LO21 project report: UTComputer

Alexandre Ballet, Anton Ippolitov (GI02)

P16

Contents

Abstract

In this report, we will present our application developed 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 every time it is modified.
It also manages the saving/restoring of context via the *DbManager* class when the application terminates/launches : it saves the content of the *Pile* instance, all *Variable* and *Programme* objects as well as the global settings (keyboard on/off, error sound on/off 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 all user input : it 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 : it adds *Memento* states to the *mementoList* which is used for undo/redo operations.

We chose to centralise all 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. It has a *QString* attribute called *message*, used to display *QComputerExceptions* on a *QLineEdit* widget of the main window.
The *Pile* object can also create a *Memento* object holding the current state of the stack. It can also restore a state of the stack from a textitMemento object. All textitMemento objects are managed by the textitControleur class.

1.4 Litterals

The *Litteral* class is the base class of *LitteralNumerique*, *Complexe*, *Variable*, *Programme*, *Expression* and *Atome*. It also implements a factory methods which instantiates the correct child class object from user input (a QString).
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. It simplifies itself during construction. The factory method then can even instantiate an *Entier* object if the denominator equals to 1.
A *Variable* object is composed of an id (*QString*) and a pointer to a *Litteral* object (as it can store *Entier*, *Reel*, *Rationnel* or *Complexe* objects). On construction it is automatically added to the singleton *VariableMap* which manages all Variables.
A *Programme* object is composed of an id (*QString*) and instructions (*QStringList*), as it can contain operators, that are not objects. On construction it is automatically added to the singleton *ProgrammeMap* which manages all programs. When the EVAL operator is applied, a program is parsed and executed by the

textitControleur.
An *Expression* object is composed of a QString attribute which holds the expression itself. The expression is in infix notation. When the EVAL operator is used, a function parses the expression uses an infix to postfix transformation algorithm and returns a postfix expression which is then evaluated by the *Controleur*.

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 lookup and the deletion of a *Variable* object has a complexity of $\mathcal{O}(\log(n))$. We can then find, edit and delete *Variable* objects efficiently using their id. The user can manage Variables using STO and FORGET operators but also using a *VariableEditor* window. This window can be accessed from the menu bar. It allows for editing, adding and deleting variables. It instantiates the *VariableMap* singleton and performs these simple operations using dedicated *VariableMap* methods.
Programme objects are stored in a *QMap<QString, Programme*>* attribute of a singleton *ProgrammeMap* class. Like *VariableMap*, the lookup and deletion of a *Programme* object has a complexity of $\mathcal{O}(\log(n))$. The user can also manage programs from a special *ProgramEditor* window similar to the *VariableEditor* window. This window can launch smaller notepad-like windows where the user can modify specific programs. Different delimiters are accepted such as spaces, tabs and newlines.

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. A beep sound is also played on creation.

1.8 Options

The user can change app settings from a dedicated settings window, accessible from the menu bar. It is possible to hide or show the keyboard, to turn off or on the beep sound as well as to change the number

of *Litteral* objects displayed on the QTableWidget stack in the main window. All settings are stored in the QSettings class which provides an easy to use and platform independent way to manage user settings. The *QComputer* listens for signals sent from the Options window (a signal is sent when the user changes settings) and reacts accordingly (e.g. hides or shows the keyboard).

1.9 DbManager

We chose to store app context in an SQLite database. The *DbManager* singleton does all the dirty work so the *QComputer* class can simply call appropriate methods and easily save the context when the user closes the app or restore it when the app is launched. The SQLite database has 4 tables: pile, variables, programs and settings. The *DbManager* has methods for saving textual representations of all objects into appropriate tables as well as for restoring objects from the tables.

On the next page is a UML diagram showing all our classes. We chose to represent only inheritance relationships because representing other relationships would make the diagram incredibly cluttered, unnecessarily complicated and impossible to read. Moreover, visualisations generated by Doxygen complement this diagram very well as they only represent relationships between classes.

Part 2: Architecture extensibility possibilities

Our architecture allows for easy extensibility. Here are some examples of extensibility possibilities that can be quickly 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.

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 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. This means that it is possible to create themes for the app and let users customize it.

PS: type DISCO in the command line and turn on the sound for a unique user experience in the realm of calculator apps!