EECS3311-W2016 eHealth Project Report

Submitted electronically by:

Team members	**Name**	**Prism Login**	**Signature**
Member 1:	Amanpreet Singh Walia	apsw02	A.W
Member 2:	Shehwar Elahi	cse13133	S.E
*Submitted under Prism account:		apsw02	

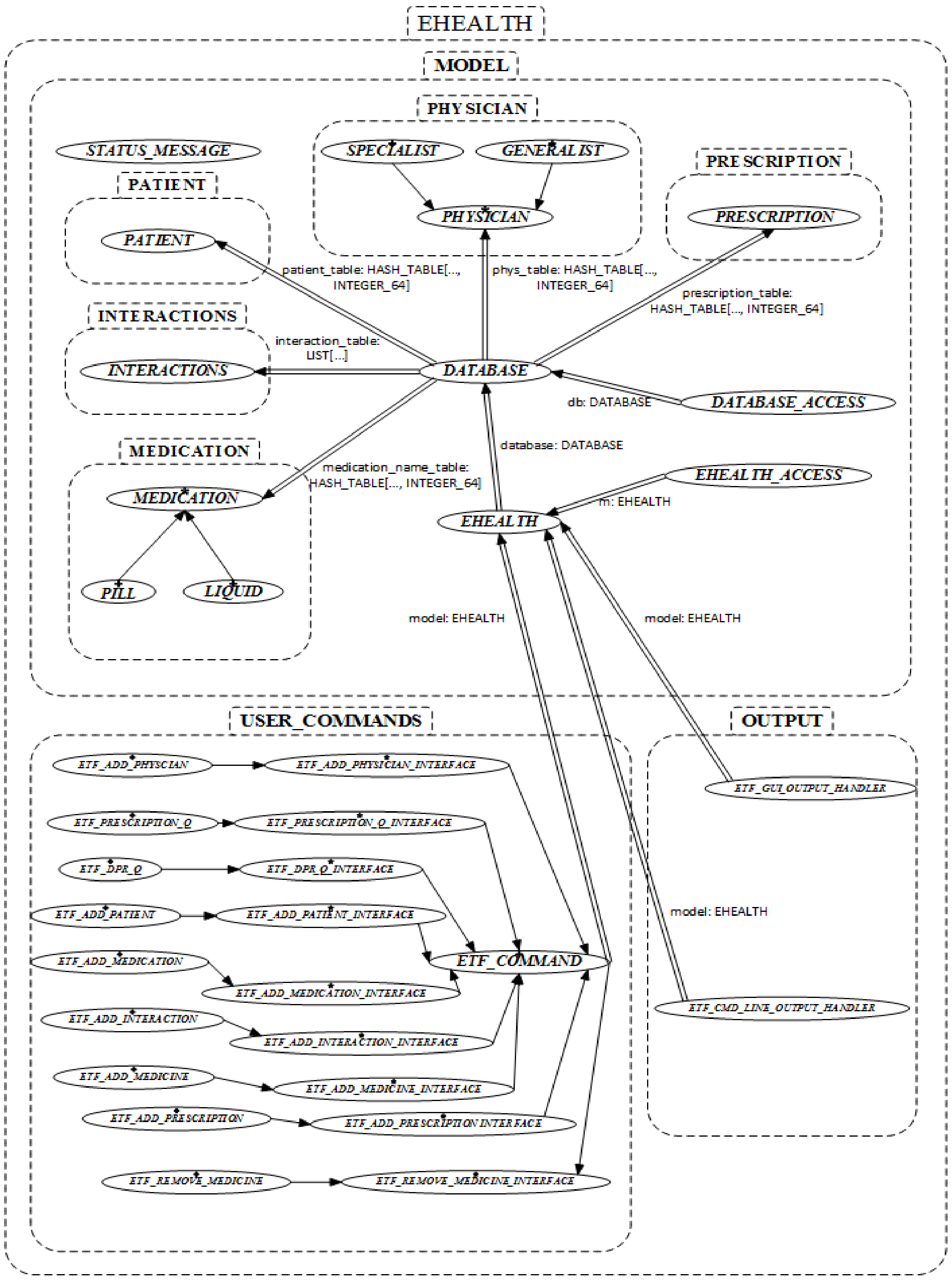
Contents

1. Requirements for Project eHealth

The subject is to develop an *eHealth prescriptions management prototype*, which will keep track of prescriptions for patients so that dangerous interactions between medications can be controlled and checked. There exist pairs of medications that when taken together have undesirable interactions; thus a safety critical application must be created which guarantees that no patient takes prescriptions with dangerous interactions. A new medicine can be added to a prescription by a generalist provided that it does not cause a dangerous interaction with other medicines taken by the patient. An interaction between the medications of a patient is allowed, provided that one of the medications in the interaction is prescribed by a specialist. You have to take into account new physicians, new medications, new prescriptions, new patients, new interactions and removing of medicines in the prescription. A console based application for user input suffices.

See *ehealth.definitions.txt* in the appendix for the grammar of the user interface. The acceptance tests *at1.expected.txt* describe some of the input-output behavior at the console for this project.

2. BON class diagram overview (architecture of the design)



The eHealth prescription management system resides in the MODEL cluster. Another cluster USER_COMMANDS is responsible for handling possible user inputs, providing status messages including relevant error messages for any inappropriate or failure inputs passed by the client. The OUTPUT cluster is responsible for handling the output for our system for both Command Line and GUI Interface.

The MODEL cluster of the eHealth system, includes PATIENT, MEDICATION, PHYSICIAN, INTERACTION, PRESCRIPTION, DATABASE, DATABASE_ACCESS, EHEALTH, EHEALTH_ACCESS and STATUS_MESSAGE classes. MEDICATION and PHYSICIAN are deferred classes. MEDICATION has been inherited by two effective classes PILL and LIQUID. PHYSICIAN has been inherited by SPECIALIST and GENERALIST class. The DATABASE class contains the data structures to store physicians, patients, prescriptions, medications and interactions. All the main features are implemented by this class and can only be accessed by EHEALTH. Only EHEALTH class can create an object of DATABASE. EHEALTH is using the database object to access all the data stored and then implements all the main functions and queries.

