CS 50:
Software Design and Implementation

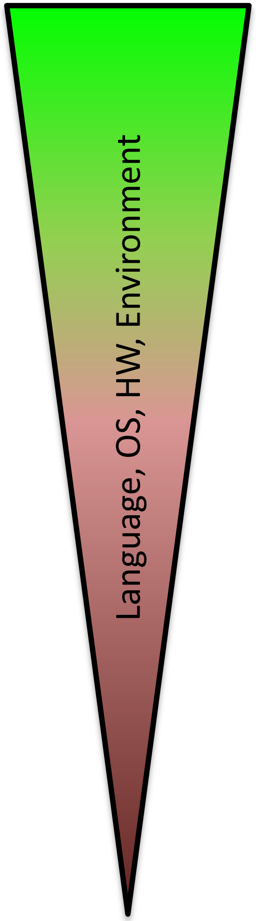
Software specifications

Software development overview

Customer

Developer

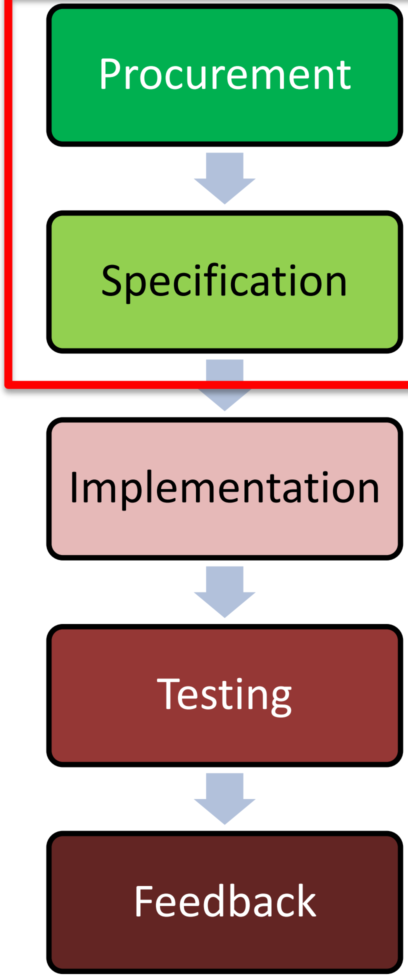
Today's focus



Independent

Language, OS, HW, Environment

Dependent



Procurement

- Common understanding of terms, customer's needs
- Regulatory, standards bodies, industry group needs

Specification

- The "shalls"
- Requirements spec: functionality, cost, performance goals
- Design spec: inputs/outputs, functional decomposition, pseudo code
- Implementation spec: define APIs, interfaces, test plan

Implementation

- Turn specifications into working code
  - Correctness
  - Clarity/Simplicity
  - Generality
  - Performance

Testing

- Unit testing
- Integration testing
- Regression testing
- Fuzz testing
- Acceptance testing

Feedback

- Demonstrate progress
- Incorporate customer feedback
- Avoid scope creep!

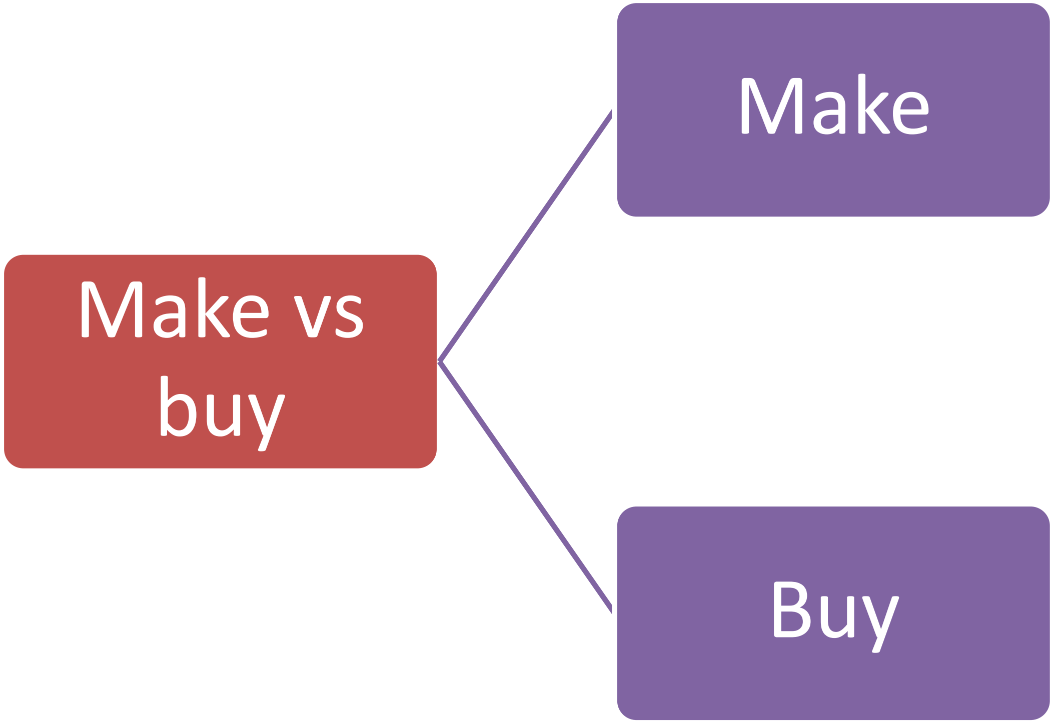
Agenda

1. Procurement

2. Specifications

3. Activity

Procurement overview



Make vs buy

Make

Buy

The first decision when a new software system is needed is to MAKE or BUY

MAKE

Pros

- Highly customizable
- Solves company's unique problem
- Own code base (what if external developer goes out of business?)
- Fits with other tools

Cons

- May be more expensive
- May need to hire employees with the right skill set

Which should you do?

No hard and fast rules!

