JanFeb 2023

Disclaimer: These resources are meant for learning purposes and comes with no guarantee of accuracy or correctness. Please do your own research and use it as a tool to double-check your own work, attempts and understanding.

Case:

There are many who make the argument that climate change is going to affect weather patterns, and, thus, the amount of rainfall that is going to fall. Some may get more, while others will get less. Tracking rainfall is, therefore, an important task. For the sake of this scenario, you may assume the following about designing an application that tracks rainfall.

- Rain data is stored using 3 pieces of data: the station that recorded the rainfall, the date for which the rainfall is recorded, and the amount of rain in mm.
- Obviously, there must be some way of recording all this data in a container.
- There should be a way of graphing the data. Initially, only a bar graph and a column graph are required. The data that is passed to the graphing application is an XML representation of the container holding all the rainfall data.
- Clearly, then, there needs to be some way of getting an XML representation of the rain record.
- The main client will be responsible for holding the record of all the rainfall data, for getting the XML representation of all the rainfall data, and for passing this to the user's choice of graph.
- Appropriate design patterns should be used as necessary.

Question 1.1:

Considering the scenario given above, draw a partial UML class diagram that captures the scenario. You should include the necessary classes, class attributes, and class relationships that are mentioned in the scenario. Class constructors and member access specifiers are not required. However, you should ensure that you include all the data members and member functions that show how data will be moved around the application. You should include the Client/GUI class. [You may use a software tool to create the UML class diagram. Use underlining to represent italics in hand-drawn UML class diagrams.] (20 marks)

Question 1.2:

It has been argued that the design pattern that would be used here is a behavioural pattern. Do you agree, indicating which pattern would be used and why it is or is not a behavioural pattern? (2 marks)

Question 1.3:

It was decided to provide the class that holds the rain data with some reflective functionality. That is, a getData() function should return a single data string that contains all the data held by the class via the class's meta-object, in a property named data. This function is used only by the meta-object and should not be available to users of the class. Write the class definition for this class showing how this would be set up. You are not required to include functionality or data members not mentioned above, or to code the implementation of the getData() function. (3 marks)

Question 2.1:

If the date is stored as a QDate (say, QDate date), give the code you would use to convert this data into the string format that is used in the XML text above? (2 marks)

Question 2.2:

Suppose that a class named RainXml would be used to generate the XML text for all the rain data.

```
class RainXml
{
public:
RainXml();
RainXml getInstance();
QString writeToXml(/*passing rain data*/);
private:
RainXml instance;
bool checkStationCode(QString stn) const;
QRegularExpression re;
};
```

This class is supposed to be implemented using a singleton design pattern.

Question 2.2.1:

The class definition code provided above does not correctly implement the classic singleton design pattern. Correct it so that it does. You are not expected to indicate the container that is used to pass rain data. (3 marks)

Question 2.2.2:

Would you agree with the decision to make this class use the singleton design pattern? Explain your reason clearly and persuasively; note that marks are only allocated for the reasoning. (2 marks)

Question 2.3:

The checkStationCode(QString stn) function from the class definition in 2.2 is used to ensure that the station code meets the correct format required, where the QString parameter is the station code that needs checking. The returned Boolean is true if the code meets requirements and false otherwise.

```
bool RainXml::checkStationCode(QString stn) const

{
// add code here
}
```

The station code should meet the following requirements. -Should begin with a capital letter. -This is followed by any 2 lowercase alphabetic characters. -This is followed by the same capital letter as the initial character of the code. -The numeric part of the code is made up of any 3 digits, where the first digit cannot be zero. -There should be no other characters before or after this code.

Question 2.3.1:

Write the QRegularExpression that would be used to check codes for correctness. QRegularExpression re(/*what would you put here*/); (7 marks)

Question 2.3.2:

Which anti-pattern would be involved if the code were not checked for meeting requirements? (1 marks)

Question 2.3.3 Provide the code for the checkStationCode(QString stn) function assuming that

QRegularExpression re, defined in 2.3.1, is a data member of the RainXml class. (2 marks)

Question 2.4:

The following code stub is used to generate the XML text given above. Using the required XML format from the start of question 2, the data property from the meta-object from 1.3, and the checkStationCode() function from 2.3, complete the code where indicated by comments.

Assume that each item of rain data can be obtained from the meta-object property set up in 1.3 – it returns the data in the form: StationCode:yyyy/mm/dd:mm. For example, if 10 mm of rain were received at station AagA100 on 1 January 2023, the data would be in the form AagA100:20230101:10.

