EECS3311-W18 – Tracker Project

Table of Contents

1 Introduction 2
2 Requirements 2
3 Safety Critical System 3
4 Specification 4
4.1 Undo/Redo 5
5 Directory structure of your project 5
6 How to submit your report and project 6
7 The structure of your report 6
8 Modes of failure 7
9 Appendix 7

In this Project, you may work on your own or in teams of at most two members.

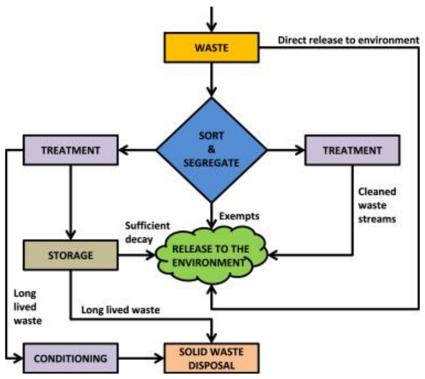
If you work in a team, you are still individually responsible for submitting the Lab by the due date. Your team should thus setup a **private** *github* repository where both members of the team have access to all the design and coding material. If the team dissolves for any reason whatsoever, each individual member of the team is responsible for completing and submitting the project on their own.

The team submits electronically only under the Prism ID of one of the team members. Please ensure that there are no duplicate submissions which may result in no grade at all. Likewise, only one of the team members submits their Design document on *moodle*.

Version 3, 02 March 2018.

1 Introduction

A tracker system monitors the position of waste products in nuclear plants and ensures their safe handling. Our customer requires a software system that operators use to manage safe tracking of radioactive waste in their various nuclear plants.



2 Requirements

We have so far elicited the following information from our customer.

Containers of material pass through various stages of processing in the tracking part of the nuclear plant. The tracking plant consist of several phases usually corresponding to the physical processes that handle the radioactive materials. Not all plants have precisely the same phases.

As an example, containers (containing a possibly radioactive material type) might arrive at an initial unpacking phase where they are stored for further processing depending on their material contents. All nuclear plants have only the following types of material: *glass*, *metal*, *plastic*, *liquid*. No other materials are tracked.

A subsequent phase might be called the *assay* phase to measure the recoverable material content of each container before passing onto the next phase. A next stage might be a *compacting* phase. A compacting phase might involve dissolving metal contents or crushing glass. Not all material types can necessarily be handled in a phase. For example, we should not move containers with liquid into a compacting phase. Finally, the products of the process might be placed in storage. There may be other phases in an instance of the tracker.

Each container has a unique identifier and contains only one type of material. It is labelled with a preliminary radiation count (in *mSv*). When a container is registered in the system, it is also placed in a phase (not necessarily an initial phase).

The *sievert* (symbol: Sv) is a unit of ionizing radiation dose in the International System of Units (SI) and is a measure of the health effect of radiation on the human body. Quantities that are measured in *sieverts* are intended to represent the stochastic health risk, which for radiation dose assessment is defined as the probability of cancer induction and genetic damage. One *sievert* carries with it a 5.5% chance of eventually developing cancer.[1]

For a given plant, there is an initial setup of two important fixed global parameters: there is a limit on the maximum radiation in any phase of the plant (in units of mSv), and there is also a limit on the maximum radiation that any container in the plant may have (in mSv). An error status message shall be signaled if there is an attempt to register a new container in the system with radiation that exceeds the container limit.

Another operation is to add a new phase (this is information provided by the Domain experts). Requirements elicitation so far yields that a new phase is specified by a phase ID, a name (e.g. "compacting"), a limit on the maximum number of containers in the phase, and a list of material types that may be treated in the phase. A phase may also be removed if there are no containers anywhere in the system. Also, it is possible for an operator to move a container from one phase to another.

Obviously when dealing with dangerous materials, it is very important to ensure that no material goes missing and that care is taken to avoid too much radioactive material getting into a phase, in case there is a buildup of dangerous substances in one area. The tracking manager is responsible for giving permission to movements of containers between processing phases to avoid dangerous situations.

