The Simplicity Workout

Giovanni Asproni
Kevlin Henney
Peter Sommerlad
Agenda

• Simplicity admiration
  – What is simplicity?
  – What makes software "beautifully simple"?
  – Higher valuation of simple code
  – Continue work on Simplicity Manifesto

• Simplifying workout
  – Practice simplifying and learn simplification
What Is Simplicity?

**simplicity** *noun*
- the quality or condition of being simple.
- a thing that is simple.

**simple** *adjective*
- presenting little or no difficulty.
- easily done or understood.
- plain and uncomplicated in form, nature, or design.

*Shorter and Concise Oxford English Dictionaries*
What Makes Software Simple?

<table>
<thead>
<tr>
<th>Economy</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>Familiarity</td>
</tr>
<tr>
<td>Utility</td>
<td>Abstraction</td>
</tr>
<tr>
<td>Fitness for purpose</td>
<td>Compression</td>
</tr>
<tr>
<td>Comprehensibility</td>
<td>Explicitness</td>
</tr>
<tr>
<td>Consistency</td>
<td>Implicitness</td>
</tr>
<tr>
<td>Unsurprising</td>
<td>Emergence</td>
</tr>
<tr>
<td>Unification</td>
<td>Doing more with less</td>
</tr>
<tr>
<td>Symmetry</td>
<td>...</td>
</tr>
</tbody>
</table>
Simply Algorithmic

- Many of the most effective and enduring algorithms are surprisingly simple
  - Tony Hoare's Quicksort is a classic example of a divide-and-conquer strategy
  - Google's MapReduce scales simple functional primitives, map and reduce, into the large
  - The STL offers a simple and powerful approach to separating and combining algorithms and data structures
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];
    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}

int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

int matchstar(int c, char *regexp, char *text)
{
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}

Brian W Kernighan and Rob Pike, *The Practice of Programming*
http://cm.bell-labs.com/cm/cs/tpop/grep.c
Google's \textit{MapReduce}

\begin{verbatim}
main() {
    dictionary wordCount;
    for each document d {
        for each word w in d {
            wordCount[w]++;
        }
    }
}

Sequential word counter

MapReduce'd word counter

\begin{verbatim}
Map(document) {
    for each word w in document {
        EmitIntermediate(w, 1);
    }
}

Reduce(word, listOfValues) {
    count = 0;
    for each v in listOfValues {
        count += v;
    }
    Emit(word, count);
}
main() {
    Driver(Map, Reduce);
}
\end{verbatim}

MapReduce framework

\begin{verbatim}
EmitIntermediate(key, value) {
    ...
}
Emit(key, value) {
    ...
}
Driver(mapFunction, reduceFunction) {
    ...
}
\end{verbatim}

Essence of MapReduce framework
Not-So-Fearsome Engines

• There is a simple heart to many powerful frameworks, platforms and environments
  – The original Unix kernel was small and expressive
  – Some testing frameworks, such as JUnit and FIT, have a simple micro-architecture
  – Languages with a unified programming model, such as LISP and Smalltalk, tend to have a small and regular interpretive core
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
             ((variable? exp) (lookup-variable-value exp env))
             ((quoted? exp) (text-of-quotation exp))
             ((assignment? exp) (eval-assignment exp env))
             ((definition? exp) (eval-definition exp env))
             ((if? exp) (eval-if exp env))
             ((lambda? exp)
               (make-procedure (lambda-parameters exp)
                               (lambda-body exp)
                               env))
             ((begin? exp)
               (eval-sequence (begin-actions exp) env))
             ((cond? exp) (eval (cond->if exp) env))
             ((application? exp)
               (apply (eval (operator exp) env)
                       (list-of-values (operands exp) env)))
             (else
              (error "Unknown expression type -- EVAL" exp))))

Harold Abelson, Gerald Jay Sussman and Julie Sussman, 
The Structure and Interpretation of Computer Programs
http://academic.evergreen.edu/curricular/fofc00/eval.html
Anatomy of FIT