You are not required to provide the code for the writeToXml() function parameter or the main loop that loops through all the rain records. You may assume that for each pass through the loop, you have access to a pointer r that points to a rain record.

```
QString RainXml::writeToXml(/*passing rain data*/)
{
QString xmlOutput;
QXmlStreamWriter writer(&xmlOutput);
// do initial setup of xml text
// loop through each rain pointer named r (do not code this)
{
// use the meta-object to get the required data
//if the station code passes the test
{
// set up the <rain> tag and its sub-tags as required
}
}
// end xml text
return xmlOutput;
}
```

(13 marks)

The plan is to search the whole record of all rainfall data for a particular station's data, and then present this data on the client screen. This search should be done in a thread. Consider the class implementation stub below (where stn is the station's data that is required).

```
StationThread::StationThread(/*all data*/, QString stn)
: record{/*all data*/}, station{stn}
{}
void StationThread::doSearch()
{
foreach(/*rain record in the data*/)
{
//get the station, date, and mm as strings
if (/*this station in the data*/ == station)
emit foundStation(/*date as string*/, /*mm as string*/);
}
}
```

Question 3.1:

Write the class definition for the StationThread class, remembering that it should be run as a thread. (8 marks)

Question 3.2:

Consider the code below that is run when data for a particular rain station is found by the code running in the thread, where QTableWidget *tableWidget and int row have already been declared and appropriately initialised.

```
void Client::handleFound(QString date, QString mm)
{
QTableWidgetItem *dateItem{new QTableWidgetItem(date)};
QTableWidgetItem *mmItem{new QTableWidgetItem(mm)};
tableWidget->setItem(row, 0, dateItem);
tableWidget->setItem(row++, 1, mmItem);
}
```

Assume the following code in Client.

```
QThread *t{new QThread};
StationThread *st{new StationThread(/*passing parameters*/)};
```

Write the code that would follow these declarations to get the thread running with the StationThread object, ensuring that the data is received from the running thread and passed on to the handleFound() function. (6 marks)

Question 3.3:

Would you agree that the QTableWidget used in 3.2 is the best approach that can be used for displaying a station's rainfall data on the client window? Give reasons for your answer. Note that marks are only allocated to your reasoning. (2 marks)

Question 3.4:

As a way of managing a backup/restore functionality, it has been decided to subclass the QTableWidget class to implement a classic memento design pattern. The start of the class definition is provided below.

```
class MyTableWidget: public QTableWidget
{
private:
friend class MyTableWidgetMemento;
MyTableWidget();
MyTableWidgetMemento* createMemento();
void setMemento(MyTableWidgetMemento *m);
};
```

Question 3.4.1:

What components usually make up the classic memento pattern, and which classes in this scenario would represent those components? (3 marks)

Question 3.4.2:

Explain why this implementation of the classic memento patten is correct or not. Marks are only awarded for the explanation. (2 marks)

Question 3.5:

It has been argued that cloud computing is the best way to go when it comes to managing all the rainfall data. Explain why this would be so in terms of cost and scale. (4 marks)

Attempt:

Question 1.1 (20 marks): UML Class Diagram

Given that we cannot generate images here, let's use a tree view to represent the UML class diagram.

```
1. RainData
    - station: QString
    - date: QDate
    - amount: double
    + getStation(): QString
    + getDate(): QDate
    + getAmount(): double
    + setData(QString, QDate, double): void
    + getData(): QString (meta-object only)

2. RainDataContainer
    - dataList: QList<RainData>
    + addData(RainData): void
    + getDataList(): QList<RainData>
    + toXml(): QString

3. Graph
    - type: QString ("Bar" or "Column")
    + setType(QString): void
    + getType(): QString
    + render(QString): void

4. Client
    - rainDataContainer: RainDataContainer
    - graph: Graph
    + addRainData(RainData): void
    + setGraphType(QString): void
    + displayGraph(): void
```

Relationships:

- `Client` has-a `RainDataContainer`
- `Client` has-a `Graph`
- `RainDataContainer` has-many `RainData`

Question 1.2 (2 marks): Design Pattern

I agree that a behavioral design pattern is appropriate here, specifically the Observer pattern. The Observer pattern would allow the `Client` class to observe changes in `RainDataContainer`. When new data is added to `RainDataContainer`, it notifies the `Client`, which then can update the graph.

