Smart Pointer Project – Description

PR2

The exam will be using the `moped` environment. The following files have to be implemented for the exam:

illness.(h/cpp)

hospital.(h/cpp)

patient.(h/cpp)

hcp.(h/cpp)

In the following, the classes are described in detail. You can choose to implement the complete class in a header file or split the declaration and definition in header and cpp file. Since the classes **Patient**, **Health_Care_Provider**, and **Hospital** may share circular dependencies, we advice to split the implementation of at least these 3 classes into separate files. All member variables must be declared private.
The exam submission may only contain the **basic** implementation. It is recommended to save your basic implementation before you start working on additional parts.

The order in which the elements in a container are output to a stream must conform to the iterator order defined for the respective container.

Basic Implementation Deadline: 19.05.2023

1 Basic Implementation

1.1 Medical_Specialty

You can choose in which file you want to implement this class. **Medical_Specialty** is an enum class containing the following values.

```
enum class Medical_Specialty{ Cardiology,Dermatology,Endocrinology,Gynecology,
Neurology, Oncology,Pathology,Pediatrics,Pulmonology,Urology};
```

The following functions have to be implemented:

std::ostream& operator<<(std::ostream& o, Medical_Specialty m) Puts a string representation of `m` into ostream `o`. The string representation of `m` should reflect the name of the constant, e.g. **Medical_Specialty::Pathology** should be transformed to **Pathology**.

1.2 Illness

Member variables:

Medical_Specialty med Describes which specialty is required to cure this illness.

std::string name Specific name of the illness.

The following constructors and functions have to be implemented:

Illness(Medical_Specialty,const std::string&) Initializes member variables. name must not be empty. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

Medical_Specialty get_specialty() const Returns `med`.

std::string get_name() const Returns `name`.

std::ostream& operator<<(std::ostream& o, const Illness& ill) Puts a string representation of `ill` into ostream `o`. Format: [`med`, `name`]

bool operator== Compares two `Illness` objects. Returns true if name and specialty are equal. It is your choice if you want to implement it in the global namespace or as a member function.

bool operator< Compares two `Illness` objects. Returns true if specialty of the left operand is less than the specialty of the right operand (conforming to the order of definition). If the specialties are equal, the function returns true if the name of the left operand is less than the name of the right operand (in lexicographical order). It is your choice if you want to implement it in the global namespace or as a member function.

1.3 Patient

Member variables:

std::string name Name of patient.

int age Age of patient in years.

std::set<Illness> illnesses Contains illnesses the patient is suffering from.

unsigned wealth Wealth of patient.

The following constructors and functions have to be implemented:

Patient(std::string,int,const std::set<Illness>&,unsigned = 200) Initializes member variables. name must not be empty. age must not be less than 0. Patient must suffer from at least one `Illness` at the beginning. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

void increase_wealth(unsigned x) Increases `wealth` by `x`.

bool pay_procedure(unsigned x) If possible, decreases `wealth` by `x` and returns `true`. Otherwise returns `false`.

void catch_disease(const Illness& x) Adds `x` to the set of illnesses.

bool requires_treatment_in(Medical_Specialty x) const Returns `true` if patient suffers from at least 1 `Illness` with specialty `x`, otherwise `false`.

bool healthy() const Returns `true` if patient suffers from no illnesses, otherwise `false`.

sttring get_name() const Returns `name` of patient.

unsigned cure(Medical_Specialty x) Removes all `Illness` objects with the specialty `x`. Returns number of removed objects.

bool operator== Compares two `Patient` objects. Returns true if all member variables of the left operand are equal to all the member variables of the right operand. It is your choice if you want to implement it in the global namespace or as a member function.

std::ostream& operator<<(std::ostream& o, const Patient& p) Puts a string representation of `p` into ostream `o`.
Format: [`name`, `age` years, {$illness_0$, ..., $illness_n$}, `wealth` moneyz]

1.4 Health_Care_Provider

Member variables:

string name Name of health care provider (HCP).

set<Medical_Specialty> topics Set of topics in which HCP is proficient in

unsigned wealth Wealth of HCP.