Parse
+ leader : String
+ tag : String
+ body : String
+ end : String
+ trailer : String
+ Parse (text : String)
+ text () : String
+ addToTag (text) : void
+ addToBody (text) : void
+ print (out : PrintWriter) : void

Fixture
+ doTables (tables : Parse) : void
+ doTable (table : Parse) : void
+ doRows (rows : Parse) : void
+ doRow (row : Parse) : void
+ doCells (cells : Parse) : void
+ doCell (cell : Parse, columnNumber : int) : void
  check (cell : Parse, TypeAdapter) : void

TypeAdapter
+ adapterFor ( : Class) : TypeAdapter
+ set (value : Object) void
+ invoke () : Object
+ parse ( : String) : Object

«creates»
static String tags[] = {"table", "tr", "td"};

public Parse (String text) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[]) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[], int level, int offset)
    throws ParseException {
    String lc = text.toLowerCase();
    int startTag = lc.indexOf("<"+tags[level]);
    int endTag = lc.indexOf(">", startTag) + 1;
    int startEnd = lc.indexOf("</"+tags[level], endTag);
    int endEnd = lc.indexOf(">", startEnd) + 1;
    int startMore = lc.indexOf("<"+tags[level], endEnd);
    if (startTag<0 || endTag<0 || startEnd<0 || endEnd<0) {
        throw new ParseException ("Can't find tag: "+tags[level], offset);
    }
    leader = text.substring(0,startTag);
    tag = text.substring(startTag, endTag);
    body = text.substring(endTag, startEnd);
    end = text.substring(startEnd,endEnd);
    trailer = text.substring(endEnd);
    if (level+1 < tags.length) {
        parts = new Parse (body, tags, level+1, offset+endTag);
        body = null;
    }
    if (startMore>=0) {
        more = new Parse (trailer, tags, level, offset+endEnd); trailer = null;
    }
}
Small and Regular Interfaces

• Simple protocols can be very flexible and highly composable, as well as easy to use
  – The Unix file (and general) I/O model and API
  – The Linda model of concurrent and distributed communication is based on few primitives
  – The RESTful (REpresentational State Transfer) approach to distributed architecture
  – The pipes and filters architecture common to Unix shells
RESTful Architecture

Key REST Principles

- Give every 'thing' an ID, e.g., a URI
- Link things together, i.e., the network is the program
- Use standard methods, e.g., HTTP verbs
- Resources with multiple representations, e.g., HTTP content negotiation
- Communicate statelessly, i.e., state is held as a resource or on the client

Stefan Tilkov, "A Brief Introduction to REST"
http://www.infoq.com/articles/rest-introduction
Writing Simple Code

• Correct, simple, fast (in that order)
  – TDD + refactoring + optimisation

• Decremental development
  – Determine what is needed of the solution (i.e., understand the problem), and...
  – Omit needless code

• Openness
  – Another pair of eyes
Writing Simple Code

What is the operating concept here? How else can we put that concept into action?

Edward De Bono, Simplicity
Quicksort in Erlang

Joe Armstrong, *Programming Erlang*
Quicksort in Erlang

```erlang
qsort2([]) -> [];  
qsort2([H | T]) -> {Less, Equal, Greater} = part(H, T, {[], [H], []}),  
    qsort2(Less)  
    ++ Equal ++  
    qsort2(Greater).

part(_, [], {L, E, G}) -> {L, E, G};  
part(X, [H | T], {L, E, G}) -> if  
    H < X -> part(X, T, {[H | L], E, G});  
    H > X -> part(X, T, {L, E, [H | G]});  
    true -> part(X, T, {L, [H | E], G})  
    end.
```

http://en.literateprograms.org/Quicksort_(Erlang)
Refactoring Workout

• Work in teams
• Look at the examples given
  – Try to understand what they do
• Refactor them to make them simpler
  – Determine what steps, both in code and outside the code, that you would take
  – Document these steps
http://wiki.hsr.ch/SimpleCode/
Wrapping Up