This software exhibits conceptual integrity as it defines encapsulation. The data stored in the DATABASE is safe from outside interference and misuse. Every aspect of eHealth is provided a class to encapsulate the data and to avoid unauthorized illegal access. The eHealth prescription management system just needs to interact with the universal database. The aspects of the database cannot be modified through other channels; thus providing a very ***robust*** API for data management and storage. This makes this software very ***reliable.*** This also makes this design *simple* as the program structure is very clear. The data storage is separated from the operation and is only accessed by the DATABASE. The client does not need to care about the storage of the data. As the design is ***simple***, it is also easy to ***maintain*** it. Every class has features needed to access its particular attributes, and there are no unneeded features. The design also exhibits ***modularity*** as each object forms a separate entity whose internal workings are decoupled from other parts of the system.

The design is ***reusable*** as the object classes created in this program can be used in other programs. For example, the STATUS_MESSAGE class can be used by other programs or softwares with very little modification. The design is ***modifiable*** as changes inside a class do not affect any other part of the program. The design also exhibits ***extendibility*** because adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

3. Table of modules — responsibilities and information hiding

1	PATIENT	**Responsibility**: implements a routine to create a PATIENT object and implements queries to access patient's attributes.	**Alternative**: implement two-way sorted list to store patients
	Concrete	**Secret**: *make_patient* visible to NONE and *create* routine visible to DATABASE class only.	

2	INTERACTION	**Responsibility**: creates interaction between two medications.	**Alternative**: implement linked list to store interactions
	Concrete	**Secret**: *make* routine visible to NONE and *create* visible to DATABASE class only.	

3	MEDICATION	**Responsibility**: creates a MEDICATION object and includes queries to access particular medication's attributes.	**Alternative**: implement two way sorted list to store medications
	Abstract	**Secret**: *create* routine visible/accessible to DATABASE class only and *make* is visible to none.	

3.1	PILL	**Responsibility**: creates a MEDICATION of type PILL.	**Alternative**: none
	Concrete	**Secret**: *make_pill* visible to NONE and create visible to DATABASE class only.	

3.2	LIQUID	**Responsibility**: creates a LIQUID object and includes queries to access particular object's attributes.	**Alternative**: none
	Concrete	**Secret**: *make_liquid* visible to NONE and *create* visible to DATABASE class only.	

