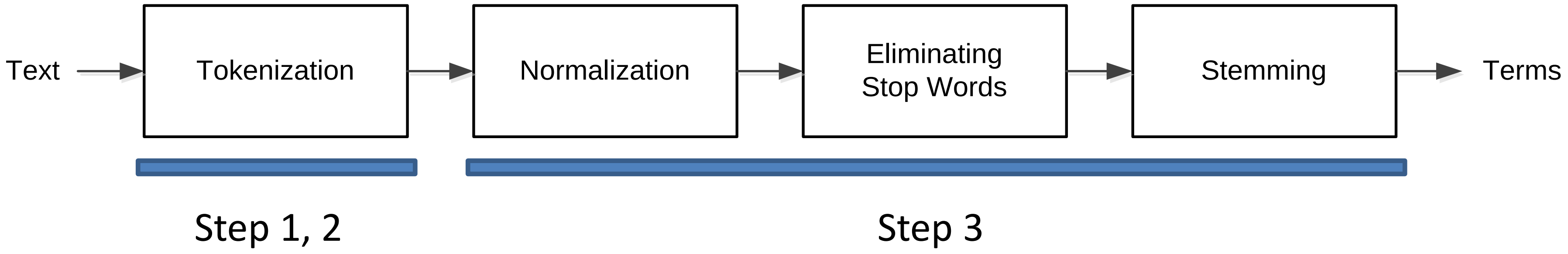


The Flyweight Pattern

- We know that a text can be large
  - There can be many, many tokens
  - This implies that we will create many, many Token instances
- It costs a lot of memory!
- **Flyweight pattern**
  - Prevents excessive object creations by reusing the same "flyweight" objects
- TokenStream **reuses the same Token instance** to carry different terms during the iteration
  - The getToken() always returns the same instance
  - A care must be taken not to create new objects inside this token instance

Core Classes - TokenStream

- Each time when **incrementToken()** is called:
  1. Reads the beginning chars of the (remaining) text
  2. Assembles chars as a term
  3. Post-processes the term (e.g., lowers case, eliminates it if it is a stop word, performs stemming, etc.). If the term is eliminated, repeats steps 1-3; otherwise, stops reading



Core Classes

- Token
  - Carries the current term
- Tokenizer
  - A TokenStream that performs the steps 1 and 2
  - Works at **char** level
- TokenFilter
  - A TokenStream that performs step 3
  - Works at **word** level

We can implement different Tokenizers and TokenFilters

- Tokenizer
  - E.g. WhiteSpaceTokenizer, LetterTokenizer, etc.
- TokenFilter
  - E.g. LowerCaseTokenFilter, StopFilter, StemFilter, etc.
- The Tokenizers and TokenFilters can be combined arbitrarily
- Remember something we have learnt?

The Decorator Pattern

- Accepts a reference to the preceding TokenStream implementation as the constructor of the succeeding class

```
public class StopFilter ... {
    public StopFilter(TokenStream input, ...) {
        this.input = input;
        ...
    }
}
```

- When create the instance, chain the Tokenizers and TokenFilters you want to use in a right order

```
new StemFilter(
       new StopFilter(
       new LowerCaseFilter(
       new LetterTokenizer(input))));
```

