Design (Chap -13)

Purpose: to define a program structure consisting of a number of modules that can be implemented independently and that, when implemented, will together satisfy the functional and performance requirements.

Design Goals:

- Meet functional and performance requirements

- Define a modular structure such that
  - the compts. Are all good abstractions
  - the structure is reasonably simple and relatively easy to implement and modify
  - the structure makes it relatively easy to incorporate the modification identified by the requirements analysis

Design Process:

- Select a target abstraction whose implementation has not yet been studied

- Identify helper abstractions that would be helpful in implementing the target and that facilitate decomposition of the problem

- Sketch how the helpers will be used in implementing the target

- Iterate until the implementation of all abstractions have been studied.

Design Notebook:

- The module dependency diagram

- A section for each abstraction containing:
  - its specification
  - its performance requirements
  - a sketch of its implementation
  - other info including justification of design decisions and discussion of alternatives, potential extensions or other modifications and info about expected context of use

Module dependency diagram:

- Nodes: represent abstractions such as data abstractions, procedure abstractions..

- Two kinds of arcs:
  - using arc
  - extension arc

- Useful for determining potential impacts of a change

Structure of Interactive Programs:

To illustrate the design process:

Consider a simple interactive search engine example that allows the user to run queries against a collection of documents.

Search Engine Example (contd.)

Requirements:

Normal case behavior: user starts the session with the search engine. User identify some documents of interest by presenting a URL of site containing documents; the engine will run searches against all those docs.

Also support for multisite searches, user can present additional URLs. Can add docs at any time, not just start of session.

Search Engine Example (contd.)

Requirements (contd.):

- User should be able to search the collection for a document with a particular title.

- User should be able to run queries against collection.
  - Query: single word (keyword)

  - "and", "the", … uninteresting keywords. Engine should know the uninteresting keywords from a file or some storage..

Search Engine Example (contd.)

Requirements (contd.):

- User should be able to search the collection for a document with a particular title.

- User should be able to run queries against collection.
  - Query: single word (keyword)

  - "and", "the", … uninteresting keywords. Engine should know the uninteresting keywords from a file or some storage..

Search Engine Example (contd.)

Requirements (contd.):

- System responds to a query by presenting info about what docs contain the keyword. This info is ordered by how many times the keyword occurs in docs. It does not present actual docs but rather provides info s.t. user can examine the matching docs further if needed.

- Multiple keywords to be supported: return docs that contain all the keywords.

Search Engine Example (contd.)

Requirements (contd.):

- Performance issues: how to carry out the queries to do it efficiently;
  - Data structures to speed up the computation
  - Querying requires visiting the web sites containing the docs.

Search Engine Example (contd.)

Requirements (contd.):

- Tracking modifications desired for future release
- Documents must be saved locally..
- Note: Trade-off betwen. speed of processing queries vs. space taken for storing docs.
- System errors: engine can fail if something goes wrong

Search Engine Example (contd.)

Requirements (contd.):

- User errors: user should be told about the error, e.g. entering uninteresting keyword, entering a word not in any doc..

Search Engine Example (contd.)

Design:

- Two parts: functional (FP) part, user interface (UI) part
- UI: interaction with user, display info to the user, accept user input
- FP: carries out user commands when provided with user input, notifies UI of the result

