**Lesson 0**

UML Diagrams

A close up of a logo

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Example:

Diagram

Description automatically generated

Example: NOV 2020

1. The client that manages the program manages

- the container (list)

+ adding the main object

2. main object

Types of objects represent children

e.g., patient and types of patients

( inheritance)

If both children have a property, the parent will have this property

e.g., patNum – patient number

3. Container (list object)

Remember the direction for the relationship (Object List )

( aggregation)

- list

+ append

4. The list object should have read/write

Write to list: WriteList

Read to list: ReadList

5. Object belonging to the main class

These are going to be our subclasses

e.g., Each patient record also has an object indicating the infection

Every time you add an object belonging to an object, it needs a

(1) Client

list: PatientList

addPatient(): void

s

(4) WriteList

write(QList<Patient>): bool

(4) ReadList

read(): QList<Patient>

(2) Patient

patNum: QString

infection: infection

addInfection(infection: void)

(3) PatientList

list: Qlist <Patient>

append(Patient\*): void

(5) Infection

infectCode: QString

(2) HealthWorker

worksAt: QString

(2) Nonhealthworker

testSite: QString

(5) viral

antiviral: bool

(5) bacterial

antibiotic: QString

Example: NOV 2017

1. The client that manages the program manages

- the container (list)

+ adding the main object

2. main object

Remember that for objects that might represent a data type, include attributes that might be useful to know

e.g. – size: float

( inheritance)

3. Container (list object)

Remember the direction for the relationship (Object List )

( aggregation)

- list

+ append

(3) ImageList

list: Qlist <Image>

append(Image\*): void

(1) Client

list: ImageList

AddImage(Image): void

(2) Image

Size: float

Image()

(4) WriteList

write(QList<Image>): bool

(4) ReadList

read(): QList<Image>

Example: JUNE 2021

1. The client that manages the program manages

- the container (list)

+ adding the main object

2. main object

Remember that for objects that might represent a data type, include attributes that might be useful to know

e.g. – size: float

( inheritance)

6. Abstract/ Pure Virtual classes

Indicate these with *Italics*

(3) VaccineList

list: Qlist <Vaccine>

addVaccine(Vaccine): void

(6) VaccineFactory

CreateVaccine(

QString: type

QString: adminster,

QString: destroy/process //1st lvl of subclasses

QString: doses/strain //2nd lvl of subclasses

): Vaccine\*

(2) Vaccine

administer: QString

returnAdminister(administer: QString)

uses

(2) Rabies

doses: int

toString(): QString

(2) Inactivated

destroyed: QString

toString(): QString

(2) Cholera

strain: QString

toString(): QString

(2) Attenuated

process: string

toString(): QString

creates

**Lesson 0**

Design

|  |  |  |
| --- | --- | --- |
|  | **Factory Method** | **Abstract Method** |
| Purpose | To not create objects | Only used to create objects |
|  | One product | Family of related or dependent products |
| Exposure of objects | Exposes method to the client for creating an object | Exposes family of related objects which may consist of these factory methods |
| Construction | Hides pattern of construction of a single object | Hides the construction of a family of related objects |
| Relationship | Composition | Inheritance. Relies on derived class or subclass |
|  |  |  |

Diagram

Description automatically generated

Diagram

Description automatically generated

|  |  |  |
| --- | --- | --- |
|  | **Singleton** | **Mono-State** |
| Purpose | Enforces structure of singularity | Enforces behavior of singularity |
| Transparency | Not transparent – know it is Singleton | Transparent – do not know it is mono-state |
| Inheritance | Can be created through inheritance | Cannot be created through inheritance |
|  | Cannot be inherited (private constructor) | Can inherit from mono-state class |
|  | Need to add static method & variable | Derived class is mono-state |
|  |  |  |

**Lesson 0**

Processes

Example: Q2.1 JUNE 2021

An application named vaccineData.exe can be used to gather vaccine data from a user.

Write the code that runs this application as a separate process, where its output is managed by the function in the main GUI/client named create().

*NB: This question requires this to be run as a process NOT a thread*

QProcess \* process = new QProcess(this);

/\*\* OLD version: Connect process to signal it sends out

1. Process is run. Sends out signal
2. Signal sent to std output
3. Received by slot

\*\*/

Connect (

process, // 1. process above

SIGNAL(readyReadStdOutput), // 2. signal sent out

this, // GUI class

SLOT(create()) // 3. slot called create

);

// Start the process

Process->start(‘vaccineData.exe’);

Example: Q2.1 JUNE 2021

Reimagined question with a class as a thread:

Class \* class = new Class; // subclass from QObject

QThread \*t new QThread;

/\*\* OLD version: Connect process to signal it sends out

1. Connect started signal (t) to slot in class
2. move class object to thread Received by slot
3. start thread

\*\*/

// Connect started signal (t) to slot in class

connect(thread, SIGNAL(started()), search, SLOT(find()));

connect(search, SIGNAL(found(Book)), client, SLOT(handleFound(Book)));

connect(search, SIGNAL(finished()), client, SLOT(handleFound(Book)));

// move class object to thread

search->moveToThread(thread);

// start thread

thread->start();

Example: Q2.2 JUNE 2021

Write the code for the create() function. It retrieves the output from the application process in 2.1 and calls the following factory method code to create an instance of a vaccine.

