Mosig – Software Engineering

Second Week

Overview:

Q&A

Workshops:

Code review
How do you test a program?
How do you evaluate the performance of a program?
Design and performance
The first step towards a netcat program of our own...

Homework:

Full code review on the version 6 of the echo program.
- Full comments
- Full javadocs

Full code review on the channel version of the echo program.
- Full comments
- Full javadocs

Workshop: Code review

Exchange your home work, with your neighbor.
You are asked to provide the following:

- Feedback on comments in the list/props code.
- Feedback on stream implementation and comments

Directly write the feedback in the files.
Send back the review to the authors.

Workshop: How do you test a program?

→ Discuss **what** you are trying to achieve and **how** you would go about it
→ Can you test a program entirely? Justify your answer.
→ Discuss how coverage helps?
→ Outline a strategy to test a program.

Workshop: How do you evaluate the performance of a program?

→ Discuss **what** you are trying to achieve and **how** you would go about it
→ Any different evaluating C versus testing Java?
→ What do you think of elapsed time as a performance measure?
→ Explain why you should consider average elapsed time?
→ Explain why you should consider standard deviation?

Using our C list implementation, we can run a basic incomplete test,
based on a random insertion in a list.

You can look at the code in C in Week1.c.list → test.c
→ **`random_string_insert_test`**
→ **`random_struct_insert_test`**

We can run it:

```
$ cd Week2.c.lists
```

```
$ ./sample -tests
...
Running tests: cflags are -Wall -std=gnu99 -g3
String random insert: 284 ms
Struct random insert: 305 ms
```

We can run it also like this:

```
$ cd Week2.c.lists
$ ./sample -tests
...
Running tests: cflags are -Wall -std=gnu99 -O3
String random insert: 140 ms
Struct random insert: 141 ms
```

Do you understand the differences: **-g3** versus **-O3** ?

Modify the makefile:

```
CONFIG_DEBUG=y
CONFIG_GPROF=y
CONFIG_COVERAGE=n
CONFIG_PERFS=n
```

and then launch:

```
$ make gprof
...
All done with tests.

gprof sample gmon.out > sample.gprof
```

and then look at the file “sample.gprof”.

Then, you can look at the coverage of that test:

```
$ make gcov
```

Workshop: Design and Performance...

Different designs could exhibit different performance characteristics. That is, certain designs are more suited to certain workloads than others.

Look at the home-grown framework for performance evaluation in Week2.java.lists → perfs. It is certainly not a full-fledge framework, but it will give you an overall idea and show you cool snippets of code for performance evaluation. Look at classes Harness and Test in package edu.ujf.perfs.

Note: it computes average elapsed times, standard deviation is missing.

We will use a simple performance benchmark, a random insert of ordered elements.
The test is in all point similar to the previous test we used on our C list implementation.

Look at the code in Week2.java.lists → perfs (folder)
→ edu.ujf.perfs.utils.TestA.
→ edu.ujf.perfs.utils.TestB.

Our performance evaluation runs the same simple benchmark on various Java implementation.
You can launch the measure by launching in a terminal the script run.sh in Week2.java.lists.
On my machine, I get:

```
Test: edu.ujf.perfs.utils.TestA
-------------------------------------------------
  Warming up...
  →  warmup: 5.143s
  Run all tests...
    Random insert (linked list): 231ms
    Random struct insert (linked list): 92ms
    Random insert (array list): 7ms
    Random insert (double linked list): 149ms
  Elapsed: 5.073s
-------------------------------------------------
Test: edu.ujf.perfs.utils.TestB
-------------------------------------------------
  Warming up...
  →  warmup: 150ms
  Run all tests...
    Random insert (optimized, linked list): 2ms
    Random insert (optimized, double linked list): 1ms
  Elapsed: 163ms
```

You are asked to discuss the various performance numbers above.
How do you explain them based on the differences you see in the various designs.

- Linked lists
- Array lists
- Double-linked lists (with new API)

Workshop (E): NETCAT

Let's conduct a more realistic design and implementation. Our starting point will be netcat, or in fact, a much simpler version. Our goal will be just the symmetrical echo between two different terminal windows.

To launch the accepting side of netcat:

```
$ nc -l 127.0.0.1 5555
```

To launch the connecting side of netcat:

```
$ nc 127.0.0.1 5555
```

Once connected, everything you type in one terrninal is echoed in the other.
Notice that the echo happens line per line, when the “enter” key is pressed.
So the specification is quite straightforward, however the implementation can be tricky.

Echo Project, source folder step1:

Look at the project Echo, we have six steps, illustrating six different implementations of a trivial echo: reading from the standard input stream and echoing the read characters to the standard output stream. Trivial? Certainly, but the various steps drive two points home:

One is to show how important it is to pay attention to the specification of the libraries you use.
Another is to illustrate different typical Java constructs/concepts.

From step1 to step6:

Can you tell which versions are correct or still buggy?

You are asked to understand what the bugs are and detect the poor coding patterns.

Write down your remarks on the code, what you like, what you dislike.

Echo Project, source folder step2:

We introduce code modularity, wanting to illustrate that a good code is a code well-structured.

Why?

First, a good structure helps writing and debugging your code.

Second, over time, a good structure helps maintaining and evolving that code. Previously, the code had no particular structure, the code was fully specific to the echo problem, with very special attention paid to code modularity.

To prepare for our long-term goal of a netcat application, we will introduce different concepts that will structure the code. At first, it will feel like it introduces complexity, and in some sense, it is true. Some might say it would be overkill for a simple echo. Although true, modularity will prove useful when tackling the more complex problem of a netcat implementation.

Look at edu.ujf.samples.channel:

Here we introduce the well-accepted concept of a producer and a consumer, communicating through a channel. Notice how the coding patterns help decouple the different parts of the code:

- Our producer is a line producer, from an input stream (any input stream).
- Our consumer is a line consumer, from a channel (any channel). It echoes to output stream (any stream).
- The channel of lines is totally independent from the producer and the consumer.
- Notice the callback pattern used by the channel.

Home work:

HOMEWORK MUST BE SURRENDERED SUNDAY BEFORE MIDNIGHT.

IT MUST BE AN ECLIPSE PROJECT

THE PROJECT NAME MUST BE COMPOSED OF THE LAST NAMES OF THE STUDENTS (GROUP)

Full code review on the **version 6** of the echo program.
- Full comments
- Full javadocs

Full code review on the edu.ujf.samples.channel package.
- Full comments
- Full javadocs

Write full JUnit tests for the edu.ujf.samples.channel package.
→ The goal is 100% coverage.