I normally prefer to buy unless

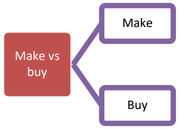
- **Company has unique need**
- **This software gives us a competitive advantage**

New software system

Code escrow:

- **Vendor puts code in repo where customer can get it if vendor goes out of business**
- **Good idea?**
- **Not practical for non-trivial systems**
- **Should you do it?**
- **YES! (be realistic about what you get)**

BUY



Pros

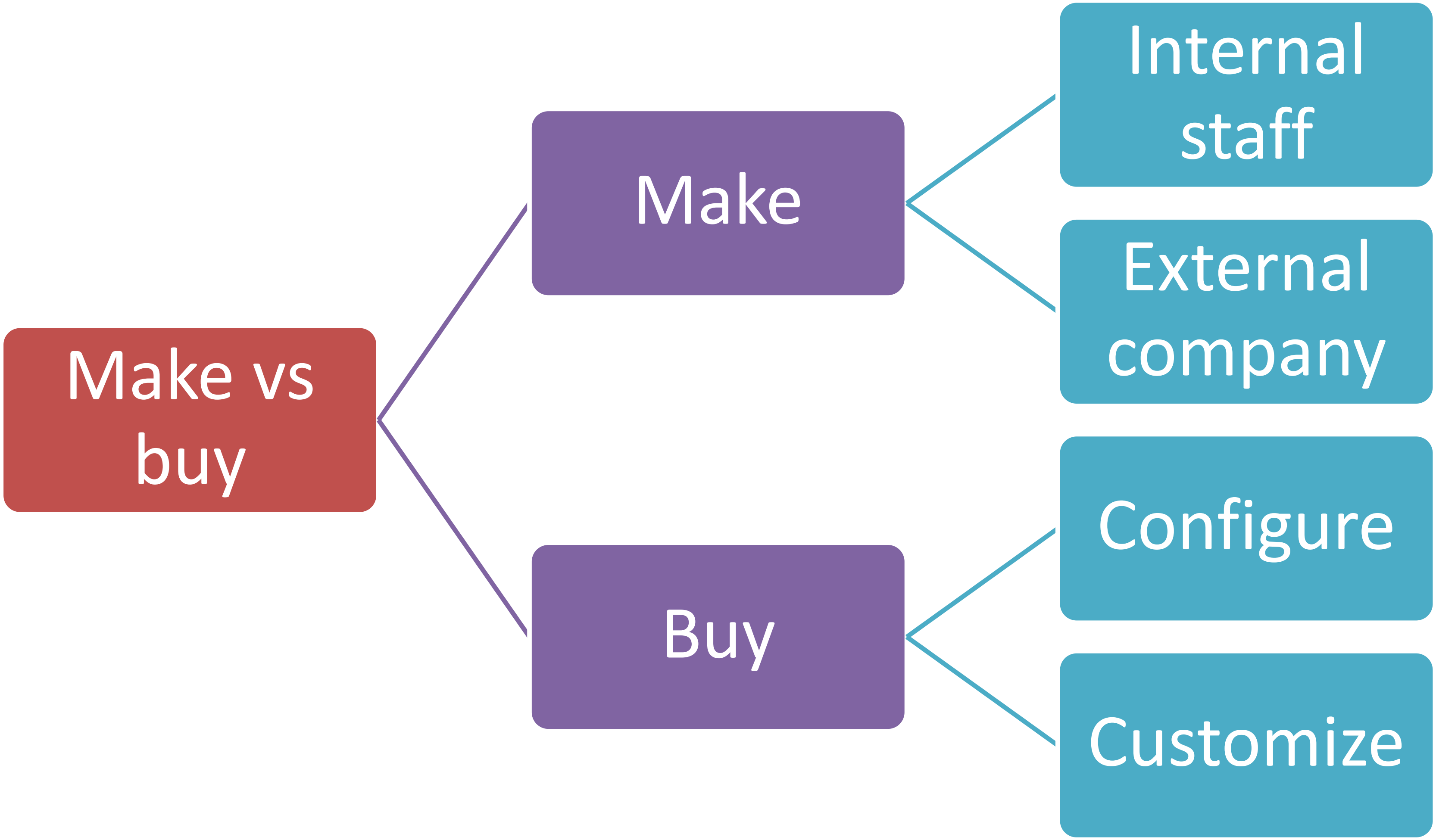
- Fast implementation time
- May have lower cost
- Vendor maintains software

Cons

- Code may belong to vendor (unless agreed otherwise)
- Get updates at vendor's pace
- Operational support may be unclear

Adapted from https://codeit.us/blog/build-vs-buy-software

Procurement overview



Make vs buy

Make

Buy

Internal staff

External company

Configure

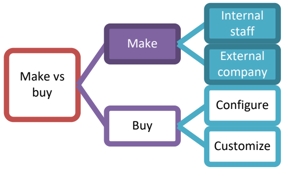
Customize

If decision is to MAKE, then the next decision is who should do the work

MAKE

Customer decides to MAKE
Step 1: choose developer

- In-house staff
- External developer



External development company

- **RFP**: Customers write detailed specs of what needs to be completed (government projects!)
- **RFI**: Specs are more open-ended, or customer may not yet be sure of specs, looking for approaches
- Companies write proposals on why they should do the work, competing to win contract
  - Tout extensive expertise in the industry or area of project
  - Normally estimate time and costs
- Customer ultimately selects a company to do the work after a “bake off”

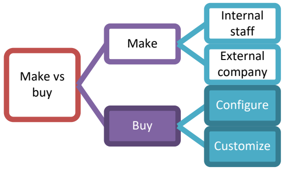
Internal staff

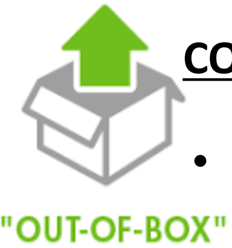
- Identify staff who will work on project
- Opportunity cost for other projects (lab time discussion)
- Identify any hardware or software that must be purchased

RFP: Request For Proposal
RFI: Request For Information

If the decision is to BUY, then the next question is to CONFIGURE or CUSTOMIZE

BUY





CONFIGURE

- Large systems (think ERP or CRM) can often be configured to a large degree
- Even so, there are limits to:
  - What the software can do
  - What the vendor is willing to do

Pros

- Cheaper and faster

Cons

- Often better off *changing business to fit software* than *changing software to fit business*

Which should you do? I prefer configuration when appropriate for business needs

CUSTOMIZE

Vendor creates extensions/modifications can be made to existing software to fit business needs

