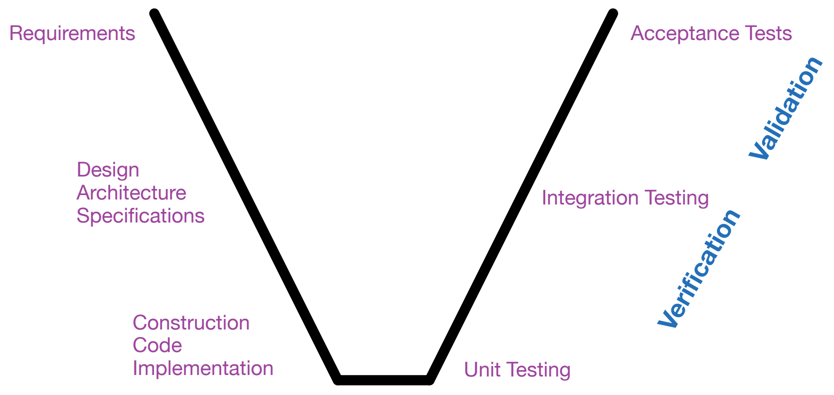
EECS3311: Lab3
Design for a Citizen Registry

January 28, 2018

Contents

List of Figures

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 a 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.
   - Provide specifications of module behaviour — free of implementation detail, via expressive preconditions, postconditions and class invariants.
     - Choose which features (commands and queries) of each module (i.e. each class) are in the API[1] (i.e. public) and which are private (for information hiding).
     - Document the API of each module (i.e. each class) by specifying the behaviour of each feature in the module via meaningful contracts using preconditions, postconditions and class invariants.

[1]In computer programming, an application programming interface (API) is a set of routine definitions, protocols, and tools for building application software, i.e. it is a set of clearly defined methods of communication between various software modules and components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer. An API may be for a web-based system, operating system, database system, computer hardware or software library. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables or remote calls. The design of an API has significant impact on its usability. The principle of information hiding describes the role of programming interfaces as enabling modular programming by hiding the implementation details of the modules so that users of modules need not understand the complexities inside the modules. Thus, the design of an API attempts to provide only the tools a client would need to use the module rather than the complete implementation of the module. The design of programming interfaces represents an important part of software architecture, the organization of a complex piece of software. `https://en.wikipedia.org/wiki/Application_programming_interface`.

  - The EiffelStudio IDE allows you to browse a class by *contract view*, full text view and flat and flat-short view. Inspect the contract view of your classes to see whether your specifications are precise and complete.

2. 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).

3. Implement the design and run the unit tests to demonstrate that the design is correct (satisfies its specification).

4. 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 document[2]). 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. These are called **acceptance tests** rather than **unit tests**. The acceptance tests check whether what you have designed is acceptable to the customer who has hired you to produce a reliable software product.

5. Finally you will submit your work for grading.

2 What your design must achieve

In the directory `/eecs/dept/course/2017-18/W/3311/labs/lab3` on Prism, there are 4 files:

```
red> pwd
/eecs/dept/course/2017-18/W/3311/labs/lab3
red> ls
at1.txt  at2.txt  registry  registry.input.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.

- `registry.input.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:

[2] *Eiffel Testing Framework for Input-Output Acceptance Testing*, see `etf.pdf`.

```
      <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:[3]

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

```
   init
->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

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

[3]Copy the course directory to your own account as you cannot save files in the course directory.

```
-- 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

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.

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.input.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.

To learn more about the ETF framework, see the help and videos provided at `http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start`.

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`. Run the "etf" command on Prism to generate the starter code defined by the input grammar.

```
       >etf   -new    registry.input.txt   registry
```

The "etf" command will generate the starter code in directory `registry`. The directory structure and files generated is shown in Fig. 3.[4]