Why it's a behavioral pattern:

- The Observer pattern is considered a behavioral pattern because it's used to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Question 1.3 (3 marks): Class Definition for RainData

Here is the class definition for the `RainData` class that holds the rain data and includes the reflective functionality.

```
#include <QString>
#include <QDate>

class RainData {
private:
    QString station;
    QDate date;
    double amount;

public:
    void setData(const QString& stn, const QDate& dt, double amt) {
        station = stn;
        date = dt;
        amount = amt;
    }

    QString getStation() const { return station; }
    QDate getDate() const { return date; }
    double getAmount() const { return amount; }

private:
    Q_PROPERTY(QString data READ getData)
    QString getData() const {
        // This function is used only by the meta-object
        return
QString("%1:%2:%3").arg(station).arg(date.toString("yyyy/MM/dd")).arg(amount);
    }
};
```

I've used the `Q_PROPERTY` macro to define a read-only property named `data`, which will invoke the `getData()` function to get its value. This function is intended to be used only by the meta-object and is not public.

Question 1.1 (20 marks): UML Class Diagram

```
1. RainData
    - station: QString
    - date: QDate
    - amount: double
    + getStation(): QString
    + getDate(): QDate
    + getAmount(): double
    + setData(QString, QDate, double): void
    + getData(): QString (meta-object only)

2. RainDataContainer
    - dataList: QList<RainData>
    + addData(RainData): void
    + getDataList(): QList<RainData>
    + toXml(): QString

3. Graph
    - type: QString ("Bar" or "Column")
    + setType(QString): void
    + getType(): QString
    + render(QString): void
```

```
4. Client
    - rainDataContainer: RainDataContainer
    - graph: Graph
    + addRainData(RainData): void
    + setGraphType(QString): void
    + displayGraph(): void
```

Relationships:

- `Client` has-a `RainDataContainer`
- `Client` has-a `Graph`
- `RainDataContainer` has-many `RainData`

Question 1.2 (2 marks): Design Pattern

I agree that a behavioral design pattern is appropriate here, specifically the Observer pattern. The Observer pattern would allow the `Client` class to observe changes in `RainDataContainer`. When new data is added to `RainDataContainer`, it notifies the `Client`, which then can update the graph.

Why it's a behavioral pattern:

- The Observer pattern is considered a behavioral pattern because it's used to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Question 1.3 (3 marks): Class Definition for RainData

Here is the class definition for the `RainData` class that holds the rain data and includes the reflective functionality.

```
#include <QString>
#include <QDate>

class RainData {
private:
    QString station;
    QDate date;
    double amount;

public:
    void setData(const QString& stn, const QDate& dt, double amt) {
        station = stn;
        date = dt;
        amount = amt;
    }

    QString getStation() const { return station; }
    QDate getDate() const { return date; }
    double getAmount() const { return amount; }
private:
    Q_PROPERTY(QString data READ getData)
    QString getData() const {
        // This function is used only by the meta-object
        return
QString("%1:%2:%3").arg(station).arg(date.toString("yyyy/MM/dd")).arg(amount);
    }
};
```

I've used the `Q_PROPERTY` macro to define a read-only property named `data`, which will invoke the `getData()` function to get its value. This function is intended to be used only by the meta-object and is not public.

Question 2.1 (2 marks): Converting QDate to String for XML

To convert a `QDate` object to a string format suitable for XML, you can use the `toString` method provided by the `QDate` class.

```
QDate date;  // Assume this is initialized
QString dateString = date.toString("yyyy-MM-dd");  // Converts to a string in the
format "YYYY-MM-DD"
```

Question 2.2.1 (3 marks): Singleton Pattern for RainXml

The class definition provided does not correctly implement the Singleton pattern. Here is how it can be corrected.

```
class RainXml {
private:
    static RainXml* instance;
    QRegularExpression re;

    RainXml() {}  // Private constructor

public:
    static RainXml* getInstance() {
        if (!instance) {
            instance = new RainXml();
        }
        return instance;
    }

    QString writeToXml(/*passing rain data*/);
    bool checkStationCode(QString stn) const;
};

// Initialize static member
RainXml* RainXml::instance = nullptr;
```

Question 2.2.2 (2 marks): Singleton Decision

I agree with the decision to make the `RainXml` class a Singleton. This is because generating XML for rain data is a centralized operation that doesn't require multiple instances of the `RainXml` class. A single instance can manage the XML representation efficiently, ensuring that there is a single point of control.

