CEN 4072

Software Testing

Team term project: Fall 2019

Team design project: *If anything can go wrong, it will!*
Submission deadline: **December 3, 2019**

Team name:
USFIDs:
E-mails:

TEAM MEMBERS' CONTRIBUTIONS

Member [Name]:

Member [Name]:

Member [Name]:

Member [Name]:

BACKGROUND

In this term project, you will get engaged in software development as well as debugging and testing. Both activities will be *team efforts* whose components are summarized in the table below.

		Weeks of the Semester													
Phase	**Activities**	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
I	Code design	•	•												
	Coding		•	•	•										
	Fault seeding			•	•										
	Documentation			•	•										
II	Test-bed building				•	•	•								
	Test data generation					•	•	•							
	Test routines					•	•	•	•						
III	Debugging							•	•	•	•	•	•	•	
	Testing								•	•	•	•	•	•	
IV	Test report											•	•	•	•
	Project report	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Your team will design and implement a simple student data manager that the Software Testing class can use to keep a record of student performance throughout the course. Once

your implementation is complete, your task will be to insert (seed) faults into the program. The severity of the faults should range from the subtle to the fatal. Once this is completed, your code will be given to another team and, at the same time, you will receive the code of a different team. The next step is then to debug and test the code that you are entirely unfamiliar with.

To have the maximum opportunity to generate faults, this class will use the *C/C++ programming language*. According to Murphy's law: in C/C++ if anything can go wrong, it will, which is what we need in this class!

OBJECTIVES

The objective of this term project is to get involved in both bug hiding as well as bug finding, i.e., you will be the hunter as well as the hunted! Your objective is to find everything that is wrong (or possibly can go wrong) with the code. Your mission will be declared successful if

found bugs and deficiencies ≥ documented seeded faults

That is, you should be able to find all bugs your classmates seeded into the code, plus some more related to the design and the implementation of the code. At the end of the class, the two teams (designers and testers) will compare notes to see how well the mission was accomplished.

COMPANY SETUP

Your team will form a software testing company, a small 3-4 member start-up company. This will be a symbolic company only; no real company formation is necessary. Company formation requires the following:

- **Name:** Give your company a name, e.g., Bug Hunters LLC.
- **Management:** decide on a top-down (CEO based) or egoless (democratic) approach. I would recommend a democratic approach.
- **Art:** the company needs a visual, artistic representation. A logo represents the company's vision, which should be conveyed visually to potential customers. This art piece can be a hand-drawn sketch or done by a professional system such as Adobe Illustrator.

Please note the following important deadlines:

- **September 10:** company name and structure are due
- **October 1:** class-roll fault-seeded program
- **October 8:** company art/logo is due
- **December 3:** company test documentation is due

PROJECT DESCRIPTION

For the project to be successful, it must be kept on schedule as detailed above. Although this is a design project with most of the details are left intentionally open, some of the components need clarification.

Phase I: Coding

Your first assignment is to design and implement a small class-roll maintenance system. For each student, the following data is needed:

- Name (up to 40 characters)
- USF ID (10 characters)
- Email (up to 40 characters)
- Grade of the presentation (numerical value from 0 (F) to 4 (A))
- Grade of essay (numerical value from 0 (F) to 4 (A))
- Grade of the term project (numerical value from 0 (F) to 4 (A))

The capabilities the system must support are:

- Read/write student data
- Add/delete students
- Retrieve student data based on a search by name, ID or email
- Update any or all data fields

The system consists of the main routine and various functions supporting the capabilities. There is no need (time) to implement a complicated user interface; a simple console-based text interface would be perfectly acceptable. The way the data is stored in a file must be made transparent to the user!

Once the coding is done, it is time to insert faults. You are free to add all sorts of bugs; however, the following types of faults are recommended:

- Un-initialized variables, pointers, and objects
- Array bound errors
- Dangling pointers
- Memory trouble: allocation and leaks
- Function call error
- Unreachable (dead) code

Now that the design and the bug seeding have been completed, you need to document your work. The documentation must have at least three components:

- Design: document your system design, including requirements, architecture, data format, and all relevant information that is necessary to use the system. This should be a short document.
- Code: insert comments inside the code so that the testers have a clear understanding of your code.
- Fault: describe each inserted fault in detail. This part of the document will be kept confidential until the last week of team demonstrations!

Please submit your class-roll maintenance system electronically to lpiegl@gmail.com by no later than October 1, 2019. Please put all functions in one file and send three files: (1) source code of the correct code, (2) source code of the fault-seeded code, and (3) the documentation in Microsoft Word. The files should be named as follows:

- **File with no bugs: Team_Polk_NoFault.cpp**
- **File with bugs: Team_Polk_FaultSeeded.cpp**
- **Documentation: Team_Polk_Documentation.doc**

(replace "Polk" with the name of your team)

Please do not use third-party components for this project. If there is a crash in the third-party code, it may become impossible to test the code and to run different data sets.

