Writing Testable Code

F17

Principles

- Logically (albeit simplistic), we can think of a test as $y = f(x)$
- A test case is thus a tuple (x, e) where x is the input and e the expected result
- This works well for the "leaf components"/bottom layers of the system, but gets more complicated when we move up the layer stack
- For example, what about $y = g(u)$ where $u = f(x)$?

(We could imagine that g above is a logger and f produces a time stamp.)

Principles (cont'd)

Again, we are testing a layered/composite function $y = g(u)$ where $u = f(x)$

- If the test cases are composed into the form (x, e), then where is the error?
- Locating an error (blame control) is extremely important
- Composition = integration testing, which is good but on a higher level
- So, how can we test g in isolation?
- We want (u, e) and not care about f

Principles (cont'd)

Consider the test program t, we can do either

- $t(x) = g(u)$ where $u = f(x)$, and
- $t(f, x) = g(f(x))$

What's the difference?

Principles (cont'd)

- $t(x) = g(u)$ where $u = f(x)$
  - The dependency to f is fixed inside the implementation of t. We can't change it or test g in isolation.
- $t(f, x) = g(f(x))$
  - f is passed as a dependency
  - Easy to test g is isolation (take f out of the equation by using test cases where the relation between $u = f(x)$ are known)
- This is higher-order programming (n.b. an object is a higher-order function)
- Extracting the behaviour breaks long chains of dependence that make components hard to reuse
- Factored out dependencies gives smaller building blocks; again easy to reuse

Good code is testable code

When writing code, we must *always* consider its *testability*

- Different functions should be testable in isolation
  - Minimise dependencies
  - Minimise possible sources of error
- Encapsulation can hamper testability
  - Can all functions be tested/accessed?
  - Can we easily supply values to test?
  - Can be easily get control input/output?
  - Code coverage vs. encapsulation?

Example 1: Logger

Let's say we have designed and implemented a library for logging C-strings to disk in `logger.c`

- The logger must be initialised in `initLogger(filename)`
- Messages are written to the logger with `logMessage(msg)`
- The logger is torn down with `destroyLogger()`
- Messages are buffered and flushed internal to the logger

Let's have a look at the code, and then talk about its testability.

```c
/* logger.c */

#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <time.h>

#include "logger.h"

/* Constants */
#define BUFSIZE 1048576
#define TIMESTAMPMAX 26

/* Module variables */
static FILE *logfile;
static char logbuffer[BUFSIZE];
static unsigned int logsiz = 0;
static time_t logtime;

/* Start code */
void initLogger(const char *fn) {
  assert(fn);
  assert(logfile == NULL);

  logfile = fopen(fn, "w");
}
```

```c
void logMessage(const char *msg) {
  assert(msg);
  assert(logfile);

  int msgsiz = strlen(msg) + 1;

  if (logsiz + msgsiz + TIMESTAMPMAX > BUFSIZE) {
    flush();
  }

  time(&logtime);
  char *timestamp = ctime(&logtime);

  while (*timestamp != '\n')
    logbuffer[logsiz++] = *timestamp++;
  logbuffer[logsiz++] = ' ';
  while (*msg)
    logbuffer[logsiz++] = *msg++;
  logbuffer[logsiz++] = '\n';
  logbuffer[logsiz] = '\0';
}

static inline void flush() {
  fwrite(logbuffer, logsiz, 1, logfile);
  logsiz = 0;
}
```

```c
void destroyLogger() {
  assert(logfile);

  flush();
  fclose(logfile);

  logfile = NULL;
}

/* Sample use
int main(void) {
  initLogger("testlog.txt");
  logMessage("Foo");
  logMessage("Bar");
  destroyLogger();
  return 0;
}
*/
```

Can you see any problems with testing the logger?

Let's examine what tests we should write

- Are log messages printed correctly?
- Are log messages' time stamps correct?

