

EECS3311: Lab3

Design of a Citizen Registry

January 29, 2017

Contents

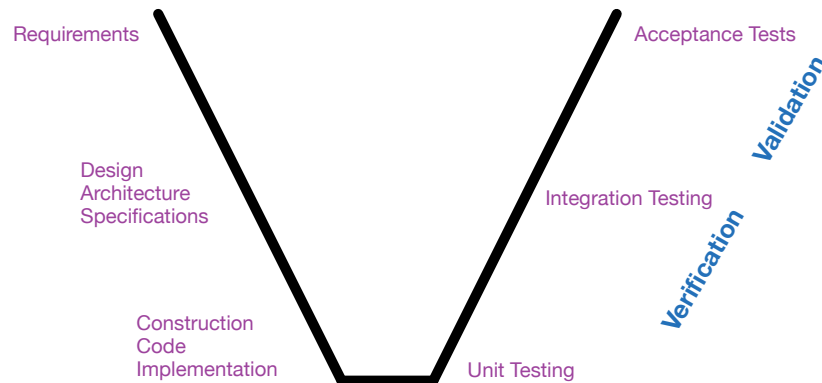
1	Learning outcomes – design of a citizen registry	3
2	What your design must achieve	4
3	Getting started	7
3.1	Use ETF to generate the starter code	7
3.2	Compile the project and run the acceptance tests	8
3.3	Develop the business logic and test it	8
4	How to submit	9
5	Design Decisions	11
6	Appendix	12
A	Validation and Verification in Software Engineering	12
B	Requirements for Citizen Registry	13
C	Analysis	14
C.1	Abstract user interface	14
D	Eiffel Testing Farmework – ETF	15
E	Design (Architecture, Specifications and SRP)	16
F	Report	16

List of Figures

1	Response of registry system to Acceptance Test at1.txt	5
2	Input Grammar for Citizen Registry App	6

3	Directories and files generated by the ETF command	7
4	ROOT class has a switch <i>unit_test</i>	9
5	Error conditions displayed to the user at the console in order they are raised	10
6	The V-model in Software Engineering	12

1 Learning outcomes – design of a citizen registry



The *requirements* for a citizen registry application are provided in the appendix of this document. Given these requirements for the registry problem, in this Lab, you will:

1. Design the business logic (in the cluster *model*) for the registry application.
 - Describe an modular architecture with clean interfaces. BON class diagrams can be used to document the architecture.
 - Decide on what the relevant modules (classes) are, their roles in the system and their operations (features).
 - Describe the relationships between the modules (e.g. client supplier, inheritance, etc.) so as to preserve modularity and information hiding.
2. Provide specifications of module behaviour — free of implementation detail, via expressive preconditions, postconditions and class invariants.
3. Provide ESPEC unit tests that will demonstrate that implementations of modules are working to specification, and that the modules co-operate together to achieve the relevant functionality (integration testing).
4. Implement the design and run the unit tests to demonstrate that the design is correct (satisfies its specification).
5. You must also demonstrate to the customer that the design satisfies their *requirements*. To do this, you use the ETF framework (described in the accompanying document¹). The framework will provide you with the ability to exercise your application with input-output blackbox acceptance tests as if you were running it from a user interface.
6. Finally you will submit your work for grading.

¹*Eiffel Testing Framework for Input-Output Acceptance Testing*, see `etf.pdf`.

2 What your design must achieve

In the directory `/eecs/dept/course/2015-16/W/3311/labs/lab3` on Prism, there are 4 files:

```
red> pwd
/eecs/course/3311/labs/Lab3
red> ls
at1.txt  at2.txt  registry  registry.definitions.txt
at1.expected.txt  at2.expected.txt
```

- **registry** is a finalized Eiffel executable that implements the citizen registry requirements using ETF. You use it as an oracle, i.e. it acts as your “always available customer” – describing their requirements to you and the way they expect the system you are designing to behave, from their point of view.
- **at1.txt** and **at2.txt** are two acceptance tests for the application. Working together with the customer, you have agreed that these are two use cases that can check that the final system you deliver — behaves according to their requirements. **at1.expected.txt** is the expected behaviour of the program you must design and should match the oracle.
- **registry.definitions.txt** is the ETF input grammar describing operations that can be invoked from the user interface.