3 Safety Critical System

This is mission critical application

- *Goal1/Setup*: Tracker operators should be able to set up a new tracking system, with phases where each phase is specified by an identifier, number of containers that can be handled, radiation levels and materials that can be handled. Operators should also be able to remove phases.
- Goal2/Operation. Operators should also be able to record new containers, specify the material type, and radiation level of the container, move the container from one phase to another, and when complete, safely remove the container from the system.
- Goal3/Safety. The tracking system shall ensure the safety of the plant so that radiation levels are safely handled, preventing the execution of erroneous commands, and notifying the operator with meaningful status and error messages when they occur.
    - No phase in the system shall have radiation greater than the maximum allowable.
    - No container in the system shall have radiation greater than the maximum allowable.
    - The count of containers in a phase shall not be greater than the maximum allowable.
    - No phase handles an unexpected material.
    - A container resides in only one phase

[1] https://en.wikipedia.org/wiki/Sievert

Please ensure that you follow all submission instructions carefully. Submissions that do not comply with the instructions will not receive a grade.

4 Specification

Requirements elicitation has resulted in an abstract grammar for the user interface.[2] Eventually, there will be a desktop application for users (the operators of the nuclear plant). However, the concrete GUIs will be developed at a later stage and are beyond the scope of this project. The concrete GUIs will support all the operations in the abstract grammar.

Also, one Use Case has been developed so far. This Use Case has been refined into an Acceptance test called at1.txt.[3]

When an operation can be performed, the system responds with "ok". However, when some operation cannot be performed, the acceptance test currently responds with a meaningful error message.[4] You are required to perform further requirements elicitation to obtain the precise error messages.

Instead of a complete specification, you are also provided with an Oracle.[5] You will thus need to develop your own additional acceptance tests at1.txt, at2.txt, at3.txt etc. (in the student folder). It is interesting to note that these acceptance tests (as structured rather than code) are independent of the programming language used to develop the system. These acceptance tests can thus be written by users who are not familiar with programming. Such acceptance tests can thus be developed before the system is ever implemented. A good set of acceptance tests is thus part of the Tracker specification.

You must develop a program that can be tested from the console. For example, executing

```
tracker -b at1.txt
```

shall produce the output *at1.expected.txt* (see course directory), with the proper agreed upon error messages. The "-b" switch stands for batch mode (there is also a "-i" switch for interactive mode). Both relative and absolute paths using Linux conventions shall be supported.

Integer identifiers (e.g. for use as an ID) are in the range 1 to 9,223,372,036,854,775,807.[6] This is the maximum number of identifiers that we need to support.

In all cases the behavior of the Oracle is definitive. Your program must match the Oracle character-by-character.

[2] In the course directory, this is: /cs/course/3311/labs/project/tracker.defns.txt.

[3] Also in the above course directory.

[4] The error messages are described in /cs/course/3311/labs/project/errors.txt. Note that where there is difference between this file and the oracle, the oracle governs. There are some additional messages associated with the undo/redo mechanism.

[5] /cs/course/3311/labs/project/oracle.exe

[6] 64 bit integers are in the range −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807, or from −(2^63) to 2^63 − 1

4.1 Undo/Redo

When nuclear accidents happen, investigators need to be able to replay the circumstances under which the accident occurred. In such cases, the system must support a simulation facility. Our nuclear customer requires that we thus support an *undo/redo* facility. This facility shall *undo* up to the last invocation of the *new_tracker* user command. The acceptance test *at2.txt* is an example that checks this simulation facility.

5 Directory structure of your project

Your project shall reside in a directory **tracking-project** with the following structure:

```
tracker-project/
├── docs
│   ├── tracker.pdf
│   └── team.txt
├── tracker.ecf
├── root
│   └── root.e
└── tests
    ├── acceptance
    │   ├── instructor
    │   │   └── at1.txt
    │   └── student
    │       ├── at1.txt
    │       ├── at2.txt
    │       ├── at3.txt
    │       └── at4.txt
    └── unit
        ├── instructor
        └── student
```

You must submit a **superset** of the above directory structure (i.e. you may add additional folders and files) so that we can compile your project. Note that you will be submitting additional directories such as *tracker* (with your model, etc.), *generated_code*, etc.

