Agile software development

Demonstration on a case study

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

A simple, minimal example of Agile software development

- Showing several development cycles
- Explaining step by steps the decisions made

The case study : a word counting program, to help automatic text analysis

- *Input* $\Longrightarrow$ a text file
- No more informations !

The case study : a word counting program, to help automatic text analysis

- *Output* $\Longrightarrow$ a text file
- one line $\Longrightarrow$ word, no. occurrences

The case study : a word counting program, to help automatic text analysis

- *Requirement* $\Longrightarrow$ high-performance, portability
- large text files ie. 100 Mo, 1 Go, ...

Example

Input

```
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
```

Output

```
Never 6
gonna 6
give 1
you 5
up 1
let 1
down 1
run 1
...
```

Table of Contents

Design

Starting minimal : a file filter

- Command-line program
- Input file through standard input *stdin*
- Output file through standard input *stdout*
- Programmed in *C* and *shell*
- Use simple & stupid algorithm

Design

Example of usage of the filter

```
cat input.txt | wc > words-count.txt
```

- *cat* prints a file to the standard output
- *wc* is our filter program
- output of *wc-filter* redirected to a file

Design

We put our *wc-filter* in a shell script. User will see

```
wordcounter input.txt
```

Design justification

Why a file filter ?

- Using standard input & output $\Longrightarrow$ no file handling needed
- Reading from a network $\Longrightarrow$ OS does it for us
- Multi-threaded buffered IO $\Longrightarrow$ OS does it for us

Design justification

Why a file filter ?

- Using pipes to combine with others programs
- Easy to adapt to future, unknown requirements

Design justification

Need to read compressed input file ?

```
cat input.txt | wc > words-count.txt
```

becomes

```
zcat input.txt.gz | wc > words-count.txt
```

Design justification

Why using *C* ?

- Meeting the high-performance requirement

Code organization

A 4 characters story

- *Reader* $\Longrightarrow$ reads input
- *WordCounter* $\Longrightarrow$ word detection, storage & counting
- *StringBuffer* $\Longrightarrow$ "dynamic" *char* array
- *DynamicArray* $\Longrightarrow$ "dynamic" *void** array

The *Reader* interface

Just reads *stdin* character by character

```
typedef struct {
  ...
} Reader;

/* Reader initialization */
void Reader_init(Reader* self);

/* Reader termination */
void Reader_destroy(Reader* self);

/* Get the next char from input */
int Reader_next(Reader* self, char* out);

/* Check if there are more chars */
int Reader_hasNext(Reader* self);
```

The *WordCounter* interface

```c
typedef struct {
  char* str;
  unsigned long long count;
} Word;

typedef struct {
  DynamicArray words;
  StringBuffer buffer;
} WordCounter;

/* WordCounter initialization */
void WordCounter_init(WordCounter* self);

/* WordCounter termination */
void WordCounter_destroy(WordCounter* self);

/* Count words in an input */
int WordCounter_run(WordCounter* self, Reader* reader);

/* Print the list of words */
void WordCounter_print(WordCounter* self, FILE* out);
```

The *DynamicArray* interface

```
typedef struct {
  ...
} DynamicArray;

/* DynamicArray initialization */
void DynamicArray_init(DynamicArray* self);

/* DynamicArray termination */
void DynamicArray_destroy(DynamicArray* self);

/* Append one item to the array */
void DynamicArray_pushBack(DynamicArray* self, void* item);

/* Get an item from the array */
void* DynamicArray_at(DynamicArray* self, size_t index);

/* Current size of the array */
size_t DynamicArray_size(DynamicArray* self);
```

The *StringBuffer* interface

```
typedef struct {
  ...
} StringBuffer;

/* StringBuffer initialization */
void StringBuffer_init(StringBuffer* self);

/* StringBuffer termination */
void StringBuffer_destroy(StringBuffer* self);

/* Empty the StringBuffer */
void StringBuffer_clear(StringBuffer* self);

/* Append one char to the StringBuffer */
void StringBuffer_cat(StringBuffer* self, char c);

/* Pointer to the string hold by the StringBuffer */
char* StringBuffer_value(StringBuffer* self);

/* Current size of the StringBuffer */
size_t StringBuffer_size(StringBuffer* self);
```

