LO21 project report: UTComputer

Alexandre Ballet, Anton Ippolitov (GI02)

P16

Contents

Abstract

In this report, we will present our application developped for the L021 course. A revolutionary and innovative calculator which will simplify your everyday life.

Part 1: Architecture

1.1 QComputer

Our *QComputer* class is the interface of our application, our *MainWindow*. It sends the content of the *QLineEdit* widget to the *Controleur* instance each time the user presses the *Enter* key or clicks on an operator button. Every instruction is sent to the *parse()* method of the *Controleur* instance by the *QLineEdit* widget. The *QComputer* object refreshes the *QTableWidget* displaying our *Pile* instance each time it is modified. It also saves/restores the context when the application terminates/launches : the content of the *Pile* instance, the *Variable* and *Programme* objects, and the global settings (keyboard, error sound and number of *Litteral* objects displayed in the *QTableWidget*).

1.2 Controleur

Our *Controleur* class is a singleton and manages every interaction with the user. It parses and processes each user input : creates the corresponding *Litteral* objects, pushes them into the *Pile* instance, applies the operators, parses and evaluates the *Expression* objects. It also manages the *Memento* class : adds *Memento* states to the *mementoList* stored in the *Pile* instance.

We chose to centralise every user input processing into the *parse()* method in order to simplify the use of the *Controleur* class. If the input is not a *Programme* or an *Expression*, which both contain *spaces*, the *parse()* method manually splits the input into seperate words to be processed by the *process()* method. The *Litteral* objects are pushed into the *Pile* instance and the operators are applied.

1.3 Pile

As there can only be one *Pile* object, our *Pile* class is a singleton. It has a *QStack* attribute (called *stack*) of pointers to *Litteral* objects. The *Pile* instance has *pop()* and *push()* methods in order to manage the stack attribute. It has a *QString* attribute called *message*, used to display *QComputerExceptions* into a *QLineEdit* widget of the main window.

1.4 Litterals

The *Litteral* class is the mother class of *LitteralNumerique*, *Complexe*, *Variable*, *Programme*, *Expression* and *Atome*.
A *LitteralNumerique* object is either an *Entier*, a *Reel* or a *Rationnel*. It has 2 virtual methods *operator<()* and *operator>()* which are implemented in the children classes and used to compare each other.
A *Complexe* object is composed of 2 *LitteralNumerique* objects.
A *Rationnel* object is composed of 2 *Entier* objects.
A *Variable* object is composed of an id (*QString*) and a *Litteral* object (as it can store *Entier*, *Reel*, *Rationnel* or *Complexe* objects).
A *Programme* object is composed of an id (*QString*) and instructions (*QStringList*), as it can contain operators, that are not objects.

1.5 Operators

An operator is not an object. When an operator is parsed by the *Controleur* instance, it is recognized as an *operatorNum*, *operatorLog* or *operatorPile*. The method *applyOperatorNum()*, *applyOperatorLog()* or *applyOperatorPile()* is then called, having as parameters the *QString* id of the operator parsed, and its arity (found in the static *QMap<QString, int>* opsNum, opsLog or opsPile, which associate each operator to its arity).

C++ code sample of the *QMap* for logical operators :

```
        static const QMap<QString, int> opsLog{
            {"=", 2},
            {"!=", 2},
            {"<", 2},
            {">", 2},
            {">=", 2},
            {"<=", 2},
            {"AND", 2},
            {"OR", 2},
            {"NOT", 1}
        };
```

1.6 VariableMap and ProgrammeMap

Variable objects are stored in a *QMap<QString, Variable*>* attribute of a singleton *VariableMap* class. This means that the research and the deletion of a *Variable* object has a complexity in O(Log(n)). We can then find, edit and delete *Variable* objects efficiently from their id. *Programme* objects are stored in a *QMap<QString, Programme*>* attribute of a singleton *ProgrammeMap* class. Like *VariableMap*, the research and deletion of a *Programme* object has a complexity in O(Log(n)).

1.7 ComputerException

Every error occuring in *UTComputer* is managed by the *ComputerException* class. Each time a *ComputerException* object is created, the message of the *Pile* instance is updated and displayed in the *QComputer* interface.

Part 2: Architecture extensibility possibilities

Our architecture allows for easy extensibility. Here are some examples of extensibility possibilities that can be easily implemented within our architecture.

2.1 Adding operators

If we want to add a new operator, most of the steps are already automated. Qt Designer is quite practical for this task: it is very easy to add a new button for the operator. The button is then automatically connected to a slot handling the button's action using a *foreach* loop:

```
//connect all keyboard buttons
QList<QPushButton*> buttons = this->findChildren<QPushButton*>();
foreach (QPushButton* button, buttons) {
    connect(button, SIGNAL(released()), this, SLOT(editCommmande()));
}
```

The next step is choosing the operator's category (e.g. stack operator, numerical operator, etc.). The operator as well as its arity must then be added to the corresponding *QMap* in the *operator.h* file.
The final step is writing a function that implements the operator's functionality and putting it into *Controleur.h*. This step is operator-specific so it cannot be automated. All the other connections and logic are completely the same, be it 10 operators or 100. //parsing infox to postfix

2.2 Settings

It also easy to add new settings to the app (for instance for changing the color of the application or changing the decimal precision). Using Qt Designer, a checkbox or an appropriate input widget can be quickly added to the settings window. This widget is then connected to a slot handling user input. This slot basically contains an update of the *QSettings* object. As the settings' values can be booleans, integers or even strings this part of the process is simple yet cannot be automated. The *QSettings* is then used in the appropriate method or slot of the target object that handles the settings for this object (for instance *QComputer* would have a slot that would react when the 'color' setting is changed). All other aspects are automated, such as restoring and saving settings from and to the SQLite database: a *foreach* loop iterates through all table rows (or settings' keys) and restores them (or saves them) automatically.

2.3 Translations

All the text that the user sees is embedded in the Qt *tr* function.... meh

2.4 Resources

Sounds and icons are managed via .qrc files (Qt Resource files) that make it easy to add, remove or edit auxiliary files for the application. It is very quick to customize the app sounds or icons using this approach.