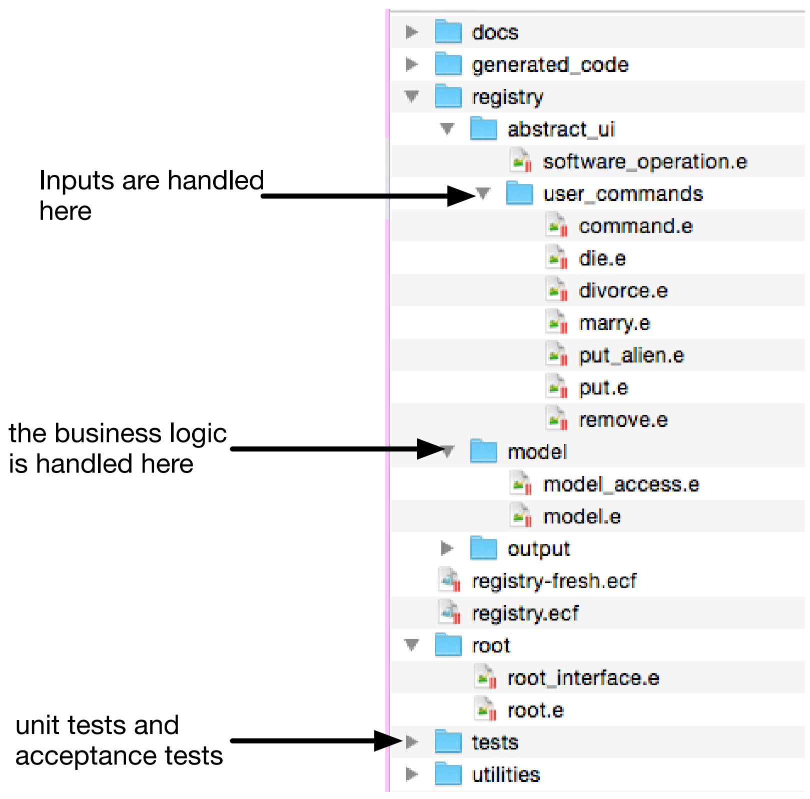


Figure 3: Directories and files generated by the ETF command

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.

[4]This image is from an older version of ETF. In the current version, all ETF files have a prefix "etf_".

3.2 Compile the project and run the acceptance tests

- Place the two acceptance tests `at1.txt` and `at2.txt` in the directory: `tests/acceptance/instructor`.
- Compile[5] the project: `estudio registry.ecf&`.
- You can now see how the project reacts to the acceptance tests by running the executable from the command line:
  `EIFGENs/EIFGENs/registry/W_code/registry -b tests/acceptance/instructor/at1.txt`

```
>registry -b at1.txt
  init
->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)
```

At this point we see only a default output. To obtain the correct output for this acceptance test, we must design and test the correctness of the business logic, which is the main activity for this lab.

3.3 Develop the business logic and test it

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`. 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.

- 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.

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

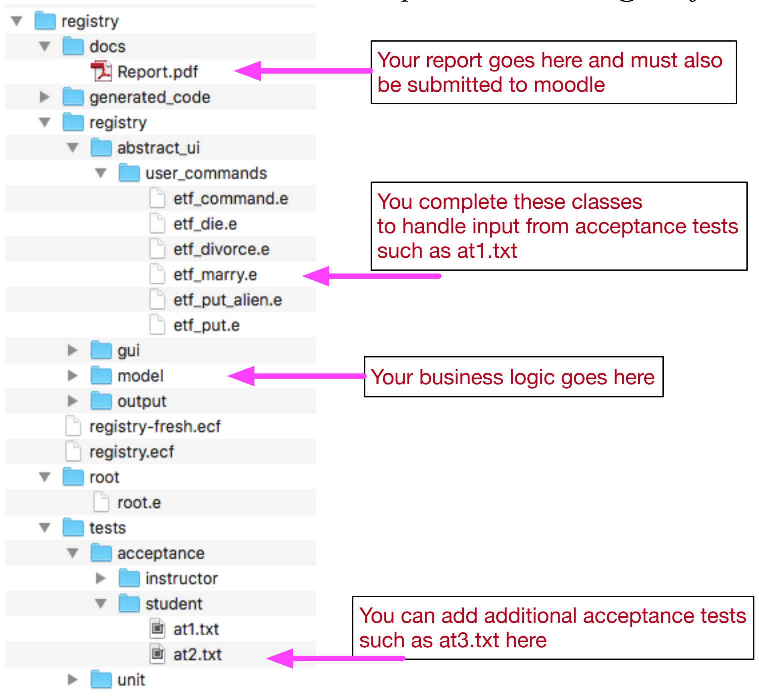
[5] Use the version of `estudio` chosen for the course. When this document was composed, it is `estudio17.05`

```
class ROOT inherit ES_SUITE ...

    unit_test: BOOLEAN = false

    make
      do
         initialize_attributes
         if unit_test then
            -- add your unit test files here...
            show_browser
            run_espec
         else
             -- run etf acceptance tests
            make_from_root
         end
      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. You must submit the top-level folder **registry** with the following structure.