4	PHYSICIAN	**Responsibility**: implements a make routine to create a PHYSICIAN object and includes queries to access features of a physician.	**Alternative**: implement two-way sorted list
	Abstract	**Secret**: *make_physician* visible to NONE and *create* visible to DATABASE class only.	

4.1	GENERALIST	**Responsibility**: creates a PHYSICIAN of type GENERALIST	**Alternative**: none
	Concrete	**Secret**: *make_generalist* visible to NONE and *create* visible to DATABASE class only.	

4.2	SPECIALIST	**Responsibility**: creates a PHYSICIAN of type SPECIALIST	**Alternative**: none
	Concrete	**Secret**: *make_specialist* visible to NONE and create visible to DATABASE class only	

5	PRESCRIPTION	**Responsibility**: implements a make routine to create PRESCRIPTION object and includes queries to access features of a prescription.	**Alternative**: implement two way sorted list to store prescriptions.
	Concrete	**Secret**: *create* routine visible/accessible to DATABASE class only and *make* visible to none.	

6	DATABASE	**Responsibility**: stores all the patients, interactions, physicians, medications and prescriptions in their particular data structures and implements all the basic features to modify them.	**Alternative**: none
	Concrete	**Secret**: features visible to eHealth and create visible to DATABASE_ACCESS only.	

7	DATABASE_ACCESS	**Responsibility**: An expanded class which allows access to DATABASE.	**Alternative**: none
	Concrete	**Secret**: *create* visible to EHEALTH class only.	

8	EHEALTH	**Responsibility**: implements basic software commands and queries.	**Alternative**: none
	Concrete	**Secret**: *create* visible to EHEALTH_ACCESS only.	

9	EHEALTH_ACCESS	**Responsibility**: expanded class creating an EHEALTH object.	**Alternative**: none
	Concrete	**Secret**: none	

10	STATUS_MESSAGE	**Responsibility**: responsible for storing all the error messages printed to the client on providing invalid input.	**Alternative**: none
	Concrete	**Secret**: none	