On Prism, if you invoke the executable at the console as follows:

```
<path>/registry -b at1.txt
```

you get the response at the console shown in Fig. 1. The “...” indicates that the actual output is much longer. On Prism, invoke the oracle so that you can see the complete output at the command line. You can also redirect the output to a file:²

```
<path>/registry -b at1.txt > at1.actual.txt
```

Your job is to design a system in Eiffel that executes like the **registry** oracle in the above example. The “-b” flag is the batch mode. There is also a “-i” interactive mode. You can also run the oracle on the second acceptance test to see what output behaviour it produces.

As you can see, the ETF framework provides you with the ability to exercise your application with input-output blackbox acceptance tests as if you were running it from a user interface. Your customer (the officials at Passport Canada), knowing nothing about computer languages, can understand these tests and help you to write them. These acceptance tests are thus very different from the ESpec unit tests.

²Copy the course directory to your own account as you cannot save files in the course directory.

```

    ok
->put(1,"Joe",[15, 2, 1990])
    ok
    Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Single
->put(2,"Pam",[31, 3, 1991])
    ok
    Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Single
    Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Single
->marry(1,2,[8, 12, 2010])
    ok
    Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
    Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]
->put_alien(3,"Bob",[17, 2, 1995],"England")
    ok
    Bob; ID: 3; Born: 1995-02-17; Citizen: England; Single
    Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
    Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]
->put_alien(4,"Kim",[31, 3, 1991],"France")
    ok
    Bob; ID: 3; Born: 1995-02-17; Citizen: England; Single
    Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
    Kim; ID: 4; Born: 1991-03-31; Citizen: France; Single
    Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]
    ...

```

Figure 1: Response of registry system to Acceptance Test at1.txt

ETF Input Grammar

The key to producing acceptance tests in ETF, is to define an input grammar. You get to this grammar by eliciting requirements from your customer (see Appendix).

The file `registry.definitions.txt` describes the input grammar for the citizen registry with five operations (*put*, *put_alien*, *marry*, *divorce*, and *die*) as shown in Fig. 2.

The ETF framework uses the input grammar to generate starter code that takes care of parsing and handling the input/output from the user interface (the console). This leaves the software developer free to focus on the design and construction of the business logic. The design of the various user interfaces is thereby decoupled from the design of the application. The design of the concrete user interfaces will anyway require the special expertise of HCI.

```

-- input commands for citizen registry
system registry

--date
type DAY = 1..31
type MONTH = 1..12
type YEAR = 1900..3000
type DATE = TUPLE[d:DAY;m:MONTH;y:YEAR]

type ID = INT
type NAME = STRING
type COUNTRY = STRING

put(id: ID; name1: NAME; dob: DATE)
    -- citizen id, name1, date of birth

put_alien(id: ID; name1: NAME; dob: DATE; country: COUNTRY)
    -- alien id, name, date of birth, country of citizenship

marry(id1: ID; id2: ID; date: DATE)
    -- register marriage of person id1 to id2 on given date
divorce(a_id1:ID; a_id2:ID)

die(id:ID)
    -- death of person id

```

Figure 2: Input Grammar for Citizen Registry App

3 Getting started

3.1 Use ETF to generate the starter code

Create a directory in your Prism account (or on your SEL-VM) called **registry**, and execute the “etf” command on Prism to generate the starter code defined by the input grammar in the **registry** directory:

```
mkdir registry
etf -new registry.definitions.txt registry
```

The “etf” command will generate the starter code in directory **registry**. The directory structure and files generated is shown in Fig. 3.