```
▼ registry
  ▼ docs
      Report.pdf
  ▶ generated_code
  ▼ registry
    ▼ abstract_ui
      ▼ user_commands
          etf_command.e
          etf_die.e
          etf_divorce.e
          etf_marry.e
          etf_put_alien.e
          etf_put.e
    ▶ gui
    ▶ model
    ▶ output
    registry-fresh.ecf
    registry.ecf
  ▼ root
      root.e
  ▼ tests
    ▼ acceptance
      ▶ instructor
      ▼ student
          at1.txt
          at2.txt
    ▶ unit
```

Your report goes here and must also be submitted to moodle
You complete these classes to handle input from acceptance tests such as at1.txt
Your business logic goes here
You can add additional acceptance tests such as at3.txt here

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:

```
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

```
ec -c_compile -finalize -config registry.ecf
```

Before your final submit, set the *unit_test* to false (i.e. etf-mode).

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

Your **Report.pdf** will be short and will involve using the IDE to generate a **BON diagram** of your business logic and a **contract view** of the class that supports the business logic for marriage. More details of this report will be provided at the course wiki. If you do not submit a suitable report, then you will lose marks for this Lab.

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 spouse of thecurrent person is the current person themselves. There are many such consistency specifications in this application.

You have already used ARRAY and LIST. There is also the class 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 $person : ID \nrightarrow PERSON$ is a 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
```

We are not imposing an architecture, only suggesting possible design decisions that you might make in coming up with a clean modular architecture. The names of the classes and their relationships to each other is entirely up to you.

Class *REGISTRY* is also where you might 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.

On the other hand, perhaps the basic operations might arguably be supported elsewhere in the business logic (in cluster *model*), rather than making REGISTRY into a superman class.

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.

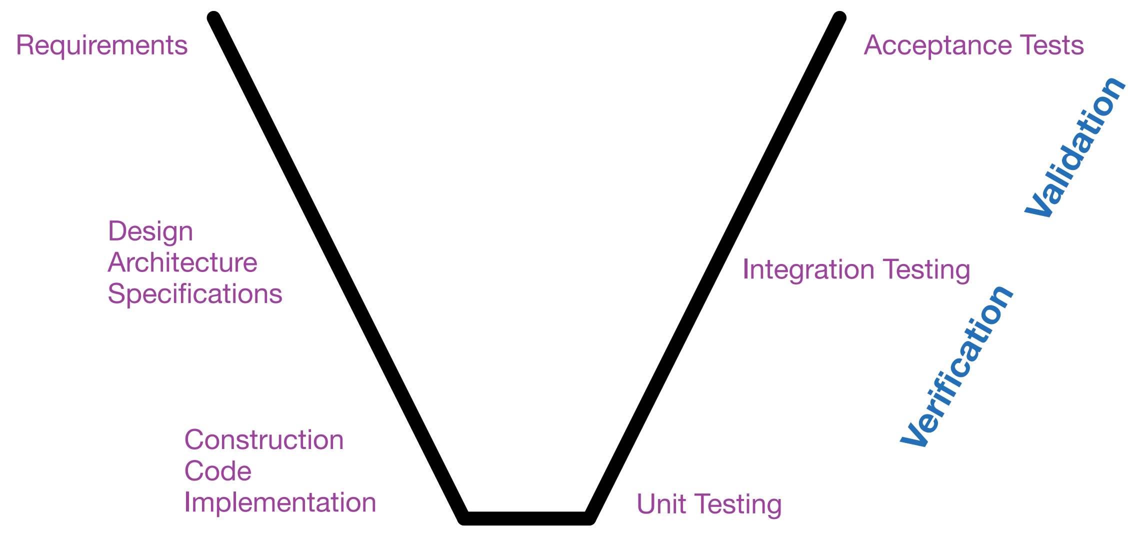
One of the design decisions you must make is how to handle error conditions in your implementation.

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 governs.

6 Appendix

A Validation and Verification in Software Engineering



The terms **Verification** and **Validation** are commonly used in software engineering to mean two different types of analysis. The usual definitions are:

Validation: Are we building the right system?

Verification: Are we building the system right?

Validation is concerned with checking that the system will meet the customer's actual needs (the requirements), while **Verification** is concerned with whether the system is well-engineered, safe, and error-free.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is fit for use and useful to the customer.

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 those who are not citizens (visitors, landed residents, etc.). 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:[6]

```
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

[6]Obviously, 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 the context of a Lab, 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:

```
  init
->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]
```

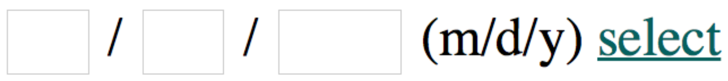
The above format shows an input operation performed by a user (for example by submitting a form from a webapp in which a new citizen "Joe" is added to the database) and then the resulting state of the database. A sequence of such operations and their effect forms the use case. The message "ok" indicates 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.[7]

C.1.1 Eiffel Testing Farmework – ETF

The ETF framework, described in Section 3, 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 to the requirements of the customer.