- Might be able to get license for internal developers to extend/modify system

Pros

- Get software that fits business

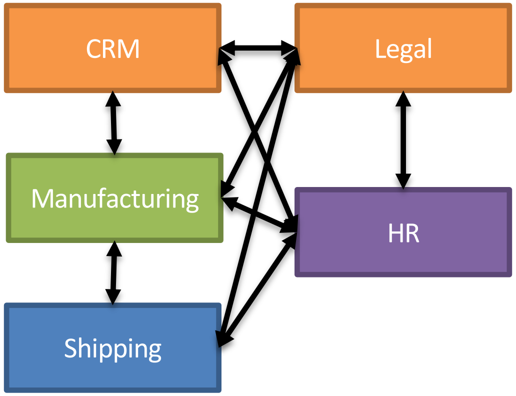
Cons

- Vendor support multiple versions
- Vendor may not build exactly what customer wants
- Customizations need to update when software updates

ERP = Enterprise Resource Planning
CRM = Customer Relationship Management

Best of breed components vs integrated system

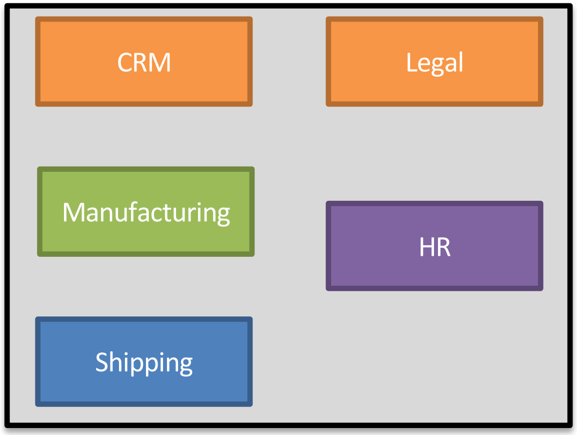
Best of breed



Which should you do?

- **I've done both**
- **I don't have a good answer**
- **My best advice: if it doesn't give you a competitive edge, *lean* toward integrated solution (slightly fewer operational headaches)**

Integrated solution



Pros

- Better targeted solutions – can choose products with rich functionality
- Easier for vendors to be the best in their area if they do one thing (hard to be the best at everything!)
- Access to the latest technology
- Modular application maintenance may reduce company disruptions

Cons

- Likely need to integrate components yourself
- Brittle – if components change, must re-integrate

Pros

- One vendor – "Jack of all trades" system with everything under one umbrella
- Components designed to work together
- Unified support – "one throat to choke"

Cons

- Probably not the best at anything if trying to do everything
- Tied to a single ecosystem – committing to one system and their technology roadmap for the entire company

Adapted from https://www.mapcom.com/blog/best-of-breed-vs-monolithic-systems-finding-the-best-software-solutions-philosophy/