Question 2.3.1 (7 marks): QRegularExpression for Station Code

The QRegularExpression to check if the station code meets the given criteria would be as follows:

```
QRegularExpression re("^[A-Z][a-z]{2}[A-Z][1-9][0-9]{2}$");
```

Question 2.3.2 (1 mark): Anti-Pattern

If the station code were not checked for meeting requirements, it would be an example of the "Lack of Validation" anti-pattern, which can lead to various issues such as incorrect data processing and potential security vulnerabilities.

Question 2.3.3 (2 marks): checkStationCode Function

Here is how the `checkStationCode` function could be implemented:

```
bool RainXml::checkStationCode(QString stn) const {
    return re.match(stn).hasMatch();
}
```

Question 2.4 (13 marks): writeToXml Function

Here is a stub for the `writeToXml` function, incorporating the required functionality:

```
QString RainXml::writeToXml(/*passing rain data*/)
{
    QString xmlOutput;
    QXmlStreamWriter writer(&xmlOutput);
    writer.writeStartDocument();
    writer.writeStartElement("rainData");

    // Loop through each rain pointer named r (not coding this loop)
    {
        // Use the meta-object to get the required data
        QString data = r->getData();  // Assume r is a pointer to a RainData
object

        // If the station code passes the test
        if (checkStationCode(data.split(":")[0])) {
            writer.writeStartElement("rain");

            writer.writeTextElement("station", data.split(":")[0]);
            writer.writeTextElement("date", data.split(":")[1]);
            writer.writeTextElement("amount", data.split(":")[2]);

            writer.writeEndElement();  // End of "rain"
        }
    }

    writer.writeEndElement();  // End of "rainData"
    writer.writeEndDocument();

    return xmlOutput;
}
```

Question 3.1 (8 marks): StationThread Class Definition

To run the `StationThread` class as a thread in Qt, you can subclass `QRunnable` or `QThread`. Here, I'll use `QThread` for demonstration.

```
#include <QThread>
#include <QString>
#include <QList>

class StationThread : public QThread
{
    Q_OBJECT
private:
    QList<RainData> record;  // Assume RainData is the class that holds individual rain records
    QString station;

public:
    StationThread(const QList<RainData>& allData, const QString& stn)
        : record(allData), station(stn) {}

    void run() override {
        doSearch();
    }

    void doSearch();

signals:
    void foundStation(QString date, QString mm);
};
```

Question 3.2 (6 marks): Thread Execution and Signal Handling

The following code will set up the thread and connect its signals.

```
QThread* t = new QThread;
StationThread* st = new StationThread(/*passing parameters*/);

st->moveToThread(t);

connect(st, &StationThread::foundStation, &Client::handleFound);
connect(t, &QThread::started, st);
connect(t, &QThread::finished, t, &QThread::deleteLater);

t->start();
```

Question 3.3 (2 marks): QTableWidget Decision

Yes, I agree that using `QTableWidget` is a suitable approach for displaying the station's rainfall data. It's a high-level API that provides built-in functionality for item selection, sorting, and other common requirements, which saves development time. Also, it allows easy mapping of data to rows and columns, which is quite intuitive for the user.

Question 3.4.1 (3 marks): Classic Memento Pattern Components

The classic memento pattern usually consists of three main components:

- Originator: The object whose state needs to be saved and restored. (`MyTableWidget` in this case)
- Memento: The object that actually stores the state of the Originator. (`MyTableWidgetMemento`)
- Caretaker: The object that keeps track of multiple memento. (Could be a `BackupManager` class or the `Client` class in this scenario)

Question 3.4.2 (2 marks): Memento Pattern Implementation

The implementation appears to be on the right track for applying the Memento pattern. The `MyTableWidget` class (the Originator) has methods to create and set a memento (`MyTableWidgetMemento`). However, without seeing the implementation of these methods and the `MyTableWidgetMemento` class, it's hard to definitively say if the pattern is correctly implemented.

Question 3.5 (4 marks): Cloud Computing for Rainfall Data Management

Utilizing cloud computing for managing all the rainfall data would be advantageous in terms of both cost and scale:

- **Cost-Efficiency**: Cloud services often operate on a pay-as-you-go model, meaning you only pay for the storage and computational power you actually use. This eliminates the upfront cost of setting up and maintaining physical servers.

- **Scalability**: Cloud platforms can easily adapt to changing needs. If more storage or computational power is needed, it can be added instantly without the need to purchase and set up additional physical hardware.