In the *docs* folder you must submit

- *tracker.pdf*: this is the report of your team, professionally prepared
- *team.txt*: is the Prism logins of the members of your team

The *team.txt* file looks like this:

```
   cse99783
   cse67999
```

The Prism login on the first line of the *team.txt* file shall also be the login for the team submission. If *team.txt* is not precisely as specified, you will not receive a grade. You may work on your own (in which case there is only one login) or in a team of at most two members.

You shall write at least four acceptance tests of your own, which appear in the *student* directory. (The test in the *instructor* directory is the one that is given to you by the instructor). It is advisable that you also write unit tests to test your code, but no lower limit is supplied.

You shall ensure that there are no EIFGENs directories in your submission, i.e. you must eclean before you submit.

6 How to submit your report and project

There is:

- (a) an electronic submission via the *submit* command
- (b) you must also submit your report *tracker.pdf* at the Moodle site by the due date.

Electronic submission:

```
submit -l 3311 project tracker-project
```

Please wait patiently for the submit script to check your submission and terminate normally. To help you, we check that your file structure is correct, that your project compiles and succeeds on the tests that we provided you with. If the check script reports errors, ensure that you fix those errors. Use the scheduled labs and office hours to obtain help so that you submit a reliable software product. You may submit as many times as you like up to the deadline. We grade your last submission.

7 The structure of your report

Your report must follow the 3311 SDD structure template documented at
https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:sdd:start

Please read all the details at the above URL carefully.

Your document must show which **contracts** you developed to support the safety critical **specification** goal (Goal 3) mentioned in section 3.

8 Modes of failure

We grade your project by:

- Testing your code submission for correctness
- Evaluating your design in the report that you provide

We test your project for correctness by running our own suite of acceptance tests. The behavior of your system must be **character-by-character** the same as the Oracle specification. The following problems are serious and will impact on your ability to obtain a passing grade:

- Your project does not meet the required submission guidelines and directory structure
- Your project does not compile
- An exception is generated when running one of our acceptance tests
- Your system does not terminate when running one of our acceptance tests
- Your system does not provide the correct status messages when running one of our acceptance tests
- The messages are correct but the business logic is incorrect (i.e. the output is wrong either in a detail or in the macro sense)

Given that this is a mission critical application, you must ensure the correctness of your system and demonstrate that you have taken sufficient precautions to ensure its correctness.

9 Appendix

Software designers are experts at developing software products that are correct, robust, efficient and maintainable. Correctness is the ability of software products to perform according to *specification*. Robustness is the ability of a software system to react appropriately to abnormal conditions. Software is maintainable if it has a well-designed *architecture* according to the principles of abstraction, modularity, and information hiding. We expect your software product to demonstrate these attributes for a passing grade.

It is crucial that you do **regression testing** of your software product before submission. You may submit as many times as needed. The last submission is graded. Consult the many lecture notes on testing, test driven development via **unit testing**, and regression testing via **acceptance tests**. A design that is not correct is not a good design.

Below, we repeat some thoughts on the importance of *specifications* posted on the course forum.[7]

Q: All throughout your writings you emphasize the necessity not only of coming up with sound algorithms for solving problems, but also of proving formally, mathematically, their

[7] http://www.budiu.info/blog/lamport.html

correctness. Not just using program testing, but using mathematical reasoning for all possible circumstances. However, the currently available formal methods are unable to prove correctness of large software systems, such as real operating systems. What is your advice to software developers for bridging this gap between algorithms (which can be analyzed) and full software systems?

A: You seem to be implicitly asserting the usual argument that program verification cannot prove the correctness of a complete operating system [*ed*: actually this has now been done] , so it is useless for real systems. The same reasoning would hold that because you can't implement a complete operating system in C (since you need a fair amount of assembly code), C is useless for building real systems. While this argument has obviously never been applied to C, it has in fact been used to dismiss any number of other programming languages.