The following constructors and functions have to be implemented:

Health_Care_Provider(std::string n, const set<Medical_Specialty>&,unsigned wealth=500
Initializes member variables. name must not be empty. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

std::string get_name() const Returns name

virtual ~Health_Care_Provider() Default Destructor.

void increase_wealth(unsigned x) Increases wealth by `x`.

virtual unsigned perform_procedure(unsigned,shared_ptr<Patient>, Medical_Specialty) = 0
Not implemented.

virtual string hcp_type() const = 0 Not Implemented

bool pay_license(unsigned x) If possible, `wealth` is decreased by `x` and `true` is returned. Otherwise `false`.

virtual void receive_license(Medical_Specialty x) Adds `x` to `topics`.

bool eligible_for(Medical_Specialty m) Return `true` if m is present in `topics`

std::ostream& operator<<(std::ostream& o, const Health_Care_Provider& h)
Puts a string representation of `h` into ostream `o`.
Format: [`name`, {$topics_0$, ..., $topics_n$}, `wealth` moneyz, `hcp_type`]

1.5 Hospital

Member variables:

string name Name of hospital.

map<string,shared_ptr<Health_Care_Provider>> hcps Key is the name of a HCP.

map<string,weak_ptr<Patient>> patients Key is the name of a Patient.

The following constructors and functions have to be implemented:

Hospital(std::string name) Initializes member variables. name must not be empty. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

bool sign_hcp(shared_ptr<Health_Care_Provider> m) If `m` is not present in `hcps`, `m` is inserted into `hcps` and `true` is returned, otherwise `false`.

bool admit_patient(shared_ptr<Patient> m) If `m` is not present in `patients`, or is present but expired, `m` is inserted into `patients` and `true` is returned, otherwise `false`.

bool dismiss_hcp(string n) Removes entry with key `n` from `hcps`. If at least one entry is removed, return `true`, otherwise `false`.

shared_ptr<Health_Care_Provider> get_hcp(string n) const If no HCP with name `n` is found, a runtime_error is thrown. Otherwise the shared_ptr is returned.

shared_ptr<Patient> get_patient(string n) const If no Patient with name `n` is found, or the weak_ptr is expired, a runtime_error is thrown. Otherwise a shared_ptr is returned.

bool dismiss_patient(string n) Removes entry with key `n` from `patients`. If at least one entry of a **not expired** patient is removed, return `true`, otherwise `false`.

std::ostream& operator<<(std::ostream& o, const Hospital& p) Puts a string representation of `p` into ostream `o`.
Format: [`name`, hcps {hcp_0, ..., hcp_n}, patients {$patient_0$,..., $patient_n$}]

1.6 Teaching_Health_Care_Provider : public Health_Care_Provider

Member variables:

unsigned fee Teaching fee.

The following constructors and functions have to be implemented:

Teaching_Health_Care_Provider(unsigned fee,std::string n, const set<Medical_Specialty>&,unsigned wealth=500
Initializes member variables. Same rules as Health_Care_Provider Constructor. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

unsigned perform_procedure(unsigned x,shared_ptr<Patient> p, Medical_Specialty m)
If `this` is eligible for `m` and `p` requires treatment in `m` and `p` can pay the price x, `p` is cured from `m`. Wealth is increased by `x`. Returns number of cured illnesses.

string hcp_type() const Returns "Teacher".

bool teach(Medical_Specialty m,shared_ptr<Health_Care_Provider> target)
If `target` points to `this`, knows `m` already, cannot afford to pay `fee` or `this` does not know `m`, `false` is returned. Otherwise wealth is increased by `fee`, `target` pays the `fee` and **receives** a license for the topic `m` and `true` is returned.

1.7 Smart_Health_Care_Provider : public Health_Care_Provider

Member variables:

unsigned fee Learning fee.

The following constructors and functions have to be implemented:

Smart_Health_Care_Provider(unsigned fee,std::string n, const set<Medical_Specialty>&,unsigned wealth=500)
Initializes member variables. Same rules as Health_Care_Provider Constructor. If at least one of the parameters is not matching the requirements, an exception of type `std::runtime_error` must be thrown.

unsigned perform_procedure(unsigned x,shared_ptr<Patient> p, Medical_Specialty m)
Returns 0.

string hcp_type() const Returns "Smart".

void receive_license(Medical_Specialty m) Adds the license to the topics. In addition, the wealth of `this` is increased by `fee`.

2 Additional Task

Extend the `Hospital` class by a log, i.e. an object, that tracks the total turnover for each of the hospital's `Health_Care_Providers`. (Turnover here means the sum of all the absolute changes of

wealth). To store the information, save weak_ptrs to the `HCP`-objects along with the respective turnover value. Extend and modify the classes as needed. Note that `HCP`-objects can be destroyed. The following function has to be implemented in `Hospital`:

map<weak_ptr<Health_Care_Provider>,unsigned> getTopTurnover() Returns the 3 `HCP`-objects of the **Hospital** (`this`-object) with the highest turnovers and their respective turnover values. If there are less than three `HCP`-objects that still exist, then the returned map will contain less entries accordingly. If there are multiple `HCP`-objects having the same turnover value so that the resulting map would contain more than three entries, the returned map must only contain three entries (arbitrarily ignoring some of the `HCP`-objects with identical turnover values).