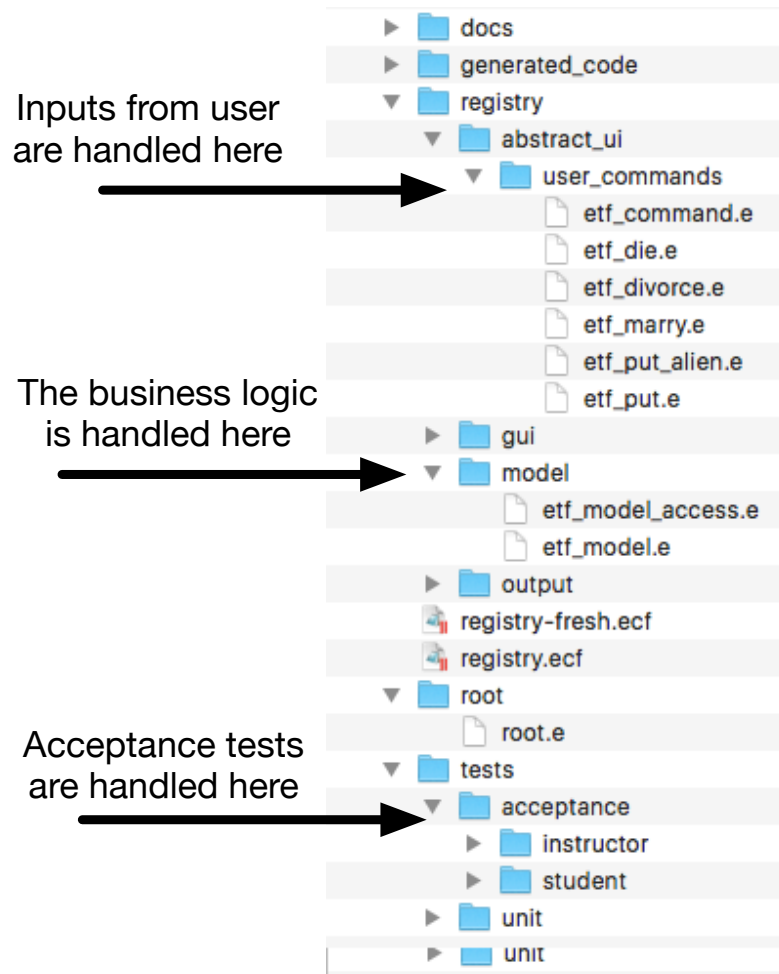


Figure 3: Directories and files generated by the ETF command

It is required that you preserve the generated directory structure in your final submission.

The business logic you develop must be placed in the *model* cluster and your tests in the *tests* cluster.

3.2 Compile the project and run the acceptance tests

When you first compile the project, and execute, a GUI pops up. See the ETF documentation.³ However, we will execute in console mode. So change the setting in the root file to

```
class ROOT ...
  switch: INTEGER
    -- Running mode of ETF application.
  do
    Result := etf_cl_show_history
  end
```

and follow the ETF documentation in the footnote³ for precise instructions. Set up a symbolic link to the EIFGEN executable as explained in the ETF documentation (e.g. let the symbolic link be **registry.exe**) and execute the acceptance test as:

registry.exe -b tests/acceptance/student/at1.txt.

```
>registry -b at1.txt
ok
->put(1,"Joe",[15, 2, 1990])
System state: default model state (1)
->put(2,"Pam",[31, 3, 1991])
System state: default model state (2)
->marry(1,2)
System state: default model state (3)
->put_alien(3,"Bob",[17, 2, 1995],"England")
System state: default model state (4)
->put_alien(2,"Kim",[31, 3, 1991],"France")
System state: default model state (5)
->marry(3,4)
```

3.3 Develop the business logic and test it

You can now:

- Develop the business logic for the citizen registry application.
- Get the two acceptance tests at1.txt and at2.txt working via ETF.
- Write unit tests for the business logic as required. Put your own tests in **tests/unit/student**.
- Write your own acceptance tests. Put your own tests in **tests/acceptance/student**.