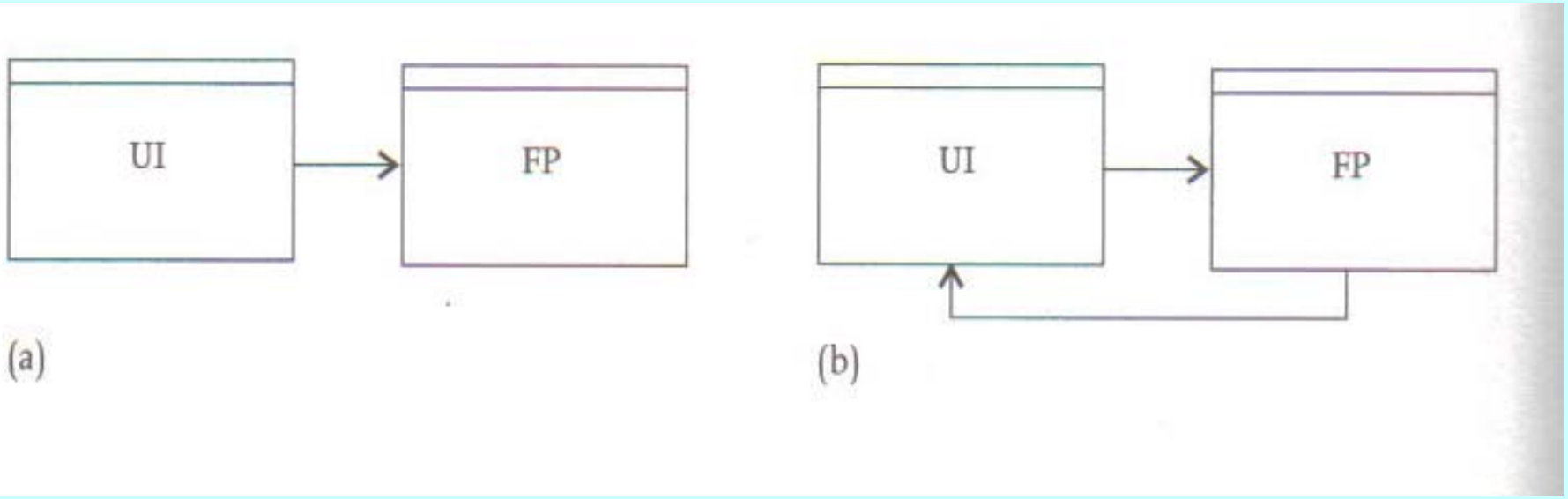
Search Engine Example (contd.)

Benefits of separating UI and FP:

- FP can be changed (to correct an error or improve performance) without having to change the UI
- UI can be changed (to provide different look and feel) without having to change the FP
- Several different UIs for same FP, including a UI that acts a driver for the FP. This allows thorough regression testing of FP

Search Engine Example (contd.)

Two ways to connect UI and FP:



We start with first structure…

Search Engine Example (contd.)

Design (contd.):

- UI design: beyond scope of this example.

- FP design: indpt. of any particular UI. Focus on specifying FP methods that are used by a UI to carry out requests.

Search Engine Example (contd.)

Specification of class Engine (our FP):

```
class Engine {
    // OVERVIEW: An engine has a state as described in the search engine
    //    data model. The methods throw the NotPossibleException
    //    when there is a problem; the exception contains a string explaining
    //    the problem. All instance methods modify the state of this.

    // constructors
    Engine( ) throws NotPossibleException
        // EFFECTS: If the uninteresting words cannot be retrieved from the
        //    persistent state throws NotPossibleException else creates NK and
        //    initializes the application state appropriately.

    // methods                                  → NOT
    Query queryFirst (String w) throws NotPossibleException
        // EFFECTS: If ¬WORD(w) or w in NK throws NotPossibleException else
        //    sets Key = { w }, performs the new query, and returns the result.
```

Search Engine Example (contd.)

Specification of class Engine (contd.)

```
Query queryMore (String w) throws NotPossibleException
    // EFFECTS: If ¬WORD(w) or w in NK or Key = { } or w in Key throws
    //    NotPossibleException else adds w to Key and returns the query result.

Doc findDoc (String t) throws NotPossibleException
    // EFFECTS: If t not in Title throws NotPossibleException
    //    else returns the document with title t.

Query addDocs (String u) 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 the query result that includes any matching new documents.
}
```

Search Engine Example (contd.)

Engine class uses two other data abstractions:

- **Doc**: the way that UI gets hold of a document; to display a doc, it needs access to its title and text.