Complexity

All those interfaces have minimal, low-complexity implementations

Reader

Reads *stdin* character by character

Complexity

All those interfaces have minimal, low-complexity implementations

StringBuffer, DynamicArray

Dynamically allocated array
Double the size of the array if not enough space

Complexity

All those interfaces have minimal, low-complexity implementations

WordCounter

For each word detected in the input

- Compare all previous words
- If a word is found, increment counter
- If no words found, add it

The *WordCounter_run* function

```
int WordCounter_run(WordCounter* self, Reader* reader) {
  char current;

  int insideWord = 0;
  while(Reader_hasNext(reader)) {
    if (!Reader_next(reader, &current))
      return 0;

    /* Current character is a space */
    if (isspace(current)) {
      if (insideWord) {
        insideWord = 0;
        WordCounter_countCurrentWord(self);
      }
    }
    /* Current character is NOT a space */
    else {
      if (!insideWord) {
        insideWord = 1;
        StringBuffer_clear(&(self->buffer));
      }

      StringBuffer_cat(&(self->buffer), current);
    }
  }
  if (insideWord)
    WordCounter_countCurrentWord(self);

  return 1;
}
```

The *main* function

Putting all together

```
int main(int argc, char* argv[]) {
  int ret;
  Reader reader;
  WordCounter wordCounter;

  /* Init */
  Reader_init(&reader);
  WordCounter_init(&wordCounter);

  /* Do the job */
  ret = WordCounter_run(&wordCounter, &reader);
  if (!ret)
    fprintf(stderr, "Error while processing input: %s\n", strerror(errno));
  else
    WordCounter_print(&wordCounter, stdout);

  /* Job done */
  WordCounter_destroy(&wordCounter);
  Reader_destroy(&reader);

  return ret ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Correctness

Checking if the words are properly counted

- a 50 lines *Python* script
- randomly generate a text with predetermined word counts
- allows billions of tests on enormous file size

Quality

Checking for the usual suspects with debugging tools

- Memory leaks
- Access to uninitialized memory

Speed

Using the *time* command, with real text files

file	size	process. time
Lancet article	424 Ko	0.66 sec.
Paradise Lost	476 Ko	1.41 sec.
Wikipedia 2008	71 Mo	13 min. 16 sec.

Summary of step 1

A first, minimal but complete version

- Fast to get a working solution
- Open to changes (filter approach)
- Several people can work on it (components)
- No barrier to optimization

Table of Contents

Feedback

Let's show what we did to our user

- Getting more details
- Fixing eventual misunderstanding

Feedback

User not happy with the output

```
Foundation 8
*** 42
****** 58
TABLE 1
Form: 2
Who 2
([images 1
II. 4
J. 7
Waters 1
B) 2
Special 9
Problems: 2
up—that 1
is, 34
coding—a 1
```

Feedback

User not happy with output $\Longrightarrow$ lot of "garbage" data

- We should discard all non-letters
- We should be case-insensitive

Feedback

User not happy with output $\Longrightarrow$ random order

- Most frequent words first
- Alphabetical order

Feedback

User not happy with the speed

- 13 minutes for less than 100 Mo
- What about 1000 Go/days ?

Speed improvement

How can we do the same thing, faster ?

Speed improvement

Check time spent in *WordCounter_run* call tree

Speed improvement

Most of the time spent searching for a matching word

- Linear search $\Longrightarrow$ $O(n)$ complexity
- Linear search $\Longrightarrow$ more words $=$ more time
- Hash table $\Longrightarrow$ $O(1)$ complexity
- Hash table $\Longrightarrow$ more words $\neq$ more time

Speed improvement

Words will be stored in a hash table

- Need to implement a hash table
- Need to find a good hashing for strings

Filtering the input

WordCounter_run before

```
while(Reader_hasNext(reader)) {
  if (!Reader_next(reader, &current))
    return 0;

  if (isspace(current)) {
    ...
  }
  else {
    ...
    StringBuffer_cat(&(self->buffer), current);
  }
}
```

Filtering the input

WordCounter_run after

