CITS5501 2020 project

Version: 0.2
Date: 5 May 2020
Please check the CITS5501 website to ensure that you have the latest version.

The goal of this assignment is to assess your understanding of class and unit testing, syntax-based testing, and Alloy modelling.

- The assignment contributes **35%** towards your final mark this semester, and is to be completed as individual work. It is marked out of 35.
- The deadline for this assignment is **23:59 pm, Sunday 31st May**.
- The assignment is to be done individually.
- Submit your assignment as either one zipped file, or multiple individual files, using **cssubmit**. Please do not use other archive file formats (e.g. `7z`, `.dmg`, `.rar`, `.sqx`).
- Your submission should contain a PDF report containing answers to questions 1–2, plus an Alloy "`.als`" file with your answer to question 3.
- You are expected to have read and understood the University Guidelines on Academic Conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.
- You must submit your project before the submission deadline above. There are significant Penalties for Late Submission (click the link for details).

You are part of the software development team for Exron, a power generation company. Specifically, you work in the trade division, which trades in the energy resources market (electricity, gas, and carbon emissions).

The company trades in these commodities just as investors might trade in shares or futures – for example, a power producer (company A) could agree on a contract with another company (company B) to provide 500 Megawatts-hours for a given price in 10 months' time; and company B can sell its rights under the contract on the commodities market. Your company can buy and sell contracts in order to ensure it has enough supply for customers, or to make profit from excess supply.

Your team has been tasked with developing a new interface to the company's existing trading systems, which will allow traders to specify transactions to make using a specially-designed programming language (that is, a *domain-specific language*).

1. Classes and unit tests

Consider the following partially-specified classes:

- Interface `TradingOrganization`.

  Has getter and setter methods for:

  - `organizationID`, a `String`
  - `registeredAddress`, a `String`
  - `name`, a `String`
  - `emailAddress`, a `String`

- Interface `Formula`.

  Has a method:
  `double calculateResult()`

- class `Contract`

  Has instance variables:

  - `Date deliveryStartDate`
  - `Date deliveryEndDate`

- class `DieselContract`

  - Inherits from `Contract`.

  Has instance variables:

  - `ContractStorageDatabase db`
  - `double gallonsPerDay`
  - `TradingOrganization otherContractParty`
  - `Formula priceCalculationFormula`

  Has a method:
  `void finalizeContract()`.

When `finalizeContract` is executed on a properly initialized `DieselContract` object, the details of the contract are stored in a persistent database (the `db`) instance variable, and electronic authorization of the contract is performed over the network using the `otherContractParty`'s email address.

Answer the following questions. If you need to make any assumptions, state what they are.

a. Identify two plausible *class invariants*, each of which applies to one of these classes. (If both your invariants apply to the same class, that is fine.)

   Explain what the invariants are. What implications would it have for the behaviour of the system if the invariants were broken?

   ($\frac{1}{4}$ to $\frac{1}{3}$ page) [**5 marks**]

solution:
Plausible invariants are:

- That instance variables (`deliveryStartDate`, `deliveryEndDate`, `db`, `otherContractParty`, `priceCalculationFormula`) should not be null, for their respective classes.
  (This is usually assumed to be the case for Java code, but does qualify as an acceptable invariant.)
- Assuming classes are implemented that have instance variables for `organizationID`, `registeredAddress`, `name`, and `emailAddress` (for the `TradingOrganization` interface), that these should not be null.
- That the `organizationID` of an object implementing `TradingOrganization` must be distinct from all other organization IDs.
- That `deliveryStartDate` must be no later than `deliveryEndDate`, for `Contract`.
- `gallonsPerDay` must not be 0 (for `DieselContract`)
- Depending on the assumptions made: That `gallonsPerDay` must not be negative. (Alternatively: this could represent a contract to receive, rather than to supply.)
- email addresses must be valid addresses, complying with relevant specifications

If class invariants are broken, that means an object is now in an *erroneous* or *inconsistent* state – it is semantically "meaningless". For objects to operate correctly requires that they be in a consistent state, so we can now no longer rely on the correct operation of those objects (and nor, consequently, on behaviour of the system).

b. If writing a JUnit unit test for the `finalizeContract` method, what *test fixtures* might be required? Give at least two examples.

Sketch out code for a test class plus fixtures using a language and testing framework of your choice (Java or Python; if using another language, check with the unit coordinator first).