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].



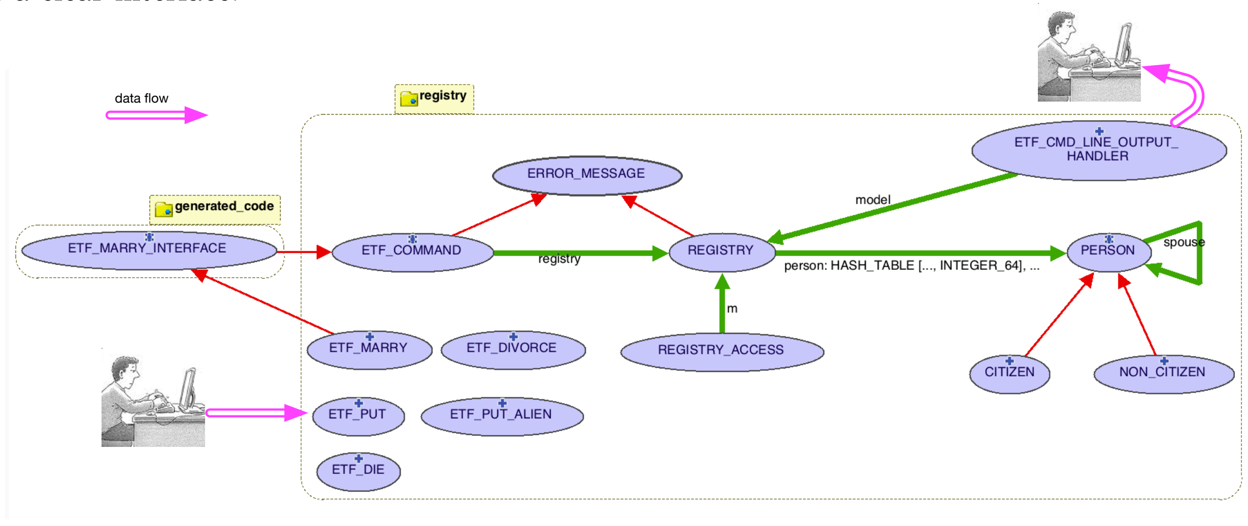
Even so, we may not be guaranteed that the dates are valid. For example, the date [31, 2, 1990] might satisfy the form checks in a webapp, but the date is still not legal. ETF provides the software developer with an *input grammar* that can be used to distinguish what checks are performed at the user 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.

D Design

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

[7]Unit 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.

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.



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.

D.1 Design Patterns

There are a number of design patterns in use in the ETF based architecture. For example, the Singleton Design Pattern is used to access class REGISTRY as a central source for the data of the business logic. Thus all command input classes access the same registry data via the once-only REGISTRY_ACCESS.

The MVC (model-view-controller) design pattern separates the model (the business logic) from the view (the user). In our version, the business logic (the model classes) have no dependency on the user inputs or output at all. Thus, the business logic may be used in a mobile or webapp as is, with only the user interface changing.

E Additional Exercises

Some (but not all) of the items may be required in your *Report.pdf*.

- Use the BON diagram tool of the IDE to generate the BON static class diagram view of the design.
- Complete contracts for and implement `marry` and `divorce`. Write unit tests to ensure that the implementations satisfy the specifications (contracts).
- Write contracts and unit tests for a feature `descendants`: LIST[CITIZEN] that calculates all the descendants of the current citizen.

- Use the debugger to see why tests fail and to draw the BON dynamic view. In particular check out the difference between a feature such as *spouse* in a BON class diagram and the dynamic field spouse in a UML sequence diagram
- Use the IDE document generator to generate the contract view and the flat (contract) view of classes such as PERSON or CITIZEN.
- What are your thoughts on your use of modularity, abstraction and information hiding in the design of the business logic.