People fiercely resist any effort to make them change what they do. **Given how bad they are at writing programs, one might naively expect programmers to be eager to try new approaches**. But human psychology doesn't work that way, and instead programmers will find any excuse to dismiss an approach that would require them to learn something new. On the other hand, they are quick to embrace the latest fad (extreme programming, templates, etc.) that requires only superficial changes and allows them to continue doing things basically the same as before. In this context, it is only fair to mention that people working in the area of verification are no less human than programmers, and they also are very reluctant to change what they do just because it isn't working.

The fundamental idea behind verification is that one should think about what a program is supposed to do before writing it. Thinking is a difficult process that requires a lot of effort. Write a book based on a selection of distorted anecdotes showing that instincts are superior to rational judgment and you get a best seller. Imagine how popular a book would be that urged people to engage in difficult study to develop their ability to think so they could rid themselves of the irrational and often destructive beliefs they now cherish. So, trying to get people to think is dangerous. Over the centuries, many have been killed in the attempt. Fortunately, when applied to programming rather than more sensitive subjects, preaching rational thought leads to polite indifference rather than violence. However, the small number of programmers who are willing to consider such a radical alternative to their current practice will find that thinking offers great benefits. Spending a few hours thinking before writing code can save days of debugging and rewriting.

The idea of doing something before coding is not so radical. Any number of methods, employing varying degrees of formalism, have been advocated. Many of them involve drawing pictures. The implicit message underlying them is that these methods save you from the difficult task of thinking. If you just use the right language or draw the right kind of pictures, everything will become easy. The best of these methods trick you into thinking. They offer some incentive in the way of tools or screen-flash that sugar coats the bitter pill of having to think about what you're doing. The worst give you a happy sense of accomplishment and leave you with no more understanding of what your program is supposed to do than you started with. The more a method depends on pictures, the more likely it is to fall in the latter class.

At best, a method or language or formalism can help you to think in one way. And there is no single way of thinking that is best for all problems. I can offer only two general pieces of advice on how to think. The first is to write. As the cartoonist Guindon once wrote, "writing is nature's way of showing you how fuzzy your thinking is." Before writing a piece of code, write a description of exactly what that piece of code is supposed to accomplish. This applies whether the piece is an entire program, a procedure, or a few lines of code that are sufficiently non-obvious to require thought. The best place to write such a description is in a comment.

People have come up with lots of reasons for why comments are useless. This is to be expected. Writing is difficult, and people always find excuses to avoid doing difficult tasks. Writing is difficult for two reasons: (i) writing requires thought and thinking is difficult, and (ii) the physical act of putting thoughts into words is difficult. There's not much you can do about (i), but there's a straightforward solution to (ii) — namely, writing. The more you write, the easier the physical process becomes. I recommend that you start practicing with email. Instead of just dashing off an email, write it. Make sure that it expresses exactly what you mean to say, in a way that the reader will be sure to understand it.

Remember that I am not telling you to comment your code after you write it. You should comment code before you write it.

Once you start writing what your program is supposed to do, you will find that words are often a very inconvenient way to express what you want to say. Try describing the mean of a set of numbers in words. If you succeed in describing it precisely and unambiguously, you'll find that you've written a formula as a sentence. This is a silly thing to do. In a great many cases, mathematics is a much more convenient language than English for describing what a piece of code is supposed to do. However, it is only a convenient language if you are fluent in it. You are undoubtedly fluent in arithmetic. You have no trouble understanding

The mean of the numbers a1, … , an equals (a1 + … + an) / n.

The kinds of things you need to describe in programming require more than simple arithmetic. They also require simple concepts of sets, functions, and logic. You should be as familiar with these concepts as you are with arithmetic. The way to achieve familiarity with them is the same way you did with arithmetic: lots of practice.

As you get better at using math to describe things, you may discover that you need to be more precise than mathematicians usually are. When your code for computing the mean of n numbers crashes because it was executed with n = 0, you will realize that the description of the mean above was not precise enough because it didn't define the mean of 0 numbers. At that point, you may want to learn some formal language for writing math. But if you've been doing this diligently, you will have the experience to decide which languages will actually help you solve your problem.