You do *not* need to write the actual tests; just class instance variables and any fixture-related code.

(Up to a page) [**5 marks**]

solution: fixtures:
Test fixtures include any *state* that must be set up for the test. Since this is a unit test, the fixtures will typically be instance variables of the test class, and may be *mocks*, but certainly shouldn't be e.g. connections to live databases – those are too slow for unit tests.

So, examples of plausible test fixtures are (depending on exactly what the test is, and how it is written):

- values for `deliveryStateDate` and `deliveryEndDate`
- a mock value for `db`
- a value for `otherContractParty` (including all necessary state for that value); could be a mock
- a value for `priceCalculationFormula`; probably a mock

solution: test class
A typical test class might look something like this:

```java
class DieselContractTest {

  // assume we have fixtures for db, otherContractParty,
  // and priceCalculationFormula
  ContractStorageDatabase testDB;
  TradingOrganization testParty;
  Formula testFormula;

  // setUp method assuming the Mockito framework is
  // used - pseudocode is fine
  @Before
  public void setUp(){
    testDB = mock(ContractStorageDatabase.class);
    testParty = mock(TradingOrganization.class);
    testFormula = mock(Formula.class);

    // code to specify behaviour of mocks goes here
  }

  @After
  public void tearDown(){
    // good practice to set fixtures to null
    testDB = null;
    testParty = null;
    testFormula = null;
  }

  // actual tests would go here
}
```

c. For the fixtures you described in question 1(b): if you were writing an integration test, would your approach be any different? Explain how and why.

($\frac{1}{4}$ to $\frac{1}{3}$ page) **[5 marks]**

solution:
If writing an integration test, we aim to test that multiple classes work correctly together.

So for each integration test, some of the objects that would have been *mocks* in a unit test, will be replaced by real objects of that class.

Some of our tests might test that our Java code works correctly with the database. For such tests, we would use a real database (though one containing fake data - i.e. *not* a "live", production database).

Likewise, at some point we will went to test integration with the email system, so rather than a "mock" object representing a mail server, we would actually connect to a mail server (albeit one that doesn't send actual emails to external addresses).
[See e.g. https://mailtrap.io/ for an example of a company providing such a service.]

d. If you were designing tests for the `finalizeContract()` method using input space partitioning, what would the *parameters* be? Identify at least two *characteristics* you could use for performing input space partitioning. Explain why they are appropriate, and how you would use them in writing unit tests.

($\frac{1}{2}$ to 1 page) **[5 marks]**

solution:
The *parameters* consist of all state and values that must be supplied in order to make a call to `finalizeContract`.

Thus, this includes the values for the `deliveryStartDate`, `deliveryEndDate`, `db`, `gallonsPerDay`, `otherContractParty` and `priceCalculationFormula` instance variables. (Assuming these are the only non-local variables used by the method.)

Many characteristics are possible, but some plausible characteristics include:

- Difference between `deliveryStartDate` and `deliveryEndDate` (0 days, 1 day, more than 1 day).

Because errors often happen around "boundaries":

- `deliveryStartDate` is the first day of a week
- `deliveryStartDate` is the first day of a month
- `deliveryStartDate` is the first day of a year

(And similarly for `deliveryEndDate`, and for the *last* day of a week, month or year.)

Assuming negative `gallonsPerDay` is acceptable (see previous answers):

- sign of `gallonsPerDay`, positive or negative

Note that the above characteristics partition up the *valid* values of each parameter - the normal case.

How we treat invalid values depends on the assumptions made.

If there are preconditions that state the result is *undefined* for invalid values - then we don't write tests for them (because no behaviour is specified, so no test can be written).

If on the other hand there are preconditions stating that an *exception* will be thrown, then we should write tests for that, to ensure the correct exceptions are thrown.

In general, we don't bother to write tests for null values, since in Java systems we normally assume that null is not a valid value, and that the Java runtime itself will throw a `NullPointerException`. (If we wrote a test for that, we'd be testing the Java runtime - not our code - and there's usually little point in doing so.)

To use our characteristics in tests: the input values for each test would consist of some combination of partitions from our various characteristics (excluding infeasible combinations).