Problem list

1. The logger always writes to a file on the disk
2. The size of the buffer is fixed and requires a somewhat large test data
3. Time stamps are created internally so comparing two tests not possible (will always differ on time)

This design is not so good – the code is difficult to test, reuse and modify

An improved logger

1. Control where the output goes
2. Control over buffer size
3. Control of time stamps

```c
/* better-logger.c */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <time.h>

/* Constants and helper functions */
#define TIMESTAMPMAX 26
#define initLogger(p) initLoggerWithPath(p, BUFSIZE)

/* Module variables */
static FILE *logstream;
static char *logbuffer;
static unsigned int logsiz;
static time_t logtime;
static unsigned int BUFSIZE = 1048576;
static unsigned int USES_BUFFER = 1;

void initLoggerWithPath(const char *_fn, unsigned int _bufsiz) {
  assert(_fn);
  assert(logstream == NULL);

  initLoggerWithStream(fopen(_fn, "w"), _bufsiz);
}
```

```c
void initLoggerWithStream(FILE *_logstream, unsigned int _bufsiz) {
  assert(_logstream);
  assert(logstream == NULL);

  BUFSIZE = _bufsiz;
  logstream = _logstream;

  if (USES_BUFFER = BUFSIZ > 0) {
    logbuffer = (char*) malloc(BUFSIZE);
  }
}

static inline void flush() {
  fwrite(logbuffer, logsiz, 1, logstream);
}

static inline void flushAndReset() {
  flush();
  logsiz = 0;
}

const char *buffer() {
  return logbuffer;
}
```

```c
void logMessageWithTime(const char *_msg, const time_t *_logtime) {
  assert(_msg);
  assert(_logtime);

  const int msgSize = strlen(_msg) + TIMESTAMPMAX;

  if (USES_BUFFER) {
    if (logsiz + msgSize > BUFSIZ) flushAndReset();
  } else {
    if (BUFSIZ < msgSize) logbuffer = realloc(logbuffer, BUFSIZ = msgSize);
  }

  char *timestamp = ctime(_logtime);

  while (*timestamp != '\n')
    logbuffer[logsiz++] = *timestamp++;
  logbuffer[logsiz++] = ' ';
  while (*_msg)
    logbuffer[logsiz++] = *_msg++;
  logbuffer[logsiz++] = '\n';
  logbuffer[logsiz] = '\0';

  if (!USES_BUFFER) flushAndReset();
}
```

```
void logMessage(const char *_msg) {
  assert(_msg);

  time(&logtime);
  logMessageWithTime(_msg, &logtime);
}

void destroyLogger() {
  assert(logstream);

  flushAndReset();
  fclose(logstream);
  free(logbuffer);

  logstream = NULL;
}

/* Sample use
int main(void) {
  initLogger("/dev/null");
  logMessage("Foo");
  fprintf(stderr, "In buffer: %s", buffer());
  destroyLogger();
  return 0;
}
*/
```

Observations

- Testing the logging isolated from file handling is now possible
- Code slightly more complex, but mostly just does the “obvious things” and depends on library functions being correct
- The code is now easy to test, reuse and extend
- (But what about timestamps?)

```python
from time import ctime

def buffered(time, msg):
    global logbuffer
    if len(logbuffer) > 1024:
        for msg in logbuffer:
            print msg
        logbuffer = [time + " " + msg]
    else:
        logbuffer.append(time + " " + msg)

def unbuffered(time, msg):
    print time, msg

def initLogger(behaviour = buffered):
    global logbehaviour, logbuffer
    logbehaviour = behaviour
    if behaviour is buffered:
        logbuffer = []

def logMessage(msg, time = ctime()):
    logbehaviour(time, msg)

initLogger()
logMessage("Foo")
logMessage("Bar")
```

But even better

We can create a behaviour that simply prints to a specified string and that's that.

```
def createStoreToArrayBehaviour(buffer):
    def storeToBuffer(time, msg):
        buffer.append([time, msg])
    return storeToBuffer

myBuffer = []
initLogger(createStoreToArrayBehaviour(myBuffer))
logMessage("Foo")
logMessage("Bar")
for msg in myBuffer:
    for element in msg:
        print element
```

Which prints the elements as expected.