4. Expanded description of design decisions

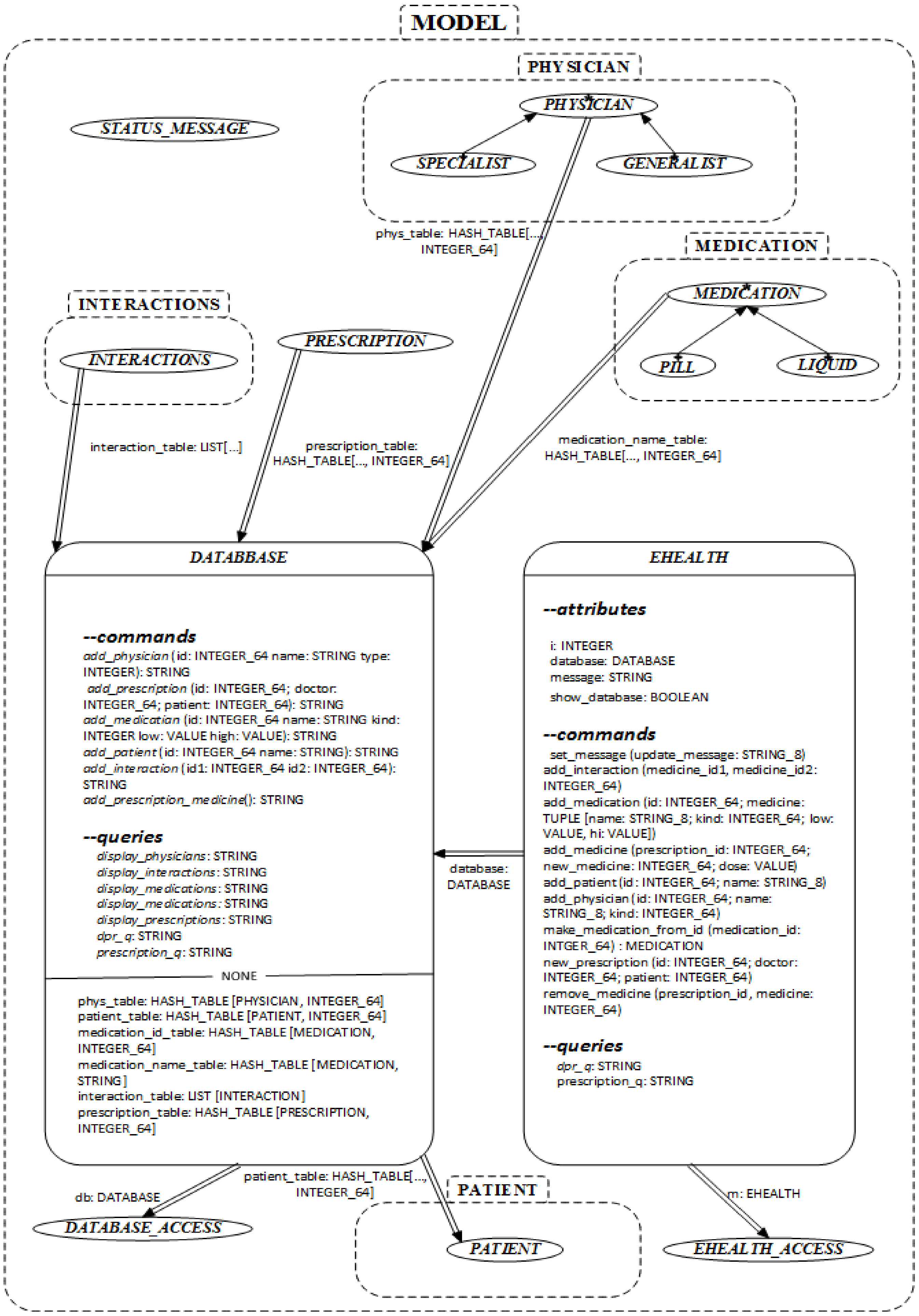
The eHealth prescription management software uses the ***Singleton Design Pattern***. PATIENT, PRESCRIPTION, MEDICATION, INTERACTION and PHYSICIAN classes have been created to make a patient, prescription, medication, interaction, and physician object respectfully. MEDICATION is a deferred class and is inherited by two other classes, 'PILL' and 'LIQUID'. PHYSICIAN is also a deferred class inherited by two concrete classes 'GENERALIST' and 'SPECIALIST'. These classes implement routines and queries to mutate and access the attributes of a particular object. Taking care of the ***information-hiding principle***, the create routine of all these classes is visible to only one class, which is the DATABASE class. DATABASE class is the universal database, which stores all the patients, physicians, medications, interactions and prescriptions using different data structures. This is the class which encapsulates all the data.

In the DATABASE class, HASH_TABLE has been used to store patients where patient id is the key and PATIENT is the value. The data structure used to store physicians is also HASH_TABLE, taking physician id as the key and PHYSICIAN object as value. Similarly, prescriptions and medications are also stored using HASH_TABLEs taking prescription id and medication id as keys, and PRESCRIPTION and MEDICATION as values. HASH_TABLE has been used because it gives easy access to a particular prescription, patient, medication or physician as each object has its own separate id. The usage of HASH_TABLE makes the system fast, as the average time complexity of inserting or removing an element from a HASH_TABLE is O (1). Interactions are stored in an ARRAY_LIST, which takes INTERACTION objects.

Some other data structures have been used to increase the efficiency of the program.

Singleton pattern restricts the instantiation of a class to one object. The software implements the Singleton design pattern by instantiating only one DATABASE object in the EHEALTH class. The EHEALTH class will use the DATABASE object to access all the main routines and queries implemented in the DATABASE. A single object of the DATABASE is allowed so that the design is safe as the data cannot be modified by any other class except the EHEALTH. DATABASE class includes the full implementation *'add_physician', 'add_medication', 'add_prescription', 'add_patient', 'add_interaction', 'remove_medicine' and 'add_medicine'* commands. It also includes '*dpr_q*' and '*prescription_q*' queries. All these commands and queries are also implemented in the EHEALTH which uses the corresponding commands and queries from the DATABASE class using the DATABASE class object.