```
while(Reader_hasNext(reader)) {
  if (!Reader_next(reader, &current))
    return 0;

  current = WordCounter_transformChar(self, current);
  if (current == '␣') {
    ...
  }
  else {
    ...
    StringBuffer_cat(&(self->buffer), current);
  }
}
```

Filtering the input

WordCounter_transformChar

```
char WordCounter_transformChar(WordCounter* self, char c) {
  if (!isalpha(c))
    return '_';

  return tolower(c);
}
```

The *WordCounter* interface

```c
typedef struct {
  char* str;
  unsigned long long count;
} Word;

typedef struct {
  DynamicArray words;
  StringBuffer buffer;
} WordCounter;

/* WordCounter initialization */
void WordCounter_init(WordCounter* self);

/* WordCounter termination */
void WordCounter_destroy(WordCounter* self);

/* Count words in an input */
int WordCounter_run(WordCounter* self, Reader* reader);

/* How much words found */
size_t WordCounter_nbWords(const WordCounter* self);

/* Copy words to an array */
void WordCounter_getWords(WordCounter* self, Word** array);
```

The *HashTable* interface

```
typedef size_t HashTableKey;

typedef struct {
  ...
} HashTable;

typedef struct {
  ...
} HashTableIterator;


void HashTable_init(HashTable* self);

void HashTable_destroy(HashTable* self);

size_t HashTable_size(HashTable* self);

HashTableIterator HashTable_begin(HashTable* self);

HashTableIterator HashTable_end(HashTable* self);

void HashTable_insert(HashTable* self, HashTableKey key, void* item);

HashTableIterator HashTable_find(HashTable* self, HashTableKey key, void* item);

HashTableIterator HashTableIterator_next(HashTableIterator it);

void* HashTableIterator_item(HashTableIterator it)
```

Sorting the output

```
void
printWords(WordCounter* wordCounter, FILE* out) {
  Word** words;
  size_t i, nbWords;

  /* Retrieve the words */
  nbWords = WordCounter_nbWords(wordCounter);
  words = (Word**)malloc(sizeof(Word*) * nbWords);
  WordCounter_getWords(wordCounter, words);

  /* Sort the words */
  qsort(words, nbWords, sizeof(Word*), Word_freqOrder_compare);

  /* Print the words */
  for(i = 0; i < nbWords; ++i)
    fprintf(out, "%s %llu\n", words[i]->str, words[i]->count);

  /* Job done */
  free(words);
}
```

Sorting the output

```
int
Word_freqOrder_compare(const void* a, const void* b) {
  const Word* aw;
  const Word* bw;

  aw = *(Word* const*)a;
  bw = *(Word* const*)b;

  if (aw->count == bw->count)
    return strcmp(aw->str, bw->str);

  return aw->count > bw->count ? -1 : 1;
}
```

Complexity

All those interfaces have minimal, low-complexity implementations

HashTable

Uses the simplest implementation (chained addressing)
Uses a well-known hashing function (*Murmur2*)

Quality & correctness

New features, new tests

- Run the previous tests
- Random text generator add "garbage" data
- Run an output check

Speed

Using the *time* command, with real text files

file	size	process. time	speedup
Lancet article	424 Ko	0.052 sec.	$\times 12$
Paradise Lost	476 Ko	0.065 sec.	$\times 21$
Wikipedia 2008	71 Mo	9.16 sec.	$\times 86$

Summary of step 2

A second, improved version

- Still a complete & correct version
- Still a simple program
- Still open to changes
- Closer to the user's needs

Table of Contents

Feedback

Let's show what we did to our user again

Feedback

User much happier with the speed, but hoping for more

- Every small gain can save him/her time & money

Feedback

User would like to give *PDF* and *HTML* files directly as input

Supporting *PDF* and *HTML* files with *input filters*

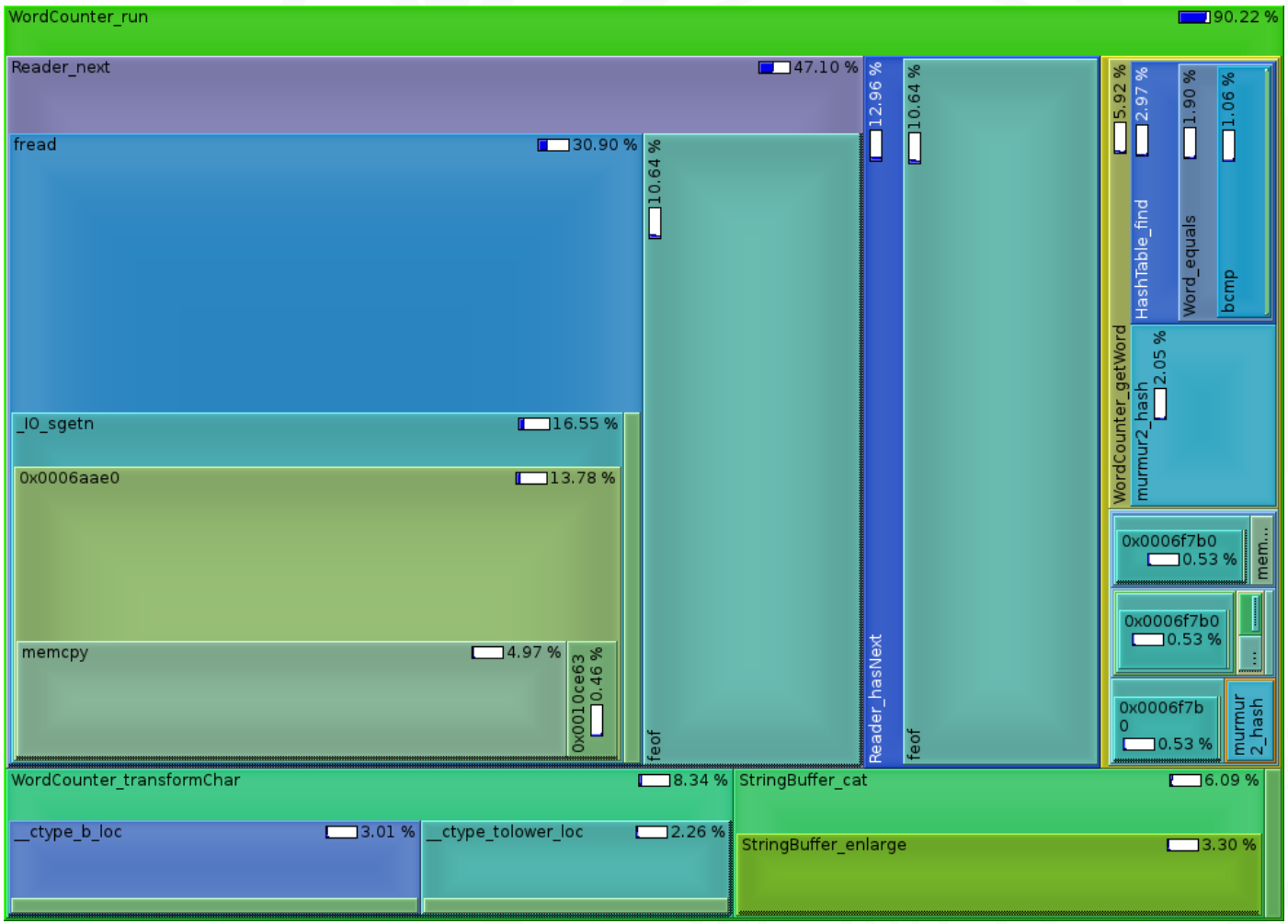
- *input filters* outputs text
- *HTML* $\Longrightarrow$ a *SAX*-based filter
- *PDF* $\Longrightarrow$ a filter using some PDF reading library

Speed improvement

How can we do the same thing, *even more* faster ?

Speed improvement

Check time spent in *WordCounter_run* call tree



Speed improvement

Improvements opportunities

1. 70% time in *Reader* $\Longrightarrow$ read blocks of data
2. 8% time in *WordCounter_transformChar* $\Longrightarrow$ precomputed table
3. Various simplifications & tricks
4. Playing with compiler options

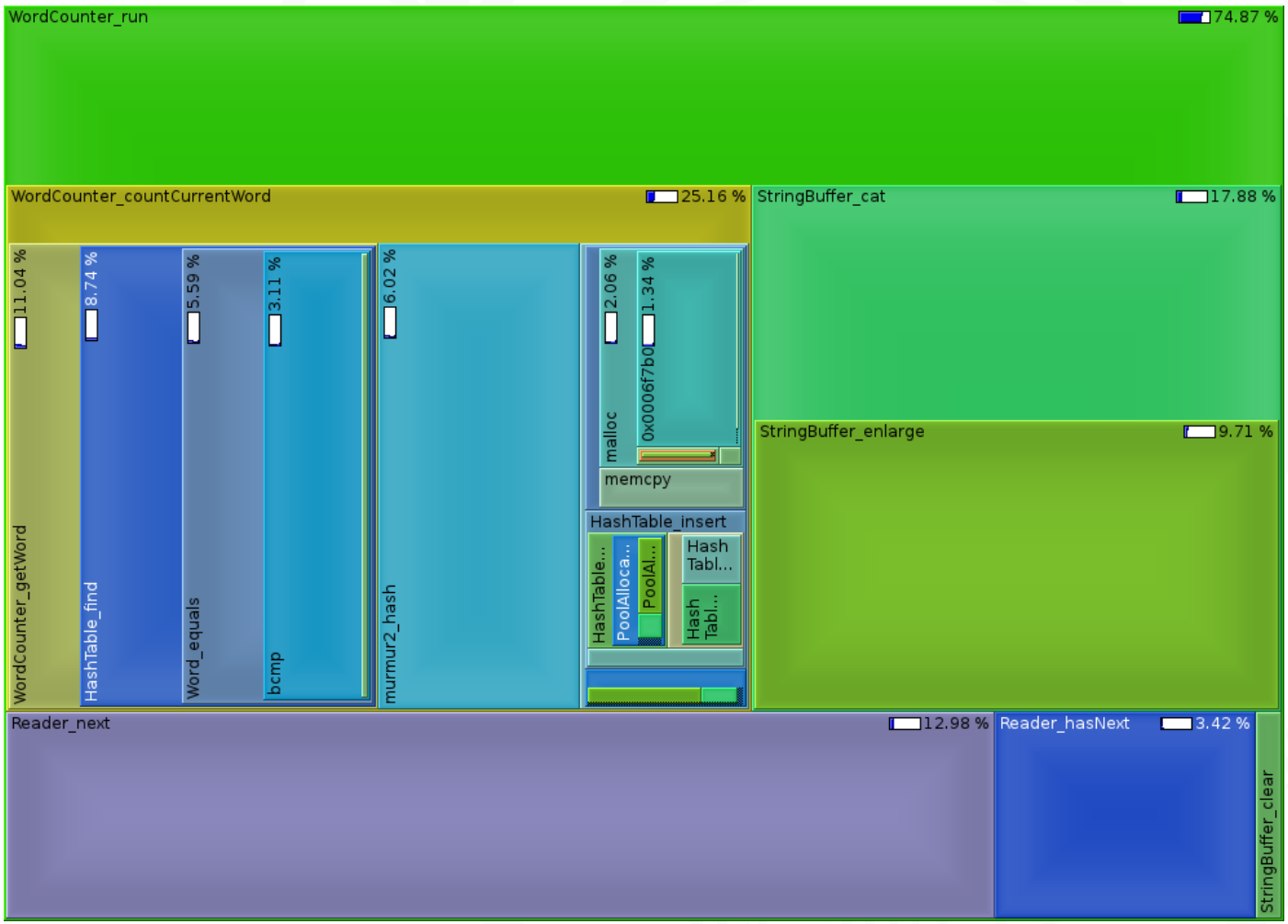
Speed

Using the *time* command, with real text files

file	size	process. time	speedup
Lancet article	424 Ko	0.018 sec.	×2.9
Paradise Lost	476 Ko	0.024 sec.	×2.7
Wikipedia 2008	71 Mo	3.1 sec.	×2.9

Speed improvement

Check time spent in *WordCounter_run* call tree



Speed

Further improvement speed is going to be lot of effort

1. 18% time appending characters
2. 13% time to read input
3. 10% time using hash table
4. 6% time hashing

Summary of step 3

A third, improved version

- Still a complete & correct version
- Not that so simple now ...
- ... but still under control !
- Reasonable insurance of quality

Table of Contents

Agility comes through various, combined ways

- Short development cycles with user's feedback
- Starts simple, minimal
- Using all opportunities available
- Proper coding discipline