- **Query**: the way UI gets hold of a query result; it needs access to query keywords, docs that match the query..

Search Engine Example (contd.)

Specification of class Doc:

```
class Doc {
    // OVERVIEW: A document contains a title and a text body.

    // methods
    String title ( )
        // EFFECTS: Returns the title of this.

    String body ( )
        // EFFECTS: Returns the body of this.
}
```

toString and repOK() should be added in all the specifications..

Search Engine Example (contd.)

Specification of class Query:

```
class Query {
    // OVERVIEW: Provides information about the keywords of a query and
    //    the documents that match those keywords. size returns the number
    //    of matches. Documents can be accessed using indexes between 0 and
    //    size. Documents are ordered by the number of matches they
    //    contain, with document 0 containing the most matches.

    // methods
    String[ ] keys ( )
        // EFFECTS: Returns the keywords of this.

    int size ( )
        // EFFECTS: Returns a count of the documents that match the query.

    Doc fetch (int i) throws IndexOutOfBoundsException
        // EFFECTS: If 0 <= i < size returns the ith matching document else
        //    throws IndexOutOfBoundsException.
}
```

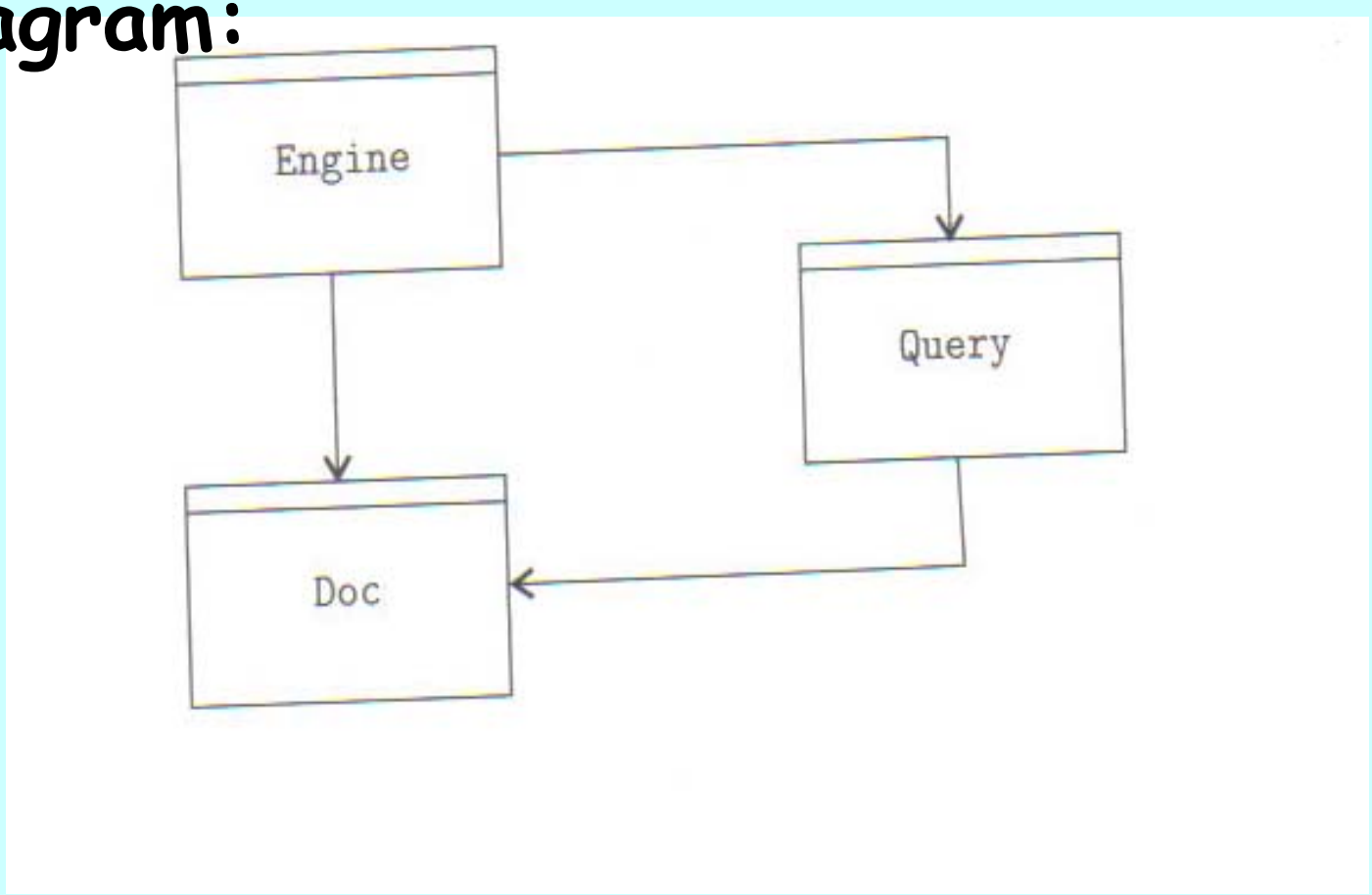
Search Engine Example (contd.)