Notes:

- Output from the application read from the standard output is in the form

vaccineType\*argument1\*argument2\*argument3.

The above come from our VaccineFactory we did in the UML diagram

(6) VaccineFactory

CreateVaccine(

QString: type

QString: adminster, // argument1

QString: destroy/process // argument2

QString: doses/strain // argument3

): Vaccine\*

/\*\*

1. Get data
2. Split into parts
3. Pass those parts to vaccineFactory

\*\*/

**Void MainWindow:: create()**

**{**

// output is received as bytes. Need to convert this later to string

QByteArray bytes = Process->readAllStdOutput();

// separate output with \* delimiter. Convert to string list

QStringList items = QString(bytes).split(“\*”)

// call factory method

vaccineFunction vf;

// call factory method

Vaccine \* v = f.createVaccine(items[0], items[1], items[2], items[3]);

**}**

Example: Q2.3 JUNE 2021

Write the code for the class declaration of the vaccine list class as well as the implementation code of the function that adds a vaccine instance to the list. Remember to take note of the process given at the start of this question (specifically point d)

/\*\*

Class

\*\*/

**class VaccineList:** public QObject // A signal is sent. Inherit from QObject

**{**

Q\_OBJECT// QObject macro

**public:**

void addVaccine(Vaccine \*); // function to add vaccine

**signals:**

void vaccineAdded(QString); //

**private:**

Qlist<Vac \*> list; // container

**}**

/\*\*

1. Add to list
2. Create message
3. Emit message

\*\*/

**Void VaccineList::addVaccine:** (Vaccine \*v)

**{**

// remember to use the same name that you use for your list object

vaccineList->append(v);

// create message you will send. Signal should be passed automatically

QString = v->toString();

// emit signal

emit vacAdded(msg);

**}**

Example: Q2.4 JUNE 2021

In point e in the process above, the text containing the string representation of a vaccine instance is added to a QListWidget on the GUI. The following code has been provided (where ui->listWidget is the QListWidget variable and string is the string representation of the vaccine).

// incorrect. We are inserting an item, not a whole QListWidget

QListWidget \*newItem = new QListWidget;

newItem->setText(string);

ui->listWidget->setItem(1, newItem);

// correct

QListWidgetItem \*newItem = new QListWidgetItem;

newItem->setText(string);

ui->listWidget->InsertItem(1, newItem);

Example: Q2.5 JUNE 2021

Could a QStringListModel have been used instead of a QListWidget? If not, explain in detail why not; if it can, explain in detail how it could be achieved. Note that there are no marks for indicating yes or no.

Yes.

View class such as QListView could be used instead.

This could be attached to the model.

Q3 JUNE 2021

The vaccine list should be saved as an XML file when the application closes.

Assume a MyVaccineList class with the following interface.

int size() const // returns the number of items in the list  
Vaccine\* at(int) // returns the Vaccine instance at the specified position in the list

The XML file should have the following structure:

**<VaccineList>**

**<vaccine class="Cholera">**

**<administered>Unknown administration method</administered>**

**<process>Unknown process</process>  
<doses>0</doses>**

**</vaccine>**

**<vaccine class="Cholera">**

**<administered>injection</administered>**

**<process>disabled</process>**

**<doses>2</doses>**

**</vaccine>**

**</VaccineList>**

Notes:

- attribute tag on vaccine (class)

- text inside of administered

- probably need to use a for/while loop to access members within the list

Example: Q3.1 JUNE 2021

Use DOM to write the data from the vaccine list to the XML structure given above. Consider the following declarations and partial code (where the vaccine list is passed as a pointer to the write() function).

Remember to use reflective programming techniques so that the code is as generic as possible.