This design has been chosen so that the stored data remains safe. If all the implementation and all the data is stored in the EHEALTH class without implementing the DATABASE, then there will be no ***information hiding***. There will be illegal access to the data which can be dangerous for the software.



```
DATABBASE

--commands
add_physician (id: INTEGER_64 name: STRING type:
INTEGER): STRING
 add_prescription (id: INTEGER_64; doctor:
INTEGER_64; patient: INTEGER_64): STRING
add_medication (id: INTEGER_64 name: STRING kind:
INTEGER low: VALUE high: VALUE): STRING
add_patient (id: INTEGER_64 name: STRING): STRING
add_interaction (id1: INTEGER_64 id2: INTEGER_64):
STRING
add_prescription_medicine(): STRING

--queries
  display_physicians: STRING
  display_interactions: STRING
  display_medications: STRING
  display_medications: STRING
  display_prescriptions: STRING
  dpr_q: STRING
  prescription_q: STRING
------------------ NONE ------------------
phys_table: HASH_TABLE [PHYSICIAN, INTEGER_64]
patient_table: HASH_TABLE [PATIENT, INTEGER_64]
medication_id_table: HASH_TABLE [MEDICATION,
INTEGER_64]
medication_name_table: HASH_TABLE [MEDICATION,
STRING]
interaction_table: LIST [INTERACTION]
prescription_table: HASH_TABLE [PRESCRIPTION,
INTEGER_64]
```

```
EHEALTH

--attributes
 i: INTEGER
 database: DATABASE
 message: STRING
 show_database: BOOLEAN

--commands
 set_message (update_message: STRING_8)
add_interaction (medicine_id1, medicine_id2:
INTEGER_64)
add_medication (id: INTEGER_64; medicine:
TUPLE [name: STRING_8; kind: INTEGER_64; low:
VALUE, hi: VALUE])
add_medicine (prescription_id: INTEGER_64;
new_medicine: INTEGER_64; dose: VALUE)
add_patient (id: INTEGER_64; name: STRING_8)
add_physician (id: INTEGER_64; name:
STRING_8; kind: INTEGER_64)
make_medication_from_id (medication_id:
INTGER_64) : MEDICATION
new_prescription (id: INTEGER_64; doctor:
INTEGER_64; patient: INTEGER_64)
remove_medicine (prescription_id, medicine:
INTEGER_64)

--queries
  dpr_q: STRING
  prescription_q: STRING
```

5. Significant Contracts (Correctness)

```
DATABASE

--commands
add_physician (id: INTEGER_64; name: STRING_8; kind: INTEGER_64)
        require
                positive_physician_id: (id > 0)
                physician_id_not_in_use: not (physician_id_in_use (id))
                valid_physician_name: is_valid_name (name)
        ensure
                phys_table.has(id)

add_patient (id : INTEGER_64; name: STRING_8)
        require
                positive_patient_id: (id > 0)
                patient_id_not_in_use: not (patient_id_in_use (id))
                valid_patient_name: is_valid_name (name)
        ensure
                patient_table.has(id)

add_medication (id: INTEGER_64; medicine: TUPLE [name: STRING_8; kind: INTEGER_64; low: VALUE; hi: VALUE])
        require
                positive_id: (id > 0)
                medication_does_not_already_exists: not (medication_id_in_use (id))
                valid_medication_name: is_valid_name (medicine.name)
                medication_name_not_in_use: not (medication_name_in_use (medicine.name))
                medicine_dose_valid: (medicine.low > zero and medicine.low <= medicine.hi)
        ensure
                medication_table.has(id)

new_prescription (id: INTEGER_64; doctor: INTEGER_64; patient: INTEGER_64)
        require
                positive_prescription_id: (id > 0)
                prescription_id_not_in_use: not (prescription_id_in_use (id))
                positive_physician_id: (doctor > 0)
                physician_id_exists: physician_id_in_use (doctor)
                positive_patient_id: (patient > 0)
                patient_id_exists: patient_id_in_use (patient)
                prescription_does_not_already_exist: not (prescription_for_patient_and_doctor_already_exists (doctor, patient))
        ensure
                prescription_table.has(id)

add_interaction (medicine_id1, medicine_id2: INTEGER_64)
        require
                positive_medicine_ids: (medicine_id1 > 0 and medicine_id2 > 0)
                medication_ids_different: not (medicine_id1 = medicine_id2)
                medication_ids_exist: (medication_id_in_use (medicine_id1) and medication_id_in_use (medicine_id2))
                unique_new_interaction: not (interaction_exists (medicine_id1, medicine_id2))
                non_conflicting_medicine: not (conflicting_medicine_exists (medicine_id1, medicine_id2))
        ensure
                interaction_table.has(INTERACTION)

make_medication_from_id (medication_id: INTEGER_64): MEDICATION
        require
                positive_medication_id: (medication_id > 0)

add_medicine (prescription_id: INTEGER_64; new_medicine: INTEGER_64; dose: VALUE)
        require
                positive_prescription_id: (prescription_id > 0)
                prescription_id_exists: prescription_id_in_use (prescription_id)
                positive_medicine_id: (new_medicine > 0)
                medication_id_exists: medication_id_in_use (new_medicine)
                medication_not_already_prescribed: not (medication_already_prescribed (prescription_id, new_medicine))

remove_medicine (prescription_id, medicine: INTEGER_64)
        require
                positive_prescription_id: (prescription_id > 0)
                prescription_exists: prescription_id_in_use (prescription_id)
                positive_medicine_id: (medicine > 0)
                medication_exists: medication_id_in_use (medicine)
                medication_exists_in_prescription: medication_in_prescription (prescription_id, medicine)

prescriptions_q (id_mn: INTEGER_64)
        require
                positive_medication_id: (id_mn > 0)
                medication_exists: medication_id_in_use (id_mn)
------------------------------------ Invariant ------------------------------------
```