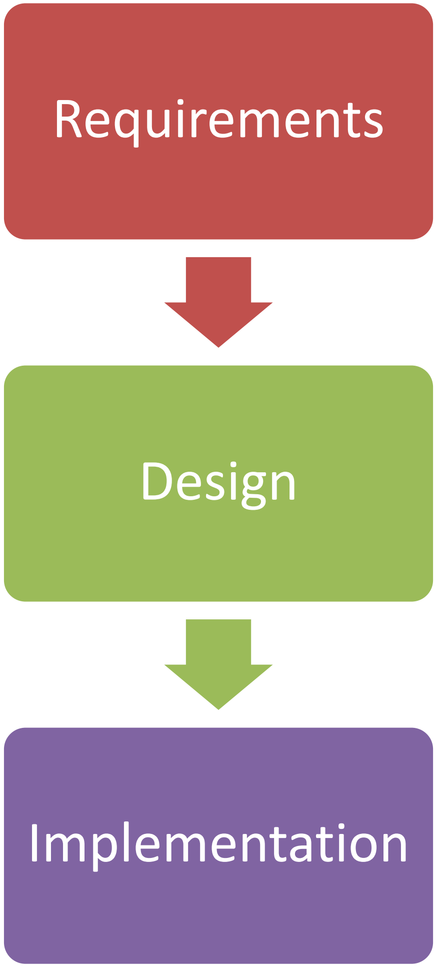
Agenda

1. Procurement

➡ 2. Specifications

3. Activity

Three specifications are commonly used to describe a software project

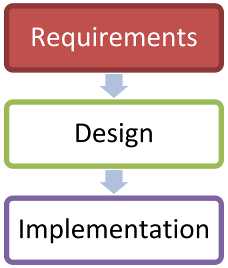


What needs to be done

What modules are needed and how do they fit together

Specifics of how modules will be implemented

Requirements specification attempts to capture what the customer wants built

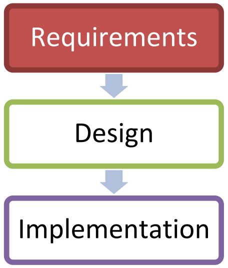


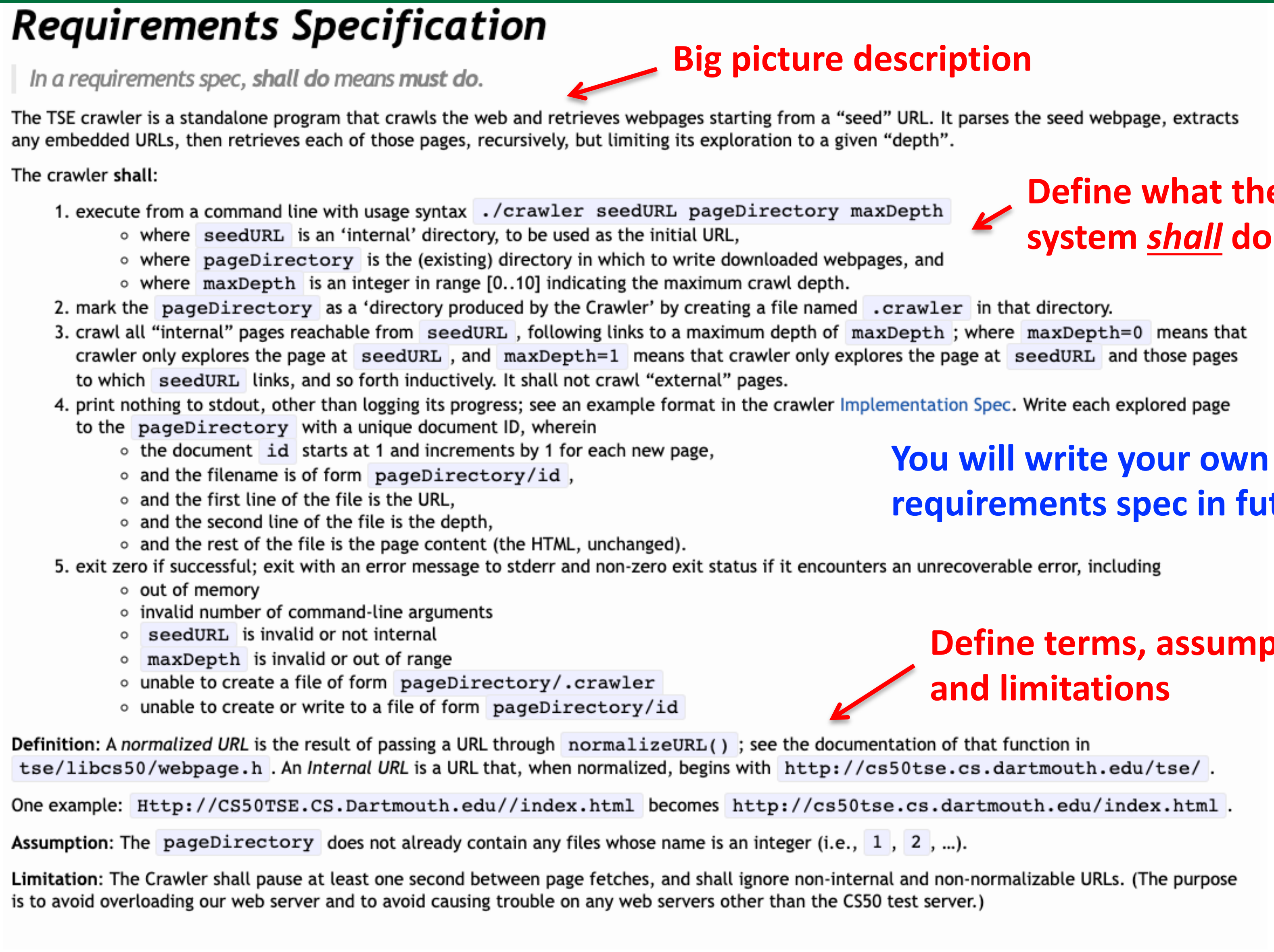
Requirements spec

- Goal: capture what the customer wants built
- Attempts to provide common understanding between customer and developer
- Defines terms, assumptions, and limitations
- *Shall* means *will*
- Often addresses:
  - Functionality - what should the system do?
  - Performance - goals for speed, size, energy efficiency, etc.
  - Compliance - with federal/state law or institutional policy
  - Compatibility - with standards or with existing systems
  - Security - against a specific threat model under certain trust assumptions
  - Cost – what should the system cost to build?
  - Timeline - when will various part of the system be completed? what are the deadlines?
  - Hardware/software - what hardware or software must be purchased or provisioned?
  - Personnel - who will work on this project (or at least what skills are required e.g., C programmer, database admin)?

Technique: have the customer say back to you what needs to be done

Crawler requirements spec gives big picture, “shall do”, and definitions





Requirements Specification

> *In a requirements spec,* ***shall do*** *means* ***must do****.*

The TSE crawler is a standalone program that crawls the web and retrieves webpages starting from a “seed” URL. It parses the seed webpage, extracts any embedded URLs, then retrieves each of those pages, recursively, but limiting its exploration to a given “depth”.

The crawler **shall**:

1. execute from a command line with usage syntax `./crawler seedURL pageDirectory maxDepth`
   - where `seedURL` is an ‘internal’ directory, to be used as the initial URL,
   - where `pageDirectory` is the (existing) directory in which to write downloaded webpages, and
   - where `maxDepth` is an integer in range [0..10] indicating the maximum crawl depth.
2. mark the `pageDirectory` as a ‘directory produced by the Crawler’ by creating a file named `.crawler` in that directory.
3. crawl all “internal” pages reachable from `seedURL`, following links to a maximum depth of `maxDepth`; where `maxDepth=0` means that crawler only explores the page at `seedURL`, and `maxDepth=1` means that crawler only explores the page at `seedURL` and those pages to which `seedURL` links, and so forth inductively. It shall not crawl “external” pages.
4. print nothing to stdout, other than logging its progress; see an example format in the crawler Implementation Spec. Write each explored page to the `pageDirectory` with a unique document ID, wherein
   - the document `id` starts at 1 and increments by 1 for each new page,
   - and the filename is of form `pageDirectory/id`,
   - and the first line of the file is the URL,
   - and the second line of the file is the depth,
   - and the rest of the file is the page content (the HTML, unchanged).
5. exit zero if successful; exit with an error message to stderr and non-zero exit status if it encounters an unrecoverable error, including
   - out of memory
   - invalid number of command-line arguments
   - `seedURL` is invalid or not internal
   - `maxDepth` is invalid or out of range
   - unable to create a file of form `pageDirectory/.crawler`
   - unable to create or write to a file of form `pageDirectory/id`

Definition: A *normalized URL* is the result of passing a URL through `normalizeURL()`; see the documentation of that function in `tse/libcs50/webpage.h`. An *Internal URL* is a URL that, when normalized, begins with `http://cs50tse.cs.dartmouth.edu/tse/`.

One example: `Http://CS50TSE.CS.Dartmouth.edu//index.html` becomes `http://cs50tse.cs.dartmouth.edu/index.html`.

Assumption: The `pageDirectory` does not already contain any files whose name is an integer (i.e., `1`, `2`, ...).

Limitation: The Crawler shall pause at least one second between page fetches, and shall ignore non-internal and non-normalizable URLs. (The purpose is to avoid overloading our web server and to avoid causing trouble on any web servers other than the CS50 test server.)