Phase II: test preparation

You received the code and part of the documentation from another team. Before you go bug hunting, some important software pieces need to be developed:

- *A test-bed environment*: this is a small system that allows you to read/write test data, run tests, analyze test results, and perhaps generate test data randomly.
- *Test data generation*: you must be able to generate test data to run your test. Test data can be produced locally, obtained from third-party vendors, or computed by special-purpose algorithms.
- *Automated testing routines*: these routines test the individual functions as well as simulate user interactions. Simple C/C++ code, as well as scripts, may be applied.
- *Debugger*: the built-in C/C++ debugger is recommended that comes with the integrated environment.
- *Code analyzer*: several tools will be introduced in this class that can be used as run-time code analyzers.

Phase III: debugging and testing

Once you got your environment and test generators in order, it is time for debugging and testing. Your debugging sessions should pay particular emphasis to the following:

- How faults could be generated during the development process. It helps a lot if you know what could go wrong, i.e., what is it that you are looking for.
- How to make programs fail. To find the bug, there must be a bug, i.e., you must be able to break the program or identify deficiencies.
- How to reproduce the problem, which involves reproducing the environment, the execution, as well as the interactions.
- How to simplify the problem, which involves simplifying the input, user interaction, and automatic as well as manual simplification.
- How to debug errors, including isolating origins, understanding control flow, tracking dependencies, slicing, and code smells.
- How to detect anomalies by capturing normal behavior, using statistical debugging and dynamic invariants, comparing coverage, and by collecting data on actual applications.
- How to establish what caused the failure, which means finding actual causes, narrowing down causes, and to use causes in debugging.

It is not necessary to apply all of the above techniques. They are listed for completeness and to assist in the debugging process. You are free to employ whatever works best for you.

Just because the program is well debugged, it does not mean it is useful. A thorough test is needed to verify its usefulness for the intended application. You will perform two types of tests:

- Open-box test: you are free to look at the source code as well as the documentation.
- Closed box test: only the executable code is available, and you need to find out what could be wrong with the code by actually running test cases.

It is recommended that you start with closed-box tests, run several examples to have a feel for what the software does and how it behaves. After you hammered the code with lots of tests and found out what may be right/wrong with it, you may then want to look under the hood.

As far as the actual tests are concerned, two sets are recommended. The first set contains the following:

- Requirements test: check how well the software satisfies the stated/intended requirements.
- Design test: poorly written software is usually traced back to lousy design. It is recommended that you look at the overall architecture, cohesion, coupling, portability, changeability, and ease of understanding.
- Quality test: all aspects of software quality are relevant from quality design to quality documentation.
- Documentation: please check both the written documentation that came with the software as well as the documentation inside the code.

The second set of tests includes:

- Stress test: please stress the system to its limits.
- Volume test: see how a large amount of data is handled.
- Security test: check how well the system protects the integrity of the data (if applicable).
- Timing test: check if the required tasks are performed within the allotted time.
- Recovery test: see how well the system recovers (if at all) from faults.

Some of the test categories may not be applicable; however, they are listed above for completeness and to give you some ideas of what kind of tests you may want to throw at the system.

Phase IV: reports

There are two reports that you need to submit by the deadline: the project report and the fault report. The fault report is part of the project report and must contain at least the following components:

- Software name, version, team ID
- Fault location: list the name of the function and the line where the fault occurs
- Fault description: a short description of the fault
- Fault severity: from severe (causes a crash) to cosmetic (could be done more efficiently or elegantly)
- Fault type: memory leak, garbage accumulation, etc.
- Fault impact: slow computation, inaccurate result, chaotic behavior, crash, etc.
- Fault scope: local infection, infects more than one unit, infects an entire module, etc.
- Suggested fix: this may not be your responsibility, however, suggest ways of eliminating the problem. Simple problems such as exceeding array bounds are easy to fix; however, others that may require a change in architecture could be quite complicated.

Once your fault report is done, please make it part of your project report whose details are given below.

DELIVERABLES

There are several deliverables:

- Company setup and visual art: name, management, and logo
- The class-roll program that you need to submit by **October 1, 2019**. After submitting your system, you will receive another system for debugging. **Each team submits and receives one class-roll program!**
- Progress report due on the second day of the presentation of the team.

- A project description document following the outline above. **Each team submits one document!**
- A short live demonstration of your debugging and testing experience during the last week of classes. This is a team demonstration! Please prepare a presentation (using PowerPoint, Prezi, etc.) to guide your report.
- A video of the full team presentation. A video link is submitted pointing to the video where it is hosted.

Using this file as a template, write your project description that contains, at a minimum, the following components:

- A short **abstract** of no more than 200 words.
- A list of **keywords** of no more than 5.
- An **introduction** section where you describe the challenges of writing code full of bugs and finding bugs in the unfamiliar code.
- A comprehensive section on **the entire project** where you describe step-by-step how the software system was tested and debugged following the outline above.
- Proper **illustrations** throughout the text.

FORMAT

Use this MS Word file as the template and enter your text directly after the team members' contribution. The length of the document should be proportional to the depth and the breadth of the project. Please name the file as follows:

Team_Name_TP.doc, e.g., **Team_Tyler_TP.doc**

When submitting the file, the subject line should contain the course prefix as well as the file name above. Example:

CEN_4072_Team_Tyler_TP.doc

SUBMISSION

Your project should be submitted electronically via e-mail attachment. Please send the file and the video link to lpiegl@gmail.com no later than **December 3, 2019**.