Starting the Design:

- Create the design notebook:
  - Construct an initial module dependency diagram
  - Specifications for the data abstractions

Search Engine Example (contd.)

First step: Initial Module dependency diagram:



Search Engine Example (contd.)

Choose first target, the Engine class (FP):

- Invent helpers

To do this:

- make a list of tasks that must be accomplished. For Engine class, each method is a task, consider each of them in turn..

Search Engine Example (contd.)

List of tasks for queryFirst method:

1. Check the input string w to be sure it is a word
2. Make sure it is an interesting word
3. Start a new query with w as the only keyword

Search Engine Example (contd.)

List of tasks for queryFirst method:

4. For each doc, determine whether it is a match (i.e. contains w)
5. For each match, determine the number of occurrences of w
6. Sort the matches by number of occurrences of w
7. Return the info about the matches and query

Search Engine Example (contd.)

- Need for a data abstraction **WordTable** that keeps track of all words, both interesting and uninteresting words
- Also save the count of occurrences in the **WordTable**
- Need for another data abstraction **MatchSet** that performs tasks 3-6

Search Engine Example (contd.)

List of tasks for queryMore method:

Only difference with queryFirst is that we are now interested in documents that have already matched.

Search Engine Example (contd.)

List of tasks for findDoc method:

Need another data abstraction : TitleTable

Search Engine Example (contd.)

List of tasks for addDocs method:

1. Need to obtain the new docs from the site named by the URL
2. Call getDocs to get the documents one after another
3. Creates new document and adds it to the TitleTable and WordTable

Search Engine Example (contd.)

List of tasks for addDocs method:

4. Must add the document to the query if one is in progress
5. Return the new query result if one is in progess; otherwise empty query set.

Search Engine Example (contd.)

Second step: Document the abstractions

```
class Comm {
   static Iterator getDocs (String u) throws NotPossibleException
      // EFFECTS: If u isn't a legitimate URL or the site it names does not
      //    respond as expected throws NotPossibleException else returns a
      //    generator that will produce the documents from site u (as strings).
}
```

Search Engine Example (contd.)

```
class WordTable {
    // OVERVIEW: Keeps track of both interesting and uninteresting words.
    //    The uninteresting words are obtained from a private file. Records
    //    the number of times each interesting word occurs in each document.

    // constructors
    WordTable ( ) throws NotPossibleException
        // EFFECTS: If the file cannot be read throws NotPossibleException
        //    else initializes the table to contain all the words in the file
        //    as uninteresting words.

    // methods
    boolean isInteresting (String w)
        // EFFECTS: If w is null or a nonword or an uninteresting word
        //    returns false else returns true.

    void addDoc (Doc d)
        // REQUIRES: d is not null
        // MODIFIES: this
        // EFFECTS: Adds all interesting words of d to this with a count
        //    of their number of occurrences.
}
```