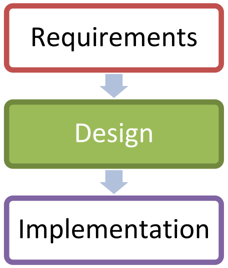
Big picture description

Define what the system *shall* do

You will write your own requirements spec in future labs

Define terms, assumptions, and limitations

The design specification describes subsystems in a hardware agnostic way


Requirements

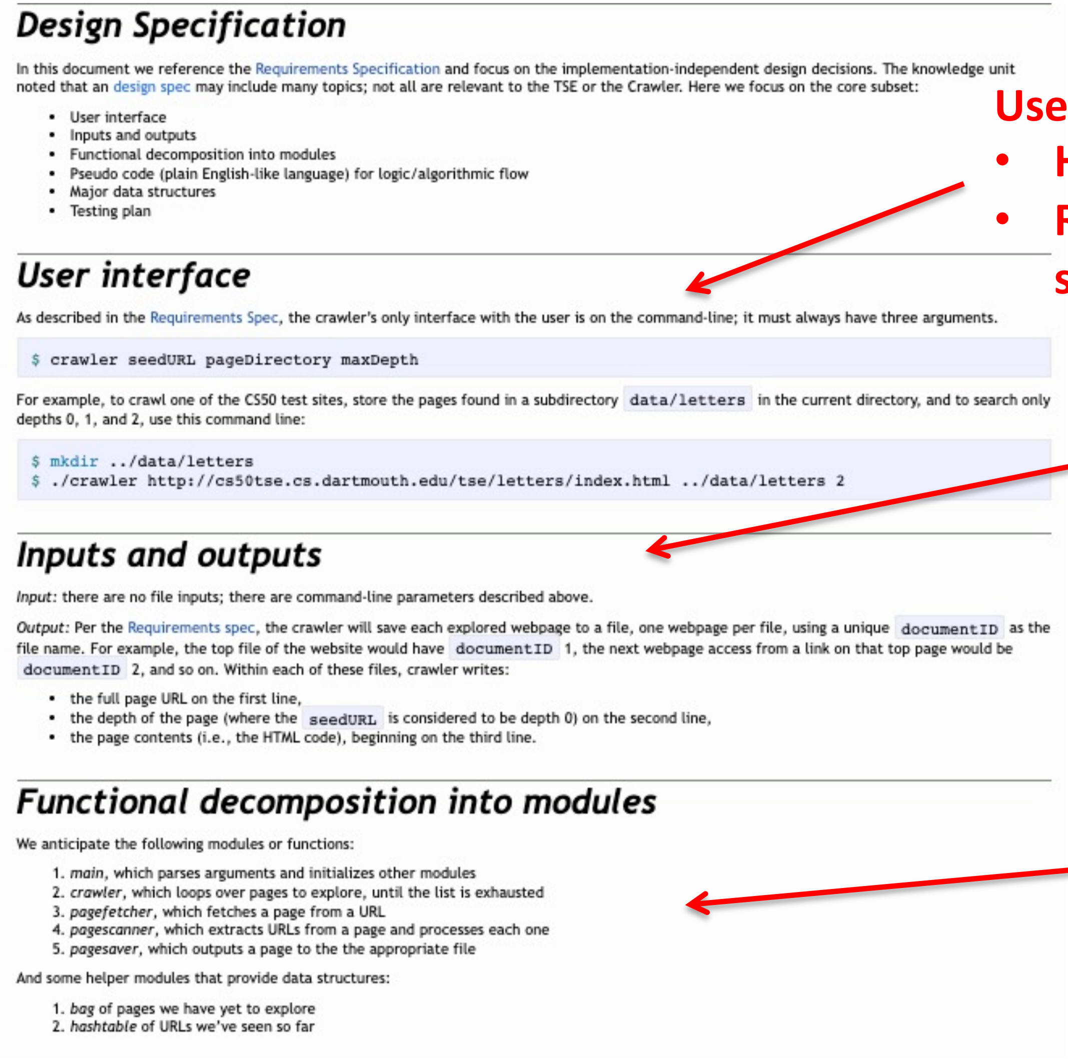
Design

Implementation

Design spec

- Goal: describe needed modules of solution
  - Hardware agnostic
  - Gives data flow through modules
- Often provides:
  - User interface
  - Inputs and outputs
  - Functional decomposition into modules
  - Pseudo code for logic/flow
  - Major data structures
  - Error handling and recovery
  - Testing plan

Crawler design spec



Design Specification

In this document we reference the Requirements Specification and focus on the implementation-independent design decisions. The knowledge unit noted that an design spec may include many topics; not all are relevant to the TSE or the Crawler. Here we focus on the core subset:

- User interface
- Inputs and outputs
- Functional decomposition into modules
- Pseudo code (plain English-like language) for logic/algorithmic flow
- Major data structures
- Testing plan

User interface

As described in the Requirements Spec, the crawler's only interface with the user is on the command-line; it must always have three arguments.

```
$ crawler seedURL pageDirectory maxDepth
```

For example, to crawl one of the CS50 test sites, store the pages found in a subdirectory `data/letters` in the current directory, and to search only depths 0, 1, and 2, use this command line:

```
$ mkdir ../data/letters
$ ./crawler http://cs50tse.cs.dartmouth.edu/tse/letters/index.html ../data/letters 2
```

Inputs and outputs

Input: there are no file inputs; there are command-line parameters described above.

Output: Per the Requirements spec, the crawler will save each explored webpage to a file, one webpage per file, using a unique `documentID` as the file name. For example, the top file of the website would have `documentID` 1, the next webpage access from a link on that top page would be `documentID` 2, and so on. Within each of these files, crawler writes:

- the full page URL on the first line,
- the depth of the page (where the `seedURL` is considered to be depth 0) on the second line,
- the page contents (i.e., the HTML code), beginning on the third line.

Functional decomposition into modules

We anticipate the following modules or functions:

1. *main*, which parses arguments and initializes other modules
2. *crawler*, which loops over pages to explore, until the list is exhausted
3. *pagefetcher*, which fetches a page from a URL
4. *pagescanner*, which extracts URLs from a page and processes each one
5. *pagesaver*, which outputs a page to the the appropriate file

And some helper modules that provide data structures:

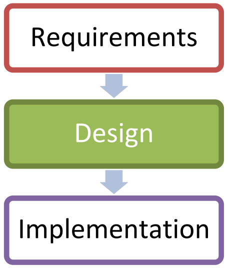
1. *bag* of pages we have yet to explore
2. *hashtable* of URLs we've seen so far

User interface

- **Here only command line**
- **Real project may include screen mockups**

Inputs and outputs

Modules needed

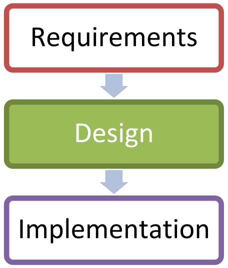


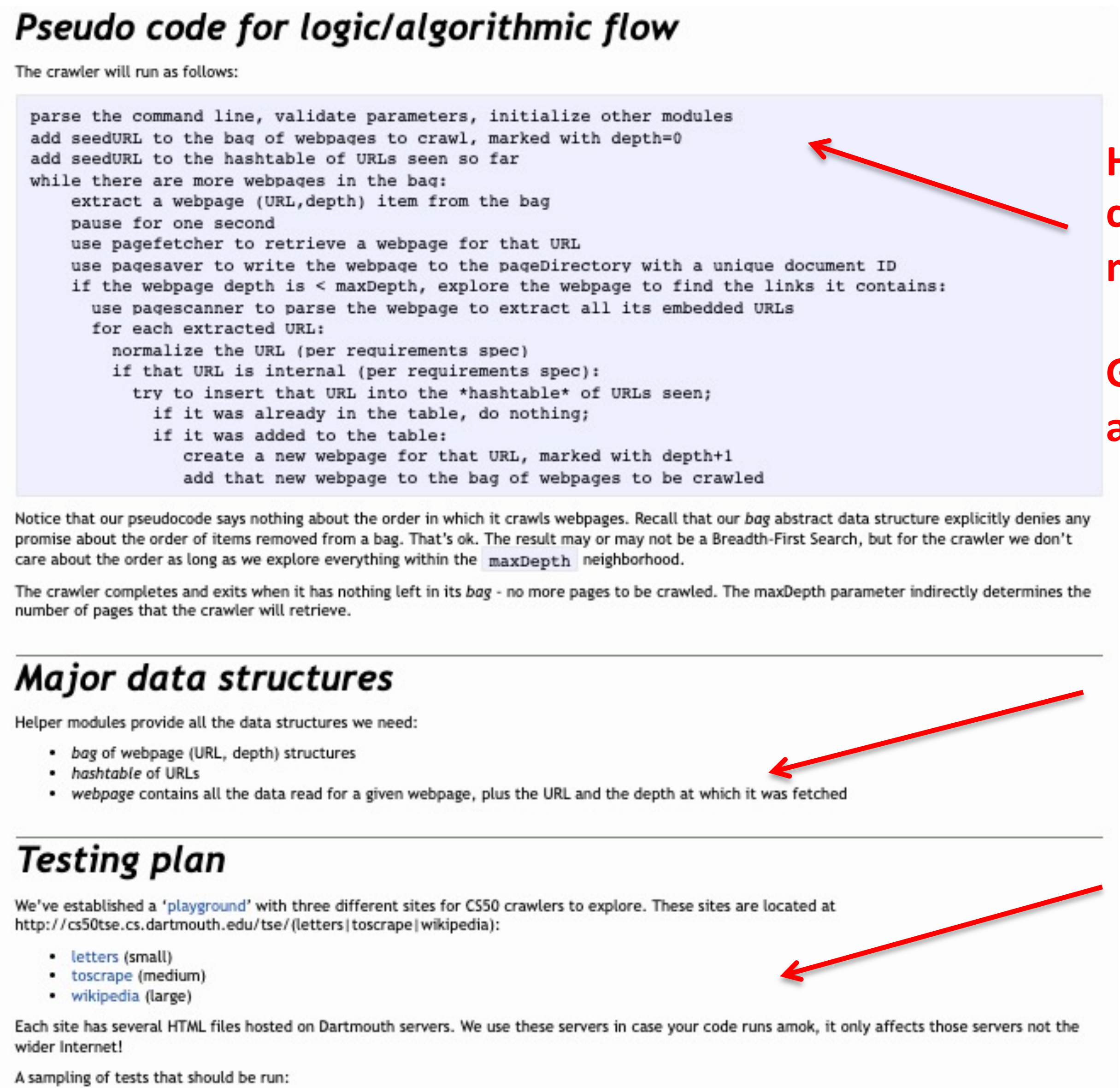
Requirements

Design

Implementation

Crawler design spec (continued)





Pseudo code for logic/algorithmic flow

The crawler will run as follows:

```
parse the command line, validate parameters, initialize other modules
add seedURL to the bag of webpages to crawl, marked with depth=0
add seedURL to the hashtable of URLs seen so far
while there are more webpages in the bag:
    extract a webpage (URL,depth) item from the bag
    pause for one second
    use pagefetcher to retrieve a webpage for that URL
    use pagesaver to write the webpage to the pageDirectory with a unique document ID
    if the webpage depth is < maxDepth, explore the webpage to find the links it contains:
      use pagescanner to parse the webpage to extract all its embedded URLs
      for each extracted URL:
        normalize the URL (per requirements spec)
        if that URL is internal (per requirements spec):
          try to insert that URL into the *hashtable* of URLs seen;
            if it was already in the table, do nothing;
            if it was added to the table:
               create a new webpage for that URL, marked with depth+1
               add that new webpage to the bag of webpages to be crawled
```

Notice that our pseudocode says nothing about the order in which it crawls webpages. Recall that our *bag* abstract data structure explicitly denies any promise about the order of items removed from a bag. That's ok. The result may or may not be a Breadth-First Search, but for the crawler we don't care about the order as long as we explore everything within the `maxDepth` neighborhood.

The crawler completes and exits when it has nothing left in its *bag* - no more pages to be crawled. The maxDepth parameter indirectly determines the number of pages that the crawler will retrieve.

Major data structures

Helper modules provide all the data structures we need:

- *bag* of webpage (URL, depth) structures
- *hashtable* of URLs
- *webpage* contains all the data read for a given webpage, plus the URL and the depth at which it was fetched

Testing plan

We've established a 'playground' with three different sites for CS50 crawlers to explore. These sites are located at http://cs50tse.cs.dartmouth.edu/tse/(letters|toscrape|wikipedia):

- letters (small)
- toscrape (medium)
- wikipedia (large)

Each site has several HTML files hosted on Dartmouth servers. We use these servers in case your code runs amok, it only affects those servers not the wider Internet!

A sampling of tests that should be run:

High-level pseudo code for stitching modules together

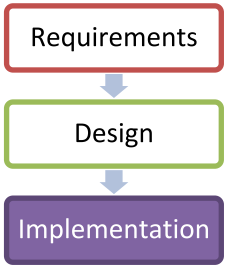
Generally not algorithm specific

Major data structurers
For crawler we will use Lab 3

Test plan

- **How will you know if the code works are expected?**

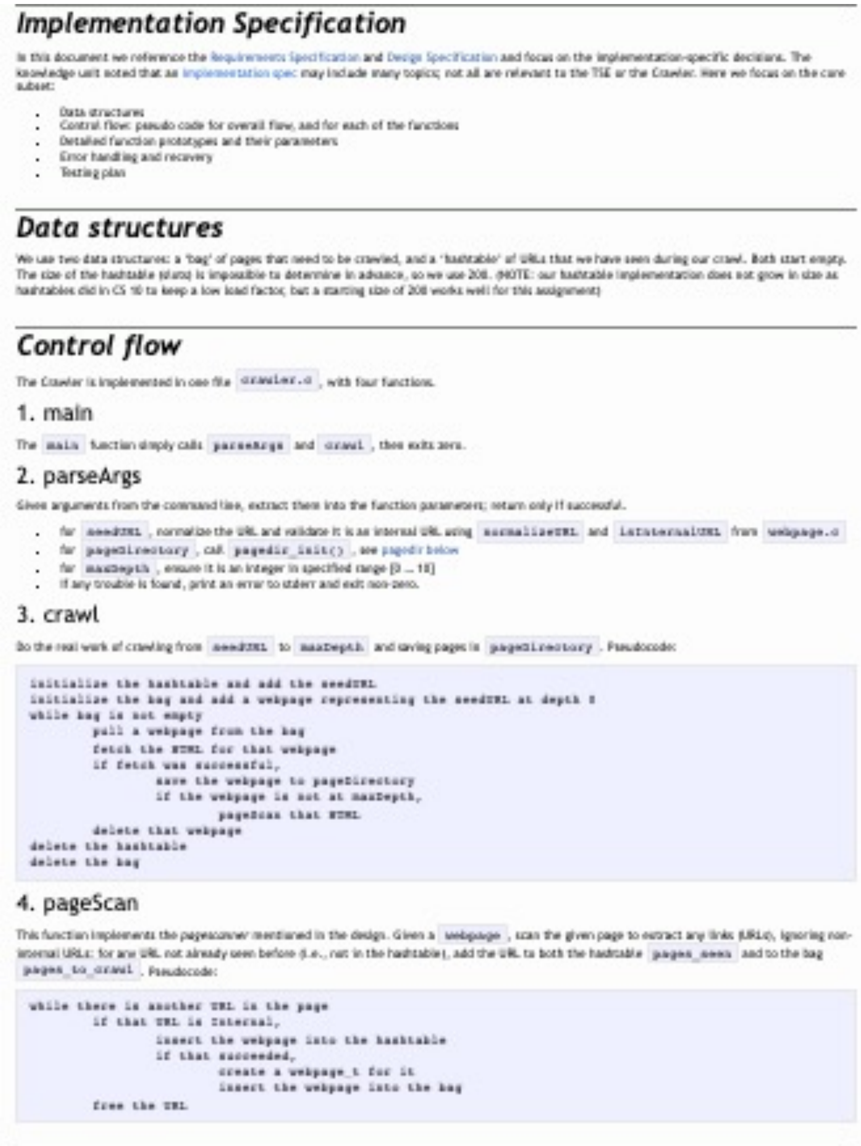
The implementation specification describes how modules will be implemented



Implementation spec

- Goal: describe modules in language, operating system, and hardware dependent manner
- Often includes:
  - Detailed pseudo code for each of the objects, components, and functions
  - Definition of detailed APIs, interfaces, function prototypes and their parameters
  - Data structures (e.g., struct names and members),
  - Security and privacy properties
  - Error handling and recovery
  - Resource management
  - Persistent storage (files, database, etc).
  - Detailed testing plan

Crawler implement spec



Implementation Specification

In this document we reference the Requirements Specification and Design Specification and focus on the implementation-specific decisions. The knowledge unit noted that an implementation spec may include many topics; not all are relevant to the TSE or the Crawler. Here we focus on the core subset:

- Data structures
- Control flow: pseudo code for overall flow, and for each of the functions
- Detailed function prototypes and their parameters
- Error handling and recovery
- Testing plan

Data structures

We use two data structures: a 'bag' of pages that need to be crawled, and a 'hashtable' of URLs that we have seen during our crawl. Both start empty. The size of the hashtable (slots) is impossible to determine in advance, so we use 200. (NOTE: our hashtable implementation does not grow in size as hashtables did in CS 10 to keep a low load factor, but a starting size of 200 works well for this assignment)

Control flow

The Crawler is implemented in one file `crawler.c`, with four functions.

1. main

The `main` function simply calls `parseArgs` and `crawl`, then exits zero.

2. parseArgs

Given arguments from the command line, extract them into the function parameters; return only if successful.

- for `seedURL`, normalize the URL and validate it is an internal URL using `normalizeURL` and `isInternalURL` from `webpage.c`
- for `pageDirectory`, call `pagedir_init()`, see pagedir below
- for `maxDepth`, ensure it is an integer in specified range [0 ... 10]
- If any trouble is found, print an error to stderr and exit non-zero.

3. crawl

Do the real work of crawling from `seedURL` to `maxDepth` and saving pages in `pageDirectory`. Pseudocode:

```
initialize the hashtable and add the seedURL
initialize the bag and add a webpage representing the seedURL at depth 0
while bag is not empty
        pull a webpage from the bag
        fetch the HTML for that webpage
        if fetch was successful,
                save the webpage to pageDirectory
                if the webpage is not at maxDepth,
                        pageScan that HTML
        delete that webpage
delete the hashtable
delete the bag
```

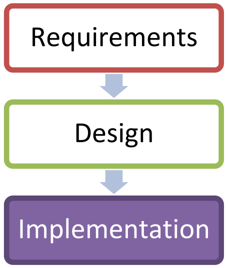
4. pageScan

This function implements the pagescanner mentioned in the design. Given a `webpage`, scan the given page to extract any links (URLs), ignoring non-internal URLs; for any URL not already seen before (i.e., not in the hashtable), add the URL to both the hashtable `pages_seen` and to the bag `pages_to_crawl`. Pseudocode:

```
while there is another URL in the page
        if that URL is Internal,
                insert the webpage into the hashtable
                if that succeeded,
                        create a webpage_t for it
                        insert the webpage into the bag
        free the URL
```

Data structures
(here from Lab 3)