The approach of ***Design by Contract (DbC)*** has been taken for designing the eHealth prescription management software. *Dbc* prescribes that formal, precise and verifiable interface specifications should be defined for software components, which extend the ordinary definition of abstract data type with preconditions, post-conditions and invariants.

Each of the routines responsible for validating patient, physician, prescription, medication, and interactions input make sure that the id is a positive integer. For example, the pre conditions for *add_patient* make sure that the id passed for the patient (which is to be added in the database) must be positive, and that the id is not already taken by any other patient already present in the database. Another precondition ensures that the name of the patient passed as a parameter must start with a letter. If a client fails to fulfill these preconditions, then an error message will be displayed and the patient will not be added to the database. If the preconditions are fulfilled, then the patient will be added to the database and an ensure clause is checking if the table storing all the patients has the particular patient added by the client.

A similar argument can be made for add_medication, new_prescription, add_interaction, add_medicine, remove_medicine, and prescriptions_q, however with the additional requirement that cross-argument validation is also required through contracting as mentioned previously. Implementing design by contracting allows our eHealth environment to be scalable to other systems that may extend our system's operations and supplement it with additional implementation details.

The *add_medicine* routine which adds a medication to the prescription requires the client to pass the prescription id and the medication id which must be positive. One of the precondition makes sure that the prescription in which medication has to be added, must be present in the list of prescriptions. Another precondition checks if the medication to be added already exists in the prescription. If that medication already exists in the prescription, then an error message is printed. If the medication has been added by a generalist and it has an interaction with another medication which is also added by the generalist, then the medication will not be added to the prescription. A precondition has been added to check that the dose entered for the medicine must be between lo and hi values. If all these preconditions are satisfied, then the medicine will be added to the prescription and an ensure clause will make sure that the medicine is now been added in the prescription.

The *remove_medicine* routine will have preconditions which will check that the prescription id passed as an argument, is a positive integer. The preconditions will also check if the prescription with this id exists, and that medication needed to be removed from that prescription is present in the prescription. The medication id must be a positive id and the medication must exist i.e. it must be present in the medication table. If all these preconditions are fulfilled by the client, then the medicine will be removed from the prescription. The post condition will ensure that the medicine is now removed from the prescription.

The *prescription_q* query requires the medication id which is passed as an argument to be a positive integer. Preconditions will also ensure that the medication id passed must be registered i.e. the medication associated with the id must be present in the HASH_TABLE storing all the medications. There are no post conditions as the query does not change the state of the database.
.

6. Summary of Testing Procedures

Test file	**Description**	**Passed**
at1.txt	Normal scenario where patients, prescriptions, medications, interactions and physicians are created, and medicines are added and removed from the prescription.	✓
at2.txt	Scenario in which client tries to add an interaction between two medications in the cases: i) where one medication is prescribed the generalist and other by the specialist ii) both medications prescribed by specialist and iii) both medications prescribed by the generalist.	✓
at3.txt	Testing the query which shows dangerous interactions.	✓
a4.txt	Checking for all failure status messages.	✓