Search Engine Example (contd.)

```
class TitleTable {
// OVERVIEW: Keeps track of documents with their titles.

// constructors
TitleTable ( )
    // EFFECTS: Initializes this to be an empty table.

// methods
void addDoc (Doc d) throws DuplicateException
    // REQUIRES: d is not null
    // MODIFIES: this
    // EFFECTS: If a document with d's title is already in this throws
    //    DuplicateException else adds d with its title to this.

Doc lookup (String t) throws NotPossibleException
    // EFFECTS: If t is null or there is no document with title t in this
    //    throws NotPossibleException else returns the document with title t.
}
```

Search Engine Example (contd.)

MatchSet and Query: looks similar: no reason to have two separate abstractions

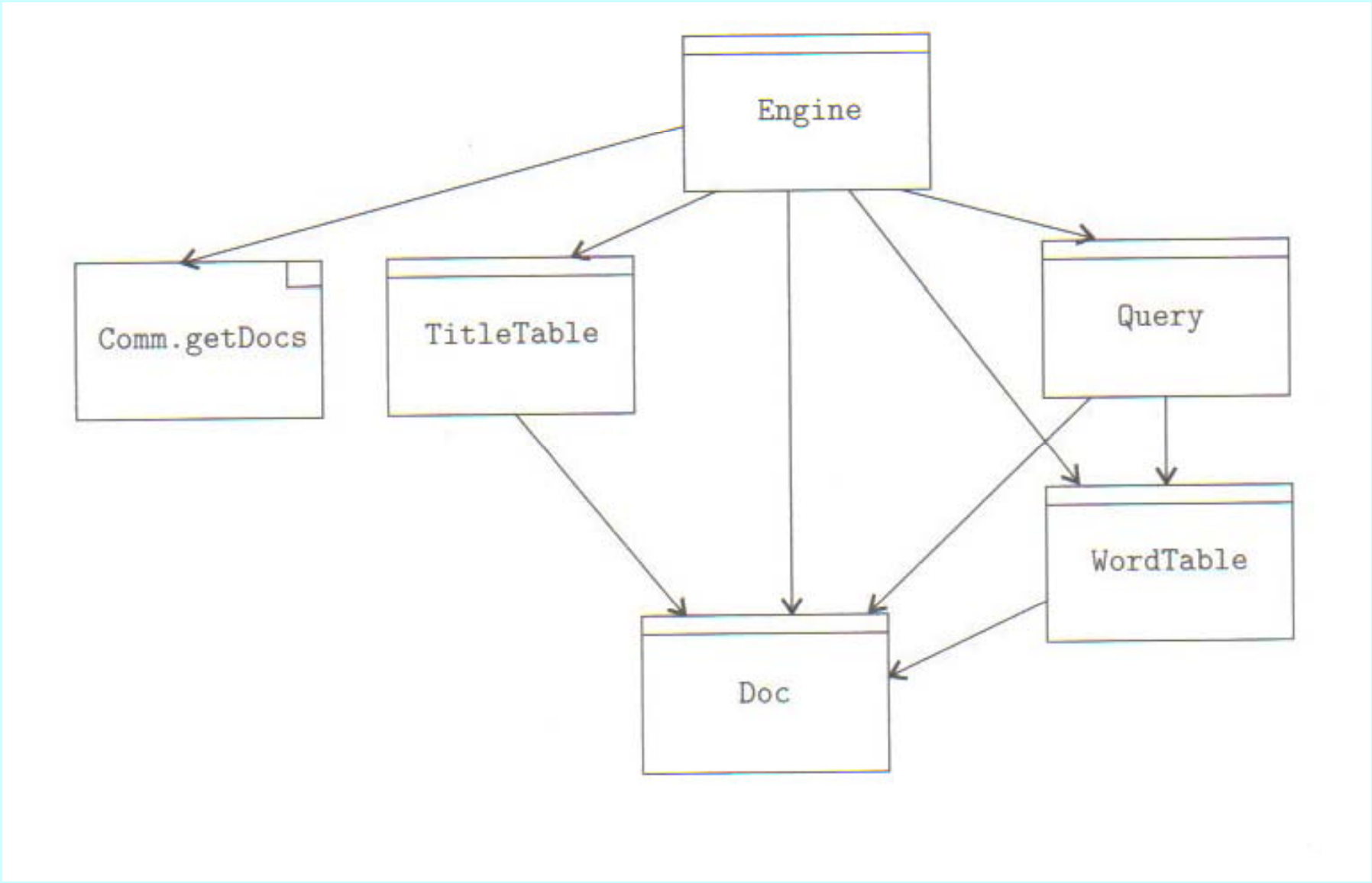
Merged into one data abstraction called Query

```
class Query {
    // OVERVIEW: as before plus

    // constructors
    Query ( )
        // EFFECTS: Returns the empty query.

    Query (WordTable wt, String w)
        // REQUIRES: wt and w are not null
        // EFFECTS: Makes a query for the single keyword w.

    // methods
    void addKey (String w) throws NotPossibleException
        // REQUIRES: w is not null
        // MODIFIES: this
        // EFFECTS: If this is empty or w is already a keyword in the query
        //     throws NotPossibleException else modifies this to contain the
        //     query for w and all keywords already in this.

    void addDoc (Doc d)
        // REQUIRES: d is not null
        // MODIFIES: this
        // EFFECTS: If this is not empty and d contains all the keywords of
        //     this adds it to this as a query result else does nothing.
}
```

```
class Doc {
    // OVERVIEW: As before plus

    // constructors
    Doc (String d) throws NotPossibleException
        // EFFECTS: If d cannot be processed as a document throws
        //    NotPossibleException else makes this be the Doc
        //    Corresponding to d.
}
```