Testing a binary search tree

```
/* bst.h */

typedef struct _tree tree;
typedef tree *Tree;
struct {
  int value;
  Tree left, right;
};

Tree mkTree(int v);
void insert(Tree t, int v);
```

Testing a binary search tree (cont'd)

What's wrong with this test and/or module?

```
Tree t = mkTree(5);
insert(&t, 1);
insert(&t, 3);
insert(&t, 7);
assert(t->element == 5)
assert(t->left->element == 1)
assert(t->left->left == NULL);
...
```

Testing a binary search tree (cont'd)

```
/* Returns the number of nodes in a tree */
int size(Tree t) {
  return (t) ? 1 + size(t->left) + size(t->right) : 0;
}

/* Returns the longest path to a leaf in a tree */
int depth(Tree t) {
  return (t) ? 1 + max(depth(t->left), depth(t->right)) : 0;
}
```

Allows us to test important properties of a tree without knowing about its implementation. Does it grow on insert? Of duplicates?

```
Tree t = mkTree(1);
assert(depth(t) == size(t) == 1)
for (int i=2; i<5; ++i) {
  insert(&t, i);
  assert(depth(t) == size(t) == i)
}
```

Testing a binary search tree (cont'd)

```
char *getPathForElement(Tree t, int element) {
  char *result = *path = (char*) malloc(depth(t));
  while (t) {
    if (element == t->element) {
      *path = '\0';
      return result;
    } else if (element < t->element) {
      *path++ = 'L';
      t = t->left;
    } else {
      *path++ = 'R';
      t = t->right;
    }
  }
  return NULL;
}
```

Allows us to trace e.g., moving elements on delete, etc.

```
Tree t = mkTree(3); insert(&t, 5); insert(&t, 1); insert(&t, 2);
assert(strcmp(getPathForElement(t, 2), "LR") == 0)
assert(strcmp(getPathForElement(t, 5), "R") == 0)
```

Testing a binary search tree (cont'd)

```
int getElementForPath(Tree t, char *path) {
  while (t && *path) {
    switch (*path++) {
    case 'L':
      t = t->left;
      continue;
    case 'R':
      t = t->right;
      continue;
    default:
      return -2;
    }
  }
  return t ? t->element : -1;
}
```

Allows us to trace e.g., moving elements on delete, etc.

```
Tree t = mkTree(3); insert(&t, 5); insert(&t, 1); insert(&t, 2);
assert(getElementForPath(t, "LR") == 2)
assert(getElementForPath(t, "R") == 5)
```

Observations

- Writing testable code will force you to stay away from certain patterns
  - Example: function that initialises an entire data structure in a single hit (constructor)
  - We want to be able to do piecemeal testing
  - Downside: can observe object in invalid state
- Avoid global state
  - Persists between tests (includes singletons)
  - No global state sometimes makes things more complex
- **Do use** the single responsibility pattern
  - Consequence: more units of code (functions, classes, modules, etc.)
- Minimise dependencies
  - ...or your tests will easily become very complex

Hints

- Picking good names is extremely important
- Be smart but avoid "fancy coding"
- Short code is readable code, too short code is unreadable
- Many small functions that can be composed increases reusability and maintainability – and therefore testability!
- (Think of testing as reuse)
- Use asserts, especially for "this should never happen"
- Avoid `NULL`
- Always initialise variables, even if you "know" they will be assigned before read later
- Remove redundant or unused things – less clutter is more readable
- Avoid assignments in boolean expressions and function arguments

Hints (cont'd)

- Test for division by zero if you don't know the value of the denominator
- Make the loop invariant clear – preferably in a single place
- Avoid many **`break`**s, **`return`**s etc. – makes code hard to follow
- Always test for the most likely case first (that's the one the next guy will be looking for anyway)
- Always test for boundary conditions and array sizes
- Inspect the return values for functions you expect to always succeed (like `malloc`)
- Whenever you use `malloc` inside a function to return data, note it in the function's documentation
- Always hand resources back properly