See the accompanying document for how to start developing the model — the business logic.

³<https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:etf:start>

Your business logic goes in the directory `model` and you can add classes (and clusters/sub-directories) as needed in this directory. You will also need to complete the input commands in the directory `user-commands`. You might also revise class `output_handler.e` to achieve the correct output. Don't change any other part of the generated code.

Ensure that your output matches that of the oracle character for character, as we will do a “diff” to check and grade your project.

In the `ROOT` class (see Fig. 4), you can switch between unit testing and acceptance testing by setting the `unit_test` flag.

```
class ROOT inherit ES_SUITE ...

  switch: INTEGER
    -- Running mode of ETF application.
  do
    Result := unit_test
  end
```

Figure 4: `ROOT` class has a switch `unit_test`

4 How to submit

Ensure that you have the specified directory structure, otherwise your project will not be graded.

In addition, you must add a report *Report.pdf* in the *doc* folder of the *registry* project. See Appendix F for the contents of the report. You must place a copy of this report it in the course Dropbox (via the course Moodle site).

When you have finished developing your project, freeze it (and check that all your unit tests pass) and that your acceptance tests match the oracle character-for-character. You can compile your project to a finalized executable (like our oracle) as follows:

```
ec15.12 -c_compile -finalize -config registry.ecf
```

- Ensure that your acceptance tests are character for character the same as the oracle.
- `eclean` your directory `registry`
- `submit 3311 Lab3 registry`

```

put(id: ID; name1: NAME; dob: DATE)
    err_id_nonpositive
    err_id_taken
    err_name_start
    err_invalid_date

put_alien(id: ID; name1: NAME; dob: DATE; country: COUNTRY)
    err_id_nonpositive
    err_id_taken
    err_name_start
    err_invalid_date
    err_country_start)

marry(id1: ID; id2: ID; date: DATE)
    err_id_same
    err_id_nonpositive
    err_invalid_date
    err_id_unused
    err_marry

divorce(a_id1:ID; a_id2:ID)
    err_id_same
    err_id_nonpositive
    err_id_unused
    err_divorce

die(id:ID)
    err_id_nonpositive
    err_id_unused
    err_dead_already

err_id_nonpositive: STRING = "id must be positive"
err_id_unused : STRING = "id not identified with a person in database"
err_id_same : STRING = "ids must be different"
err_id_taken: STRING = "id already taken"
err_name_start: STRING = "name must start with A-Z or a-z"
err_country_start: STRING = "country must start with A-Z or a-z"
err_invalid_date: STRING = "not a valid date in 1900..3000"
err_marry: STRING = "proposed marriage invalid"
err_divorce: STRING = "these are not married"
err_dead_already: STRING = "person with that id already dead"

```

Figure 5: Error conditions displayed to the user at the console in order they are raised

5 Design Decisions

You will need to describe the classes and their features as well as how the classes are related. The appendix has a BON diagram that might help you get started. For example, the BON architecture describes classes such as *REGISTRY*, *PERSON* etc.

In these classes, you can use contracts to ensure the consistency of the database. For example, in class *PERSON* you can specify that the spouse of the current person is the current person themselves. There are many such consistency specifications in this application.

You have already used *ARRAY* and *LIST*. The suggestion is that you use *HASH_TABLE* (like *MAP* in Java) to store a collection of persons in the registry by ID. For example, in class *REGISTRY* you might have:

```
person: HASH_TABLE [PERSON, ID]
```

where *ID* is *INTEGER*. Thus $person : ID \rightarrow PERSON$, i.e. it is partial function because not all integer ids have an associated person. Given that *person* is a partial function, preconditions are needed to access a person in the registry. Note that *HASH_TABLE* has been equipped with the array notation and the “across” iterator. The array notation allows you to check that a person “Pam” has *id* = 7 as follows:

```
pam: PERSON
...
check pam ~ person[7] end
```

Class *REGISTRY* is also where you would support basic operations such as *put*, *marry* etc. that will be invoked from the ETF command classes *PUT*, *MARRY* etc. In *REGISTRY*, these operations have strong preconditions to ensure the consistency of the data. Invariants can also help ensure that the data is consistent.

The generated classes such as *PUT*, *MARRY* etc. is where you receive input from the console. This is where you need defensive programming to ensure that you handle error conditions that are not checked by the input grammar. For example, two people who are underage may not marry and a suitable error message must be generated at the console to inform the user that the requested operation did not go through.

One of the design decisions you must make is how to handle error conditions in your implementation.

Of course, you also need to know what the various error conditions are. In consultation with Passport Canada, the error messages are shown in Fig 5. The errors are listed in the order they are displayed to the user at the console.

In all cases where there is a divergence between the above and the oracle, the output of the oracle in the course directory governs.

Note that you are free to use the *mathmodels* library to help specify the design, if you so choose.

6 Appendix

A Validation and Verification in Software Engineering

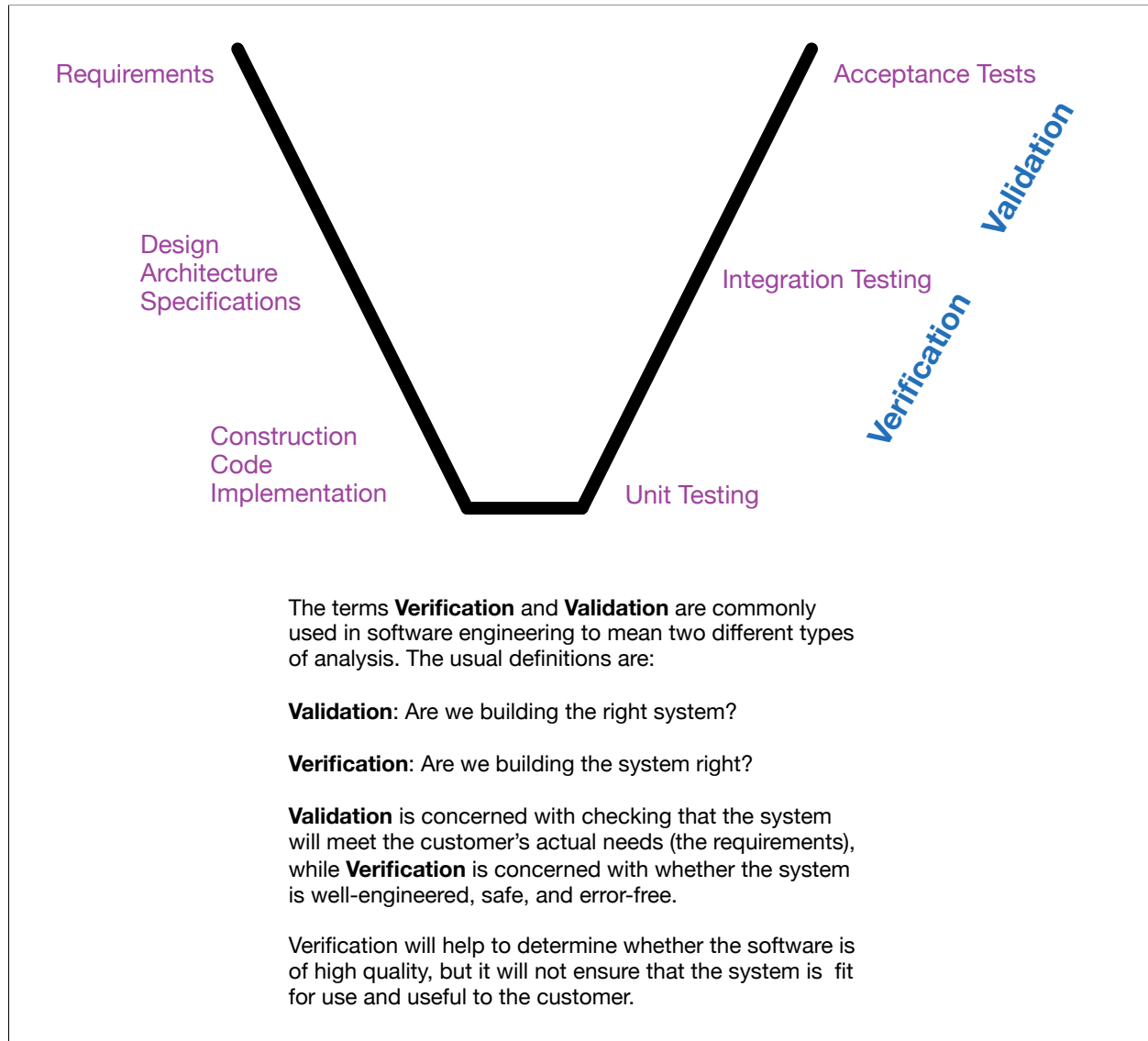


Figure 6: The V-model in Software Engineering

Bertrand Meyer writes:

The worldview underlying the Eiffel method ... [treats] the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them.

Anyone who has worked in both specification and programming knows how similar the issues are. Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility.

The same kinds of ideas, such as an object-oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations. One of the principal arguments for this approach is that it supports change and reuse. If everything could be fixed from the start, maybe it could be acceptable to switch notations between specification and implementation. But in practice specifications change and programs change, and a seamless process relying on a single notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels. (Bertrand Meyer,)

The Eiffel Testing Framework (ETF) adds the ability to introduce user requirements into a seamless process of software construction at a point before concrete user interfaces can be specified. ETF allows software developers to produce use cases that can be turned into acceptance tests, and that then free the developer to develop the business logic (the *model*) while not losing sight of the user requirements. This allows requirements to become part of a seamless development from requirements to implemented code, and allowing change even at the level of requirements.

We illustrate this below with a small example.

B Requirements for Citizen Registry

Passport Canada would like to maintain a registry of the Canadian population, citizens as well as landed residents (*aliens*). Data in the registry includes, for each person, their name, an identity number, and who they are married to. Each person is provided with a unique positive ID number. Here is a list of citizens sorted alphabetically by name and then ID:⁴

```
Bob; ID: 3; Born: 1995-02-17; Citizen: England; Single
Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
Kim; ID: 4; Born: 1991-03-31; Citizen: France; Single
Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]
```

Officials at Passport Canada can add new names to the database together with their ID and date of birth (*dob*). In the case of a newly arrived alien, their country of citizenship

⁴Obviously, in a real application we will want to keep track of parents, children, and names will be divided into last name, first name, etc. To keep this example small, in order to illustrate the ETF framework, the database provides only minimal information. A name will thus be an arbitrary string that starts with A-Z or a-z.

is also recorded. Officials also record marriages, divorces and deaths. So the state of the database at some later instant might look as follows:

```
Ada, ID: 5, Born: 1979-07-07, Citizen: Canada, Spouse: (Gus,6,[1998-05-06])
Bob, ID: 3, Born: 1995-02-17, Citizen: England, Spouse: (Pam,2,[2014-06-25])
Gus, ID: 6, Born: 1980-05-06, Citizen: Canada, Spouse: (Ada,5,[1998-05-06])
Joe, ID: 1, Born: 1990-02-15, Citizen: Canada, Deceased
Kim, ID: 4, Born: 1991-03-31, Citizen: France, Deceased
Pam, ID: 2, Born: 1991-03-31, Citizen: Canada, Spouse: (Bob,3,[2014-06-25])
Zurphy, Rex, ID: 7, Born: 1900-02-01, Citizen: Canada, Single
```

The registry will also be accessed by provincial governments to check whether two citizens may marry. To marry, citizens (or aliens) must be 18 or older.

C Analysis

On further consultation with the officials at Passport Canada, it is discovered that officials will be entering and checking the data from distributed locations using web, mobile and desktop apps. The critical concern is the integrity of the data. Dates should be legal. If a person is married then their spouse should also be reflected as being married. When a person passes away, then their spouse is no longer married. The validity of marriages needs to be checked. Citizens must be 18 or older and not currently married.

C.1 Abstract user interface

A decision was made to focus on the business logic first (the *model*) and to create the user interfaces and relevant apps at a later time.

Nevertheless, an abstract representation of the various user interfaces is required:

- So that the business logic can be tested; and
- So that officials can supply use cases without the need to know a programming language. The use cases can eventually become part of the acceptance test suite.

In consultation with the officials, a possible use case is as follows:

```
ok
->put(1,"Joe",[15, 2, 1990])
ok
Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Single
->put(2,"Pam",[31, 3, 1991])
ok
Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Single
Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Single
->marry(1,2,[8, 12, 2010])
ok
Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]
```

```

->put_alien(3,"Bob",[17, 2, 1995],"England")
ok
Bob; ID: 3; Born: 1995-02-17; Citizen: England; Single
Joe; ID: 1; Born: 1990-02-15; Citizen: Canada; Spouse: Pam,2,[2010-12-08]
Pam; ID: 2; Born: 1991-03-31; Citizen: Canada; Spouse: Joe,1,[2010-12-08]

```

The above format shows an input operation performed by a user (for example by submitting a form from a webapp in which a new citizen “Joe” is added to the database) and then the resulting state of the database. A sequence of such operations and their effect forms the use case. The message “ok” indicates that the operation was successfully performed. If an operation cannot be performed due to invalid inputs, a meaning message can be displayed instead of “ok”.

This format represents possible interactions between the various user interfaces and the business logic in a format understandable to officials not familiar with programming languages.⁵

D Eiffel Testing Framework – ETF

The ETF framework provides strong support for specifying abstract user interfaces and their interactions with the business logic. Use cases can also be transformed into acceptance tests to check that the application is performing correctly according the requirements of the customer.

Please see <http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start> for a description of the framework and videos that illustrate its use. Note that the ETF framework uses the Singleton design pattern for accessing model elements.

It is possible that the user interface (e.g. web, mobile or desktop app) might do some basic checking before posting the form. For example, users might be offered a year in the range 1900..3000, a month in the range 1..12 and a day in the range 1..31. We may therefore assume that dates will arrive in a suitable format, e.g. [15, 2, 1990].

/ / (m/d/y) select

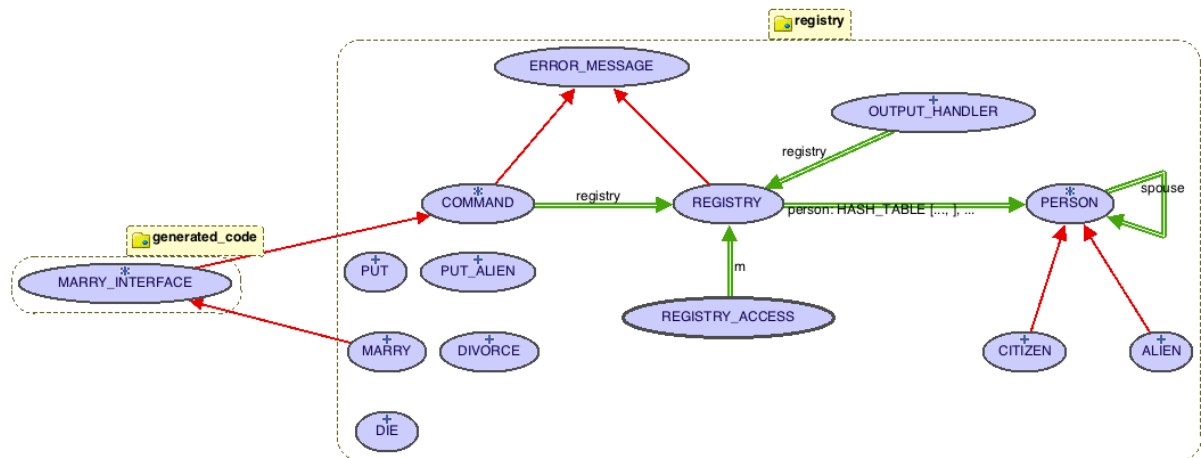
Even so, we may not be guaranteed that the dates are valid. For example, the date [31, 2, 1990] might satisfy the form checks in a webapp, but the date is still not legal. ETF provides the software developer with an *input grammar* that can be used to distinguish what checks are performed at the user interface and which checks must be done in the business logic. This separation of concerns ensures that the acceptance tests check relevant data without having to deal with possibilities that are guaranteed not to occur.

⁵Unit testing frameworks such as Junit or ESPEC are useful for testing modules and components of the system; however, they do not directly test the black-box input-output behaviour of the complete system from the perspective of a user. They also require knowledge of the relevant programming languages.

E Design (Architecture, Specifications and SRP)

The purpose of a *design* is to provide an *architecture* and a *specification* of the system free of implementation details.

A good architecture will decompose the system into modules or components and show the relationship between the various modules that is loosely coupled and where each module has a clear interface. The following diagram (although not correct) illustrates what a top level design diagram might look like (generated using the EiffelStudio/IDE):



Then each module is specified with a description that describes the behaviour of each of the operations from the point of view of a client. Dbc is one way of doing this where each class is provided with class invariants to ensure the integrity of the data and business logic and each operation is provided with preconditions and expressive postconditions.

As you choose your classes, there is an important design principle to keep in mind (among others mentioned in this course): *The Single Responsibility Principle* (SRP).

SRP states that a class should have one, and only one, reason to change. Each main responsibility should be in a separate class, because each responsibility is an axis of change. Each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the class to meet the others. **This kind of coupling leads to fragile designs that break in unexpected ways when changed.** See OOSC2 for class cohesion and coupling.

F Report

Your report must contain the following (using the LibreOffice or Visio templates for the BON diagrams in items 3 and 4):

1. Your Name and Prism Login
2. A statement describing what part of the Lab you did not complete and why?

3. Provide a top level BON diagram including solely the user commands (abstract UI) and model classes.
 - Include an initial diagram generated by the IDE, to demonstrate that you know how to use the IDE to generate a BON diagram. In this diagram, all the classes are compressed.
 - You must also include a LibreOffice/Visio version of the top level diagram, where some classes may have more (or less) detail than what can be achieved with the IDE. The idea is to provide an overview to other programmers what the main classes are and their relationship to each other.
4. Choose the most important class in your model from a contracting point of view and provide an uncompressed BON class diagram of the class (thus, not all features and contracts need be shown). For this class alone:
 - State (a) whether you have done an *excellent*, *average*, or *below average* job of applying information hiding for this class. (b) In one, or at most two paragraphs, provide evidence for your claim.
 - Do the same for the Single Responsibility Principle (SRP). State (a) whether you have done an excellent, average, or below average job of applying the SRP for this class. (b) In one, or at most two paragraphs, provide evidence for your claim.
5. For the same class as item 4, provide the RTF (IDE generated) contract view. You will need to edit the contract view to make it readable (e.g. no line wrapping, proper indentation required, etc.). See a Chapter 26 of OOSC2.
6. The report must have a professional appearance. Note that BON diagrams from LibreOffice/Visio must be suitably inserted into LibreOffice/MS-Word to ensure that the diagrams are clear and not fuzzy (so don't just take snapshots and hope that the results will be good). Visio and MS Office co-operate with each other so that you can insert the Visio image with full fidelity in the Word document. Likewise LibreOffice.