Details on individual functions

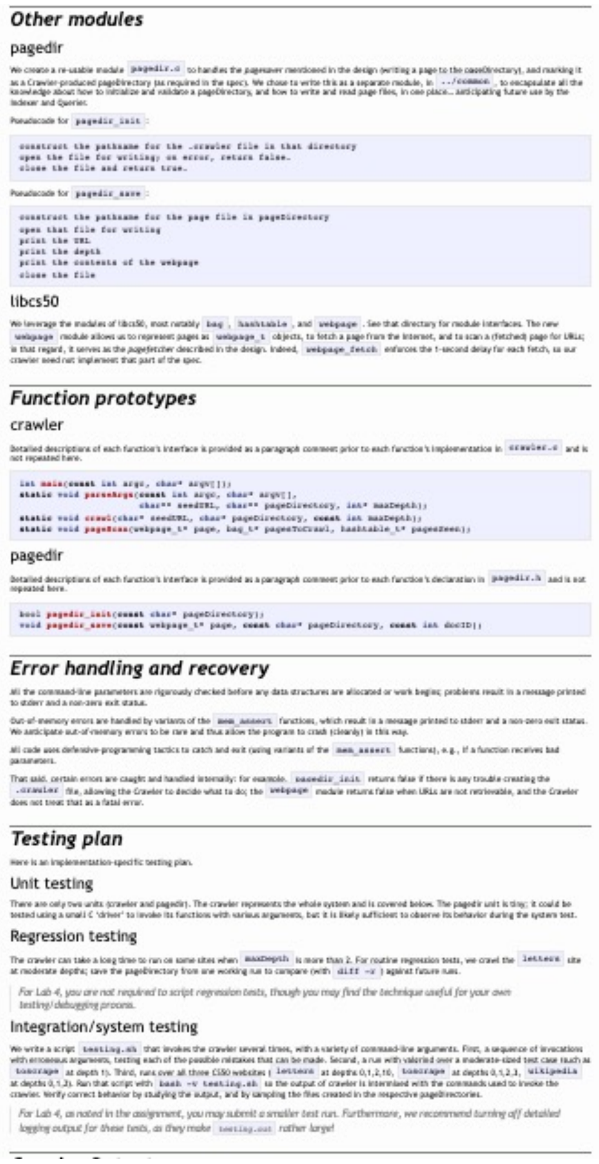


Requirements

Design

Implementation

Crawler implement spec (continued)



Other modules

pagedir

We create a re-usable module `pagedir.c` to handles the pagesaver mentioned in the design (writing a page to the pageDirectory), and marking it as a Crawler-produced pageDirectory (as required in the spec). We chose to write this as a separate module, in `../common`, to encapsulate all the knowledge about how to initialize and validate a pageDirectory, and how to write and read page files, in one place... anticipating future use by the Indexer and Querier.

Pseudocode for `pagedir_init`:

```
construct the pathname for the .crawler file in that directory
open the file for writing; on error, return false.
close the file and return true.
```

Pseudocode for `pagedir_save`:

```
construct the pathname for the page file in pageDirectory
open that file for writing
print the URL
print the depth
print the contents of the webpage
close the file
```

libcs50

We leverage the modules of libcs50, most notably `bag`, `hashtable`, and `webpage`. See that directory for module interfaces. The new `webpage` module allows us to represent pages as `webpage_t` objects, to fetch a page from the Internet, and to scan a (fetched) page for URLs; in that regard, it serves as the *pagefetcher* described in the design. Indeed, `webpage_fetch` enforces the 1-second delay for each fetch, so our crawler need not implement that part of the spec.

Function prototypes

crawler

Detailed descriptions of each function's interface is provided as a paragraph comment prior to each function's implementation in `crawler.c` and is not repeated here.

```
int main(const int argc, char* argv[]);
static void parseArgs(const int argc, char* argv[],
                      char** seedURL, char** pageDirectory, int* maxDepth);
static void crawl(char* seedURL, char* pageDirectory, const int maxDepth);
static void pageScan(webpage_t* page, bag_t* pagesToCrawl, hashtable_t* pagesSeen);
```

pagedir

Detailed descriptions of each function's interface is provided as a paragraph comment prior to each function's declaration in `pagedir.h` and is not repeated here.

```
bool pagedir_init(const char* pageDirectory);
void pagedir_save(const webpage_t* page, const char* pageDirectory, const int docID);
```

Error handling and recovery

All the command-line parameters are rigorously checked before any data structures are allocated or work begins; problems result in a message printed to stderr and a non-zero exit status.

Out-of-memory errors are handled by variants of the `mem_assert` functions, which result in a message printed to stderr and a non-zero exit status. We anticipate out-of-memory errors to be rare and thus allow the program to crash (cleanly) in this way.

All code uses defensive-programming tactics to catch and exit (using variants of the `mem_assert` functions), e.g., if a function receives bad parameters.

That said, certain errors are caught and handled internally: for example, `pagedir_init` returns false if there is any trouble creating the `.crawler` file, allowing the Crawler to decide what to do; the `webpage` module returns false when URLs are not retrievable, and the Crawler does not treat that as a fatal error.

Testing plan

Here is an implementation-specific testing plan.

Unit testing

There are only two units (crawler and pagedir). The crawler represents the whole system and is covered below. The pagedir unit is tiny; it could be tested using a small C 'driver' to invoke its functions with various arguments, but it is likely sufficient to observe its behavior during the system test.

Regression testing

The crawler can take a long time to run on some sites when `maxDepth` is more than 2. For routine regression tests, we crawl the `letters` site at moderate depths; save the pageDirectory from one working run to compare (with `diff -r`) against future runs.

> *For Lab 4, you are not required to script regression tests, though you may find the technique useful for your own testing/debugging process.*

Integration/system testing

We write a script `testing.sh` that invokes the crawler several times, with a variety of command-line arguments. First, a sequence of invocations with erroneous arguments, testing each of the possible mistakes that can be made. Second, a run with valgrind over a moderate-sized test case (such as `toscrape` at depth 1). Third, runs over all three CS50 websites (`letters` at depths 0,1,2,10, `toscrape` at depths 0,1,2,3, `wikipedia` at depths 0,1,2). Run that script with `bash -v testing.sh` so the output of crawler is intermixed with the commands used to invoke the crawler. Verify correct behavior by studying the output, and by sampling the files created in the respective pageDirectories.

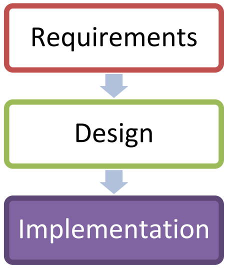
> *For Lab 4, as noted in the assignment, you may submit a smaller test run. Furthermore, we recommend turning off detailed logging output for these tests, as they make `testing.out` rather large!*

Details on individual functions

Function prototypes (e.g., an Interface in Java)

Error handling

Unit, regression, and integration test plan



Agenda

1. Procurement
2. Specifications
3. Activity