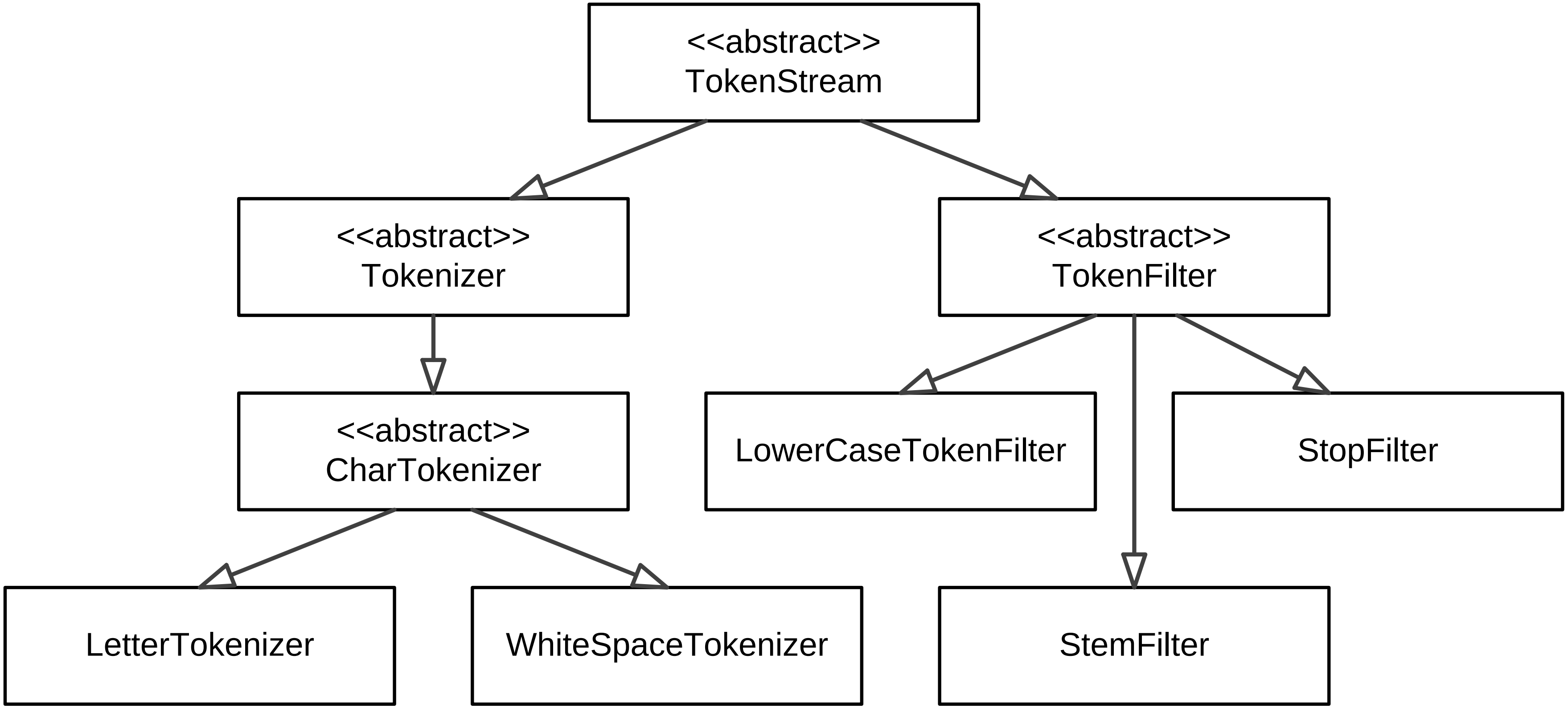
The Decorator Pattern

- When **incrementToken()** is called, every incrementToken() of preceding TokenStream is called. Every TokenStream will complete its mission (e.g. lowercase, remove stop words, etc.) in incrementToken()
- **Decorator pattern**
  - Separates concerns in different TokenStream implementations
  - But **allows dynamic chaining** of these implementations

Core Classes - Analyzer

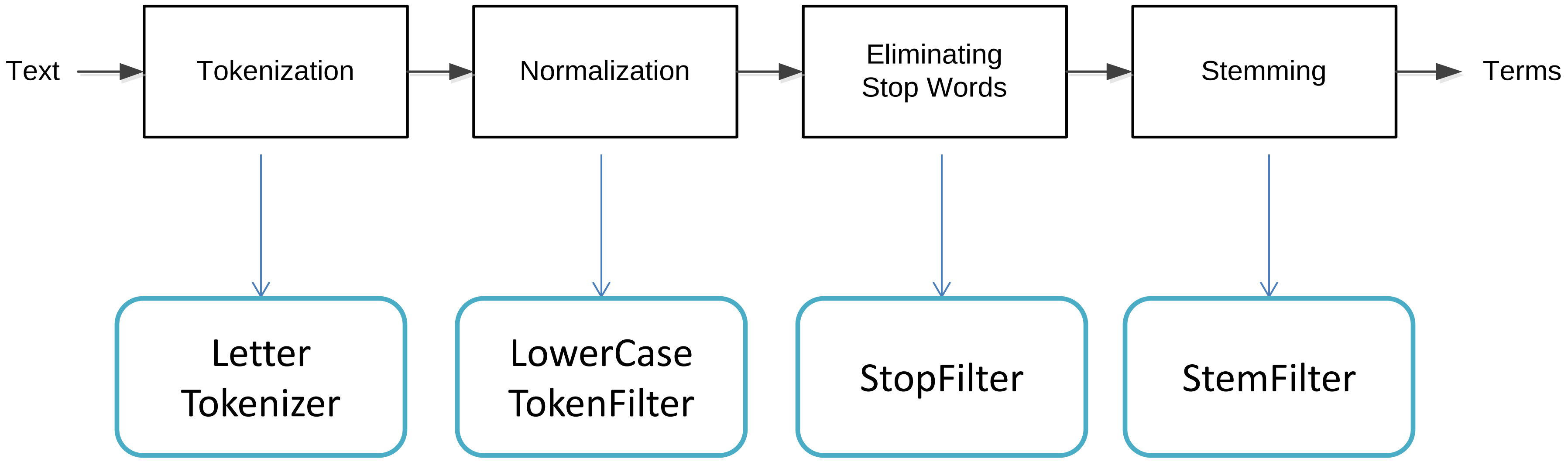
- Accepts the Reader of a text as input
    - A reader can load only the currently reading characters into memory
- It is the Analyzer that does the decorator chaining
    - You can write a specific Analyzer for specific texts without rewriting the TokenStream implementations
    - Add new TokenStream only when you need a new “step” in the analyzing process
    - When tokenStream() is called, creates an TokenStream instance (so users can use that instance to iterate tokens)
        - The instance can be “combined” from other TokenStream instances (can be of type Tokenizer or TokenFilter)

TokenStream Hierarchy



The Text Analyzer Framework

- Input: A text
- Output: terms
- Steps:



Today’s Mission

- We use the flyweight `Token` object to avoid excessive creation of `String` objects
- However, currently, the `BadLowerCaseFilter` and `BadStopFilter` will create a `String` object during each call to the `incrementToken()`
  - Create excessive `String` objects
- TODO:
  - Implement the `LowerCaseFilter` and `StopFilter` that avoid this problem
  - Let `DefAnalyzer` use the `LowerCaseFilter` and `StopFilter`

Hints

- The main method used today is in `DefAnalyzerApp` in recommender package
- In the `LowerCaseFilter`, try get `char[]` from the `Token` object, and make each char lowercase
- In the `StopFilter`, try to use the `CharArraySet.contains()` to determine if the `Token` object contains a stop word