*reflective programming - No using getters/setters!!*

Remember:

int size() const // returns the number of items in the list  
Vaccine\* at(int) // returns the Vaccine instance at the specified position in the list

**QDomDocument xmlDoc;**

**QDomElement root;**

**root = xmlDoc.createElement("VaccineList"); // set up root element**

**xmlDoc.appendChild(root);**

**void XmlWriter::write(MyVaccineList \*mvl)**

**{**

**// loop through all items in list**

**for (int item=0; item<mvl->size(; item++)**

**{**

**// 1. get the meta-object**

*mvl = my vaccine list. at() helps us access the first item*

**const QMetaObject\* meta =** mvl->at(item)->metaObject();

QString classname = QString(meta->className());

**// 2. set up the vaccine tag**

*NB: Need to use the correct tag name !!! (from the xml)*

QDomElement vaccine = doc.createElement(“vaccine”);

vaccineElement.setAttribute(“class”, classname);

root.appendChild(vaccineElement)

**// loop through all the instance’s properties**

for (int count=1; count<meta>propertyCount(); count++)

{

**// 3. access instance’s properties**

Const QmetaProperty prop = meta->property(count);

Qstring propertyName = prop.name();

Qstring propertyValue = prop.name(mvl->at(item)).toString();

**// 4. create the appropriate XML tags**

QdomElement propertyElement = doc.createElement(propertyName);

QDomText val = doc.createTextnode(propertyValue);

vaccineElement.appendChild(propertyElement);

propertyElement.appendChild(val);

}

**}**

**// write data to file**

**QFile xmlfile("vaccineList.xml");**

**xmlfile.open(QIODevice::WriteOnly);**

**QTextStream toFile(&xmlfile);**

**toFile << doc.toString();**

**xmlfile.close();**

**}**

<VaccineList>

// 1. get the meta-object

// 2. Set up the vaccine tag

<vaccine class="Cholera">

// 3. access instance’s properties

//4. create the appropriate XML tags

<administered>Unknown administration method</administered>

// 3. access instance’s properties

//4. create the appropriate XML tags

<process>Unknown process</process>

// 3. access instance’s properties

//4. create the appropriate XML tags  
 <doses>0</doses>

// 2. Set up the vaccine tag

</vaccine>

</VaccineList>

3.3 SAX only parses XML files and cannot generate it.

Example: Q3.1 JUNE 2021

Reflective programming methods will be used to access data from **vaccine instances** in the list. Explain in detail what needs to be in place in the classes involved to allow this to be implementable. You should indicate clearly what needs to be done in which classes.

Notes on reflective programming:

- Replaces the need for accessors/mutators.

- A class can contain both:

Getters/setters (accessors/mutators)

Meta properties (Q\_PROPERTY). Fixed. Available for EVERY instance of class.

- With accessors/mutators, you need to know the interface to the class. In other words, you need to know the names of the accessors/mutators you want to use.

- With reflective programming, you can just loop through the properties. Provides a generic approach to accessing properties.

**ANSWER**

Vaccine instances need to be implementable

- Vaccine must inherit from QObject

- All classes in hierarchy need to implement O\_OBJECT macro

- Q\_PROPERTY must be included in classes for which data needs to be accessed. You will not be able to access properties via reflective programming without this property.

What NOT to say:

- list must be reflective (only vaccine instances must be reflective)

- access modifiers for the properties (don’t need to be changed)

- access specifiers must be public

(NOT true. Nothing within class should be changed)

- access specifiers must be private

(NOT true. Nothing within class should be changed)

Lesson 0

Input Masks

cheatsheet

Table

Description automatically generated

Examples:

|  |  |  |
| --- | --- | --- |
| Phone number (with area code) | 0611167195 | (###) ###-#### |
| 7-digit number. Optional area code | 1167195 | (999)000-0000! |
| LL-000 | CA-745 | 2 letters, followed by 3 numbers |
| >L<??????????????? | Maria Pierre | Uppercase first letter.  Lowercase following letter |
| #999 |  | Positive or negative number.  No more than 4 letters  No decimal places |
| >L????L?000L0 | GREENGR339M3  MAY R 452B7 | Combination of mandatory (L)  And optional (?) letters  And Mandatory numbers (0)  All letters must be uppercase |
|  |  |  |
|  |  |  |

**Lesson 0**

Regular expressions

Example: Q4.1 JUNE 2021

An optional property may be added to a vaccine instance indicating the authorisation code of the user who added it. The code should be made up of three to five uppercase alphabetic characters followed by a hyphen and then 3 digits. An optional final character may be included.

Input mask

>LLL??-000?

>AAAaa-999x //lecturers’ solution

Regular expression

(A-Z)(3-5) //uppercase chars

- //hyphen

(0-9)(3,3) //3 compulsory digits

? // optional last char

(A-Z)(3-5)-(0-9)(3,3)?//lecturers’ solution

Is it necessary to use both the input mask and regular expression to ensure a valid entry, explaining why you say so? Note that there are no marks for indicating only yes or no.

Yes.

Input mask has advantages like forcing capitalization of letters when user enters them.

By itself, an input mask can be quite weak.

Regular expressions are typically stronger and allow for very complex input checking.

**Lesson 0**

Memento Method