2. Contracts specification language

Pricing formulas for contracts can be specified by traders using a dedicated language. A fragment of the BNF syntax for the language is:

```
<quantityVar> ::= "diesel" <grade> | "windpower" <grade>
                  | "coalpower <grade>"
<operator>    ::= "+" | "*" | "-" | "/"
```

```
<builtinfunc> ::= "min(" <quantityVar> "," <dateRange> ")"
                    | "max(" <quantityVar> "," <dateRange> ")"
                    | "avg(" <quantityVar> "," <dateRange> ")"
<expression>  ::= <builtinfunc> <operator> <builtinfunc> |
                  <builtinfunc> <operator> "(" <expression> ")"
```

We assume that `<grade>` and `<dateRange>` are specified elsewhere, and here we will assume they are terminal symbols. (Or, if you like, you can assume they have only one possible production.)

a. Write down two sample strings from the language defined by this grammar. Given our assumptions, explain whether it is plausible to write exhaustive tests for the language.

   ($\frac{1}{3}$ to $\frac{1}{2}$ page) [**5 marks**]

solution:
Many are possible, but here are two:

```
min(diesel <grade>, <dateRange>) + min(diesel <grade>, <dateRange
  ↪ >)

min(diesel <grade>, <dateRange>) + max(diesel <grade>, <dateRange
  ↪ >)
```

Here we've just left `<grade>` and `<dateRange>` in the final string, but we could replace them with any string we like – we are assuming they have just one production, so:

```
min(diesel A, 01/01/2020-02/01/2020) + min(diesel A,
  ↪ 01/01/2020-02/01/2020)

min(diesel A, 01/01/2020-02/01/2020) + max(diesel A,
  ↪ 01/01/2020-02/01/2020)
```

It is *not* possible to write exhaustive tests for this grammar, because an `<expression>` can grow without limit: the second production of `<expression>` means that given any expression, we can always construct a "larger" one. So this is an infinite-sized language and cannot be tested exhaustively.

b. If you wanted to achieve Terminal Symbol Coverage (TSC), how many tests would be needed? What about for Production Coverage (PDC)? Show any working and explain your reasoning.

   ($\frac{1}{2}$ to 1 page) [**5 marks**]

solution:
The number of terminals is:

- for `<quantityVar>`: 3 "normal" terminals, plus `<grade>` (deemed to be a terminal) gives 4 total.
- for `<operator>`: 4
- for `<builtinfunc>` and `<expression>`: 5 "normal" terminals (assuming duplicate terminals count as just one "token"), plus `<dateRange>` (deemed to be a terminal) gives 6 total.

So, 14.

The number of productions is:

- for `<quantityVar>`: 3
- for `<operator>`: 4
- for `<builtinfunc>`: 3
- for `<expression>`: 2

So, 12.

3. Formal methods

Write code for signatures and facts in Alloy which will do the following:

- Declare the existence of a "diesel contract" type.
- Declare the existence of a "formula" and "trading organization" types.
- Declare appropriate relationships and cardinalities for the above types.

Submit your work as an Alloy ("`.als`") file. [**5 marks**] Include explanatory comments in your model code.

solution:
Depending on the assumptions, there are many possibilities, but the minimal code for declaring these entities would be:

```
sig DieselContract {
      formula : one Formula,
      otherContractParty: one TradingOrganization
}
sig Formula {}
sig TradingOrganization {}
```

What other attributes we add depends on our assumptions about what we're trying to model. Since dates turn up in our tests quite a bit, we might want to model those. So we might add a sig `Date`, and amend `DieselContract`:

```
sig Date {}

sig DieselContract {
      formula : one Formula,
      otherContractParty: one TradingOrganization
      deliveryStartDate: Date
      deliveryEndDate: Date
}
```

Or we might model dates as ints. (Though if you do a little research, you will discover that a better solution is to model dates as an *ordered type*: `open util/ordering[Date]`.

If we do that, we can then add constraints such as:

```
fact endAfterStart {
  all c : DieselContract | c.deliveryEndDate >= c.deliveryStartDate
}
```

Report requirements

Your report should be in PDF format, and use A4 size pages. The font for body text should be between 9 and 12 points. It should contain numbered headings, with useful heading titles. Any diagrams, charts or tables used must be legible and large enough to read. All pages (except the cover, if you have one) should be numbered. If you give scholarly references, you may use any standard citation style you wish, as long as it is consistent. Cover sheets, diagrams, charts, tables, bibliographies and reference lists do not count towards any page-count maximums.