Search Engine Example (contd.)

Extended Module dependency diagram:



The Design Method:

Basic approach is to let problem structure determine program structure:

1. List the tasks to be accomplished

2. Use the list to help invent abstractions, especially data abstractions to accomplish the tasks

The Design Method (contd.):

3. Introduce abstractions to hide details

4. Each abstraction should be focused on a single purpose

Continuing the Design

- **Selecting the next Target (*candidates*)**

  - abstractions whose implementation has not yet been studied but whose specification is complete

  - Choose target T among the candidates

Engine class is not a candidate (already designed its implementation)
Query class is a candidate
Doc and WordTable are not candidates: spec may not be complete

Query Abstraction

- Query has four interesting tasks to accomplish:
  - Compute a new query given a keyword
  - Extend a query with a new keyword (addKey method)
  - Extend a query with a new Document (addDoc method)
  - Provide access to information about the query.. Providing access to documents sorted by no. of occurrences of keywords

Must create an empty Query

Query Abstraction (contd.)

How to make a query given a keyword:

(1) find all documents that contain the keyword with its count

(2) Keep track of the keyword

(3) sort the documents based on the no. of occurences of keyword

Task 1. can be accomplished: lookup of WordTable

Task 3:

Query Abstraction (contd.)

Extend a query with a new keyword (Consider the addKey method):

(1) check whether the new key is already in use

(2) find the documents matching the new key

(3) Find the documents matching new key that are already in the query

(4) sort the remaining documents by the sums of matches

Task 2. can be accomplished: lookup of WordTable

Query Abstraction (contd.)

Extend a query with a new Document (addDoc method):

-check whether each of the keywords is in the new document

-add the documents to the matching documents in sorted order

Rep for Query:

WordTable k;

Vector matches; elements are DocCnt type

String[] keys;

DocCnt is a record-like type with two fields:
the document and the count of the occurrences of keywords

Sketches of some Query methods

For the constructor (of a non-empty query):

lookup the key in the WordTable

sort the matches using quickSort

Similarly for addKey and addDoc methods

Extended Module Dependency Diagram: