

Developing Qt with Qt: A UML Tool

Por HENG^{*}, Saovorak KHOY^{*}, Thomas FANNES[‡], Stef DESMET[‡]

^{*}Master student <E-Media>, GROUP T – Leuven Engineering College, Vesaliusstraat 13, 3000 Leuven

[‡]Unit <E-Media>, GROUP T – Leuven Engineering College, Vesaliusstraat 13, 3000 Leuven,
<thomas.fannes@groept.be>, <stef.desmet@groept.be>

Abstract

The work described in this paper targets the development of a UML (Unified Modeling Language) tool as a new plug-in for Qt Creator. Qt Creator is the IDE (Integrated Development Environment) of Qt (pronounced as cute) which is a cross-platform application and UI framework. With Qt, you can code applications once and deploy them across many platforms without rewriting the code. The Qt Creator is built on an easy extendable architecture, the “Plug-in Loader”, which means that all its functionalities are plug-ins (shared library). This architecture makes it possible to extend Qt Creator with a self developed plug-in: drawing UML class diagrams and generating skeleton code in C++ out of it. The reverse operation has also been realized, imports the existing code and then generates the corresponding class diagram.

Because of the wide variety in C++ to translate OO (Object Oriented) concepts in code, the plug-in has known limitations. This may be the task of a future master thesis.

Keywords:

QT, Qt Creator, Plug-in, UML (Class Diagram), C++

Introduction

Having a good model of your software is the best way to communicate with other developers working on the project and with your customers. A good model is not only extremely important for medium and big-size projects, but it is also very useful for small ones because it will provide an overview which will help programmer to code things right at the first time. Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering ^[14]. For more efficiency in creating or modeling UML we need a good tool. There are a lot of UML tools available and with the good

functionality, e.g. round trip engineering (code generation and reverse engineering). These functionalities are really helpful for developers to go from modeling to code and especially from code to modeling which can save developers' time. Moreover, the reverse engine can provide an overview and structure of a system which is important for the software reconstruction or modification phase.

In software development, the architecture is the core of the development and its objective is to make sure that the application is extendable, reliable, and maintainable. Qt Creator is a good example of a well designed architecture. Qt Creator, the cross-platform C++ IDE which is part of the Qt SDK, functions as a plug-in loader with all its functionality implemented as a shared library. Therefore, it is possible for us to develop our own plug-in for its extension.

As mentioned earlier, there are many UML tools available such as Umbrallo, BoUML, and Eclipse UML2 Tools ^[15] with full functionalities including the round trip engineering but none of them has been created for Qt Creator and specifically oriented to Qt framework like UML tool can define signal and slots mechanism in Qt framework. In addition, it is also nice to have one environment that we can model and code synchronously. Particularly, for personal reason we want to learn more about Qt and C++. Thus it is really convincing for us to develop a UML tool as the Qt Creator plug-in. According to these motivations and time constraint, our plug-in focuses on class diagram which is a part of UML. The class diagram is the visual modeling of classes in a system or application which shows their relations and eventually also the attributes, methods.

The purpose of this paper is to motivate the development decisions and explain how the implementation has been realized. Finally, we discuss the way we generate code from the model along with

the reverse engine which imports the existing C++ code and generates back the corresponding class diagram.

1. UML Class Diagram

UML or Unified Modeling Language is a visual language for modeling and communicating about system through the use of diagram and supporting text^[4] which is widely used in every software engineering processes focusing on the object oriented paradigm. UML standard diagram is divided into two different groups, structure diagram and behavioral diagram. The class diagram belongs to the structure diagram category which is popular among the programmers.

The purpose of the class diagram is to show the types being modeled within the system. In the most UML, these types include class, package, interface and abstract class which are shown in figure 1. Class is a kind of classifier whose features are attributes and operations and is represented as the rectangle containing three compartment lists stacked vertically. The top compartment is the class's name; the middle one is the list of its attribute and the bottom compartment is the list of operations. Package is a container which possibly contains another packages or classes. And its advantage is to enable the modeler to organize the model classifier by placing them in groups which is similar to a folder in the file system. Interface is a class that may have operations but may not have attributes or associations. And it defines a service or contract as a collection of public operations^[5]. It is symbolized as rectangle with only two subdivision lists stacked vertically; the top section is the interface's name with the keyword «interface» on top and the second is the list of operations. While a class can have an actual instance, an interface needs at least one class to implement it. And abstract class is a class that contains at least one abstract or pure virtual method. Similarly to interface, it can't have an actual instance. Its picture looks the same as class except there is keyword «abstract» on top of the class' name.

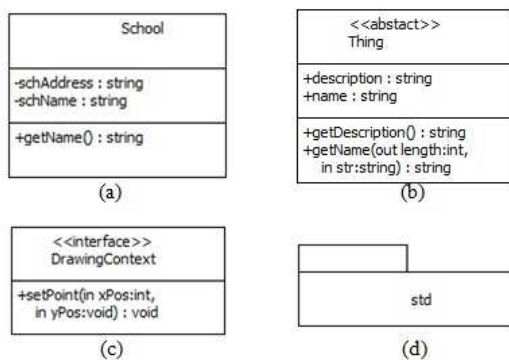


Figure 1: UML class diagram's elements; (a) class, (b) abstract class, (c) interface, (d) package

When a system is modeled, certain objects will be related to each other, and these relationships need to be modeled for clarity. A first type of relation is inheritance which means the new classes called derived-classes are created from existing classes called base-classes. The derived-classes have all the features of the base-classes and there is possibility to add new features to newly derived-classes. In UML class diagram, inheritance is represented by a peculiar triangular arrowhead (see figure 2a). Secondly, an association specifies a semantic relationship that can occur between typed instances. It is illustrated as an arrow (see figure 2b). Thirdly, aggregation is special type of association used to model a "whole-part" relationship between an aggregate, the whole and its parts^[4] which basically means that the lifecycle of a part class is independent from the whole class' lifecycle. For example, an organization and workers, worker is a part of organization. In UML, it is shown using a hollow diamond attached to the class that represents the whole (see figure 2c). And the next one is composition which is similar to aggregation relationship but the property is aggregated compositely signifying that the part class lifetime is not independent from the whole class. Its symbol is a solid rhomb with arrowhead (see figure 2d). Next, a dependency is a relationship that signifies a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s). It is pictured as an arrow with a dashed line (see figure 2e). Lastly, the nested relation represents the parent-child relation and its symbol is a circle with cross sign inside connecting with the straight line (see figure 2f).

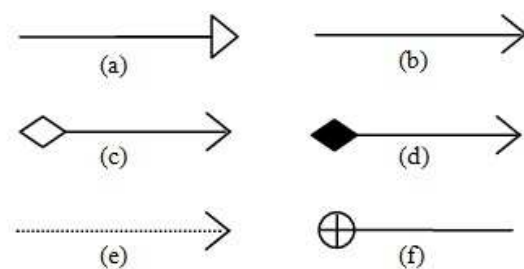


Figure 2: UML Class Diagram's relationships; (a) Inheritance, (b) Association, (c) Aggregation, (d) Composition, (e) Dependency, (f) Nested

2. Qt Creator

Software engineering is always concerning on how to make better software which it is easy to develop, maintain, deploy on many platforms and etc. Qt, a cross-platform application and UI framework, is one of the choices to develop an application once and

deploy on many platforms without rewriting source code. Moreover, Qt is a mature C++ framework for high performance cross-platform software development with extensive features:

- Portability across desktop and embedded operating system
- High run time performance and small footprint on embedded devices
- Cross platform integrated Development tools

In GUI programming, when we change one widget, we often want another widget to be notified. More generally, we want objects of any kind to be able to communicate with one another^[9]. That is why one of the central features of the Qt framework is the signal and slots mechanism, especially for communication between objects. Simply speaking, the signal and slots mechanism are similarly to one person sending text message to some friends. The signals are emitted when the object's internal state somehow has changed and when they are emitted then the slots connected to it are executed in the same way as calling function immediately one after the other. The code following the emit statement of the signal will resume when all the slots have returned. Slots are normal C++ function with one special feature that they can attach to a signal. To handle signal and slots mechanism, Qt makes an extension of a special pre-processor called moc (meta-object compiler) over standard C++ compiler.

With Qt's features, we can develop many types of applications such as web-enabled application, mobile phone, stand alone or client-server application and even the embedded operating systems on its own IDE called "Qt Creator". Qt Creator is also cross platform IDE which include:

- An advanced C++ code editor
- Integrated GUI layout and forms designer
- Project and build management tools
- Integrated, context-sensitive help system
- Visual debugger
- Rapid code navigation tools
- Support multiple platforms

Moreover, Qt Creator is built on an easy extendable architecture, being as the plug-in loader, where all its functionalities are the plug-ins. Internally Qt Creator has a plug-in manager responsible for the cooperation. This allows plug-ins to provide extensions to the functionality of other plug-ins.

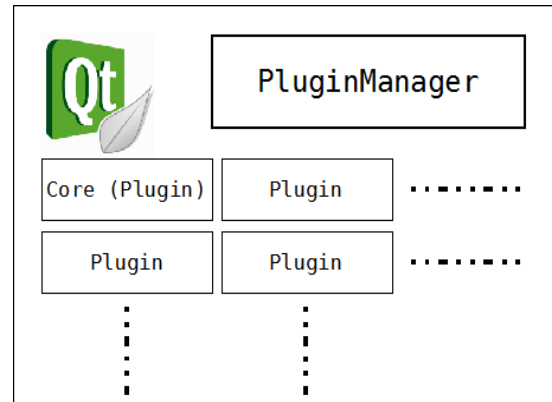


Figure 3: Qt Creator Structure

For instance, the iLocator plug-in uses the CppTools's capability to filter for class names or their methods. The plug-in manager also handles loading the right plug-ins for the user. For example, when you use the Qt Creator to open C++ file then the plug-in manager will load the C++ editor plug-in. The core of Qt Creator is implemented in the Core plug-in which is mainly a container of actions and its basic functionalities like menu actions, etc.

If we want to extend Qt Creator with our functionality, we need a better understanding of the meaning of the plug-in is and how Qt defines a plug-in.

2.1. Qt Creator Plug-in

A plug-in is a small application for extending the capabilities of a larger program and it is also known as add-in, add-on, snap-in and etc. Qt Creator is basically a plug-ins loader which means all its functionality implemented as add-in and it will be run whenever it has been requested.

From the developer point of view, Qt Creator is a big project or application that has many small applications supporting it. That is why in Qt Creator source code package, you will find the file named "plugins.pro" where it lists all the plug-in projects of Qt Creator. In those projects, you will find one class which implemented `ExtensionSystem::IPlugin` and this class is called "Plug-in class".

There are two different types of plug-in in Qt Creator; one is the normal functionality which will be called by the trigger of action, for instance, Find/Replace widget, Locator, and etc. The second one is the plug-in being as Editor like C++ editor, and Resource editor which will be loaded upon type of file that user has been selected.

2.2. How to implement Qt Creator plug-in

As mentioned earlier, Qt Creator can be normal functionality or plug-in being as Editor.

To create the normal function plug-in, we have to implement the plug-in class which has been inherited from the base class `ExtensionSystem::IPlugin`. And in the `initialize` method of the plug-in class we have to use the `addAutoReleasedObject` method if we have the object that we want to bind it to the plug-in. In addition, we also have to connect the plug-in functionalities to particular trigger or action otherwise we could not find the way to access it. Every time we run Qt Creator, it calls all plug-ins' `initialize` method i.e. it loads all its functionalities to be ready for use; therefore, most of initializations for the plug-in should be implemented in the said method. For instance: registering new menu for the plug-in.

Qt Creator has many editors for editing UI (Qt Designer) file, Project file, text file, Qt resource file, and binary file. Developing the plug-in as Editor, we have to add one more step over the implementation of normal plug-in and it is to create the new Qt Creator editor. To create a new editor for Qt Creator, we need to implement three classes which inherit from `Core::IEditorFactory`, `Core::IEditor` or `Core::IFile` and each of them handle for different issues:

- Implement the `Core::IEditorFactory` interface to provide us the methods to help creating the editor object for specified mime-type.
- And the `Core::IEditor` abstract class provides the widget that response to file type or we can call the class implement this abstract as the Editor widget because it will initialize the interface of editor.
- Class implements `Core::IFile` abstract class is responsible for customize the loading and saving data into the file that has been loaded to the editor.

Perhaps you wonder why we do not use the `QFile` class that also can load and save the data into file as well. `Core::IFile` has the method to load contents of given file into an editor widget, while `QFile` is just used for simply loading the content of file into the `QByteArray`. Also `Core::IFile` helps us by notifying us when the file's content in the Editor has been modified. Moreover, `Core::IFile` also handles reloading file content into the editor as well when the content of the file has been modified outside editor. The final phase in creating the editor as a plug-in is constructing the plug-in class where in the `initialize` method we have to add `addAutoReleasedObject` method of editor factory subclass.

3. Architecture and Data Structure

In this part we will discuss about the architecture and data structure of our plug-in UML Tool.

3.1. Model View and Controller

Model View and Controller is a software architecture which isolates the domain logic^[12] from input and presentation(GUI), permitting independent development, testing and maintenance of each^[13]. Each element has its own functionality where:

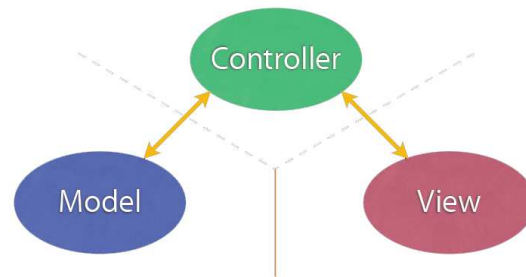


Figure 4: MVC diagram

- Model: the module for storing all the application data
- View: the window or form to represent the model or the Graphic User Interface (GUI) to response to User Interaction or input.
- Controller: the component which is responsible for managing the communication between view and model, i.e. when user changes something through the view, then controller will manage to notify models to make a change as well or the other way around. Or we can call it as application logic^[6].

So we think this architecture suits with our plug-in because it makes our plug-in extendable and also makes it clear to define on GUI, data structure and application logic which is reduced code dependency. As a result, it will save time for our compiling process.

3.2. Data Structure

As mentioned earlier, we will follow the MVC architecture and here we give clear information on data modeling of our plug-in.

Firstly, we will start with the Model part of MVC in our plug-in because we need to use it for storing the information of our plug-in and it is also the best way for the reader to see an overview of our plug-in functionalities as well.

In the class diagram section, you probably have noticed that its entity can have other entities inside it. For instance, package has several classes inside it and also can have other package as well. Or a class has inner classes. In addition, class diagram entities beside package, all have attributes or methods as their

members. And for the methods, they also have arguments or parameters as their members too. Moreover, between two entities (beside package) also has the relationship which has different meaning when it is translated to skeleton code. Basically, primitive types and user defined types are used for declaring attributes, return type of methods and data type of arguments. So our data model should be able to represent or store this kind of information.

Luckily in Qt framework, they provide us the QObject which is the based class of all Qt objects and it has the composite relationship to itself. Then it could help us to solve the problem of inner entity which each entity can have other entity in it. To be able to draw a class diagram, we need class diagram entity and the primitive type while in the class diagram entity has two different types, one is the package and class similar type (class, interface and abstract class). We divide the entity into two types because class similar type is a user defined type which is much similar in usage to primitive type. On the other hand, package is far different; it is used for grouping the class similar type but it's not used to declare other object like the primitive type and class similar type. Hence, the primitive type and class similar type are used in the same way, but it is still not 100% the same because the user cannot declare attributes and methods for primitive type but they can for user defined type (class similar type). For that reason, we are better model the primitive type and class similar type are inherited from a based class "Type" which is inherited from the package's based class "UmlElement" then package and class similar type is still in the same category while their functionalities are different. As the result, we cannot use only the primitive's based class Type for declaring attributes or return type of methods but also user defined type. There's still one more issue which is the relationship between two entities to define. To perform a relationship, we need at least two entities thus the main data for relationship is those two entities, its relationship type, and its direction which we have to define the source and destination. And in class diagram, it also has many types of relationship and each of them behave in different way and some of them have more attributes than the others. Consequently, we decided to create one abstract base class called "Relationship" that class diagram relationship can implement it. As a result, we can use this class to refer to any relationship classes plus benefit of grouping their common parts as well. UML class diagram of our data modeling is as below (details UML of data model, please check Appendix D):

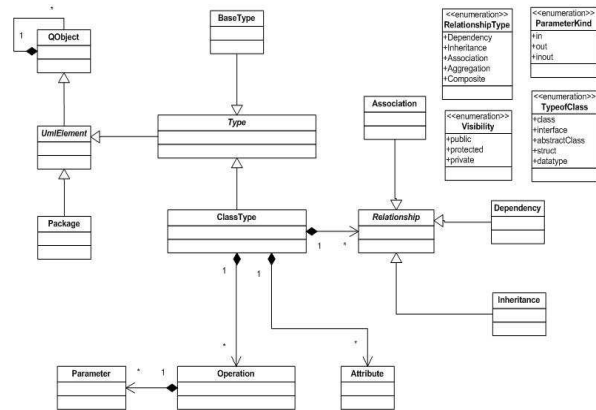


Figure 5: Data modeling class diagram

In Figure 5, perhaps you wonder where the other relations like composite and aggregation are. Based on the given definition in the review section 1 about UML class diagram, we believe that Association, Aggregation and Composite are similar. As a result, we used only class called Association to stand for those three relationship but we used RelationshipType enumeration as their attribute to notify its type (For more details please check Appendix D).

Beside the internal data that will be used for storing the class diagram, we also have to keep in mind that user could save and load it back; therefore, the diagram item should be arranged in the same way or we would say in the same position. And the good way to do so is creating the attributes for the class diagram elements to store its coordinate. Because of our class diagram elements were inherited from the same based class UmlElement, it was a better solution that we created those attributes in this class.

Secondly, we discussed about the GUI or View in MVC. After we got the complete model for the data modeling, we had to consider about how to represent it. Qt framework has the QGraphicsScene class, providing a surface for managing a large number of 2D graphical items which it is really matched with our purpose to draw the class diagram items for representing the data model. Moreover, QGraphicsScene serves as the container of QGraphicsItem which can symbolize lines, rectangles, text, or even custom items, on a 2D surface. Accordingly, it is really in agreement with our objective which we were looking for something to draw the class diagram on. Therefore, we constructed a class called "UmlScene" inheriting from QGraphicsScene which is the scene we use to draw class diagram entities and relationships. The classifier and relationship were presented by "UmlItem" and "UmlLineItem" which both of them inherit from the sub-class of QGraphicsItem.

Certainly, we had the 2D surface for drawing entity as the customized shape but we still need some more classes to render the relationship. Graphically,

relationship is not represented as unmovable straight line and text. The draw relationship it was a bit more complicated because it should be moved when an entity of relationship is moved. Basically, to draw a line, we need just two different points – the source and destination point. But in UML class diagram, source and destination points of relationship can be the same; for instance relationship to itself, which can't be drawn as straight line. For example, in figure 5, look at the QObject class diagram, it has relation to its own. It would be impossible to do so if we use only one single line to represent the relationship. Therefore, the relationship class' view has to contain set of different points that will form the path from one entity to another. To make it more interaction and logically for drawing, those points should be able to move around and the line connect from that point to others must be followed it too. Thus we need other class to stand for that kind of single line which will redraw itself whenever its end point(s) are moved; it is called "UmlSingleLine". We got the class diagram for view part as below:

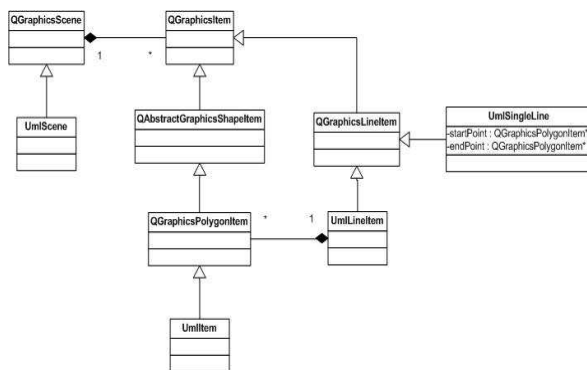


Figure 6: View class diagram

Finally, the Controller module of MVC is a class which manipulates data and view or connects the above classes together. To make sure that each plug-in has only one controller we have to use the singleton pattern. Plus with the signal and slot mechanism in Qt framework then our Controller works as the delegation which responds immediately when some internal data has been changed then it would send a signal to the view for (re)rendering. For example, adding new class item or deleting class item then it will send a signal to the graphic scene to redraw.

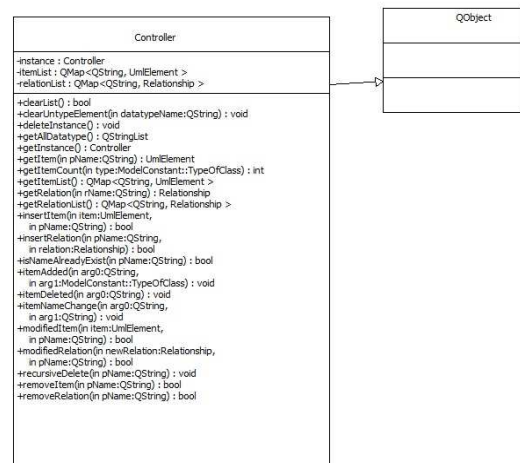


Figure 7: Controller Class

To be able sending the signal from one object to the other in Qt framework, the sender must be type of QObject and plus using the Q_OBJECT macro as well. That is why our Controller class inherited from QObject.

4. User Interface Design

After defining the architecture and data structure of our plug-in, we should consider the user interface because the effectiveness of software depends on user interface too. If we make a good user interface and simple one, then it means that our software is easy to use as well. Because of making class diagram plug-in for Qt Creator, our plug-in interface will be part of Qt Creator that is already well organized.

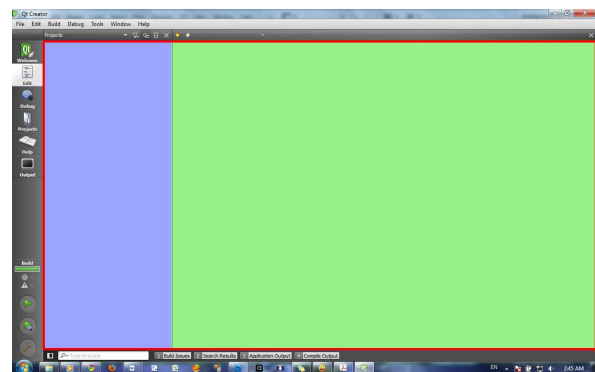


Figure 8: Qt Creator Interface with blog for planning for UML Tool

Our plug-in interface will lay on the red rectangle (see figure 8). Considering of making it easy to use and simple, we divided this area into two parts as main view; one is for list all the class diagram elements and the other one is for the diagram scene. And the class diagram elements list will be in the blue area while the

green one is for the diagram scene that we will draw everything on it because based on the HCI (Human Computer Interaction) principles, user has focused on the center part of the screen^[2] so we were trying to put the diagram scene as centered as possible. One more reason is, in Qt Creator the tool bar is always at the left hand side so we have made it consistency and it is also common way as well for the position of tool bar.

For drawing the input widget of our plug-in we relied on the Qt Creator's feature form designer. We can draw the widget in shape and place where we want it to be so it is really handy and speed up the work as well.

The way to use our plug-in, the user just simply drag the class diagram element then drop on the scene then it will have dialog for inputting that element information (For details information, please find in the Appendix E). We use drag and drop as our main action because it has much research in the Human Computer Interaction focus on how to make a good User Interface (the interaction paradigm). It mentioned that the drag and drop is direct manipulation and simple behavior which everyone can do with small learning curve^[10]. And it provides a simple visual mechanism which users can use to transfer information between and within applications which gives the immediate feedback to the user.

In Qt framework, drag and drop mechanism had been well designed. To make drag and drop work in Qt, what we had to do is just create a class from which implement the QWidget and override some methods such as mousePressEvent, dragEnterEvent, dragMoveEvent, and dropEvent. And for sending data through drag and drop event, we had used the build class in Qt frame work called QDrag because we need the index to indicate which item has been dropped on the scene.

A menu is a group of visually similar words and/or icons on a computer screen that allows the user to select an action to be performed. We thought that it is good idea that we group all the UML tool actions into one menu where the user can easily perform its task because then the user will not spend a lot of time to find where it is plus we placed it in the main menu bar of Qt Creator to simplify it for the user. Moreover, we also attached the menu as popup menu when user presses right click on screen which makes it even more convenience for user to perform tasks. Even so, to make it more flexibility we also provide the shortcut key for some of the menu's tasks as well because some of the shortcut keys have become dominant in the way of people using application already. For example, Ctrl+S is for saving file. Plus some user would like to use shortcut key rather than menu or the other way around.

Consequently, our plug-in interface would be the same as below:

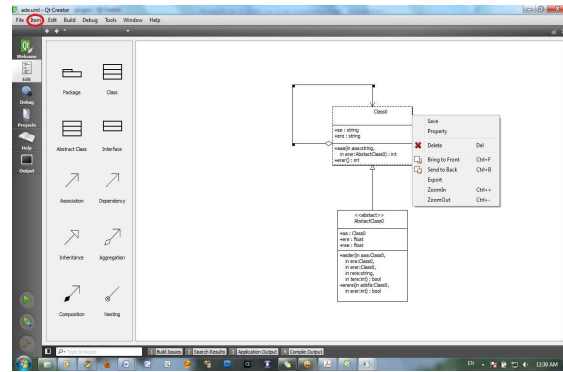


Figure 9: UML Tool Interface

On the upper-left corner, the red circle is where we supposed to add our menu; it was between File and Edit menu.

5. UML Tool Implementation

Here, we discuss about how we are going to implement the UML tool as the Qt Creator plug-in. In section of Qt Creator, we had defined two types of Qt Creator plug-in one is normal functionality and the second one is the editor.

Technically speaking, we developed it as a new Qt Creator editor plug-in called "UML editor". There are some reasons that make us did it this way. First, the nature of UML tool is independently categorized from other Qt Creator editor because it is the graphical interaction rather than the text based editor that Qt Creator had at the moment. In addition, the common sense of using application, the users must be able to save their own data into disk drive. It is true for our plug-in too; UML tool's file can be saved, loaded, modified and etc. Technically, it will be a lot of works to do just to realize the data has been changed or not in UML tool compared to the actual file in the case that we implemented as the normal function plug-in. It is the other way around to editor plug-in for which it has Core::IFile to handle this kind of stuff for us and it was already developed by just emitting a signal. Moreover, we want to explore more about Qt Creator and when we develop it this way then we also know how to implement the normal functionality plug-in as well.

The early sections had defined the architecture and data structure of our plug-in and the user interface already. So now we are better start with the way we saved data into hard drive and the round trip engineering function.

5.1. XML File Handler

After drawing the class diagram, user have a possibility to save it and open for later work. In saving process, we convert the objects and its relation into XML elements and save it to the file. An UML

element is translated into an XML entity, while the name, type, attributes and operations become XML entity's properties and child elements.

There are reasons of using XML instead other format such as QDataStream – the binary file format. First of all, UML model is not just box, lines and texts but also it is structured data and XML is a way of defining languages of structured data. So naturally they are matched. Moreover, Qt provides an excellent XML handler library which facilitated us to write and read the XML back easily. Many other UML tools, such as StarUML^[12] and Altova UModel^[11], also convert the UML diagram into XML. Those tools have additional method, using the XMI – XML Metadata Interchange, to standardize their XML file which gives the modeler the ability to exchange UML model between tools. Due to above motivations and as the beginning, XML was a very good choice. And in future development, this function can be extended by associating with the XMI capability.

Back to saving process, we create an XML element named *Entity* for class and *Relationship* for relationship. In entity element, position of class in the scene, class' type and class' name are translated as the entity's properties. And all the attributes and operations of the classes were saved as the entity's sub-nodes. And for the relationship node, we store all necessary information of the relationship such as source-class, destination-class and role name as the properties of this node.

```
<UMLClassDiagram>
<Entities>
  <Entity type="0" parent="STD" ypos="-343"
    name="Class0" xpos="-835">
    <Attributes>
      <Attribute initValue="" visibility="0" multiplicity="0"
        type="string" name="name" />
    </Attributes>
    <Operations>
      <Operation visibility="0" pureVirtual="0" virtual="0"
        type="string" name="getName" />
    </Operations>
  </Entity>
</Entities>
<Relationships>
  <Relationship roleName="is Kind Of" relationshipType="1"
    from="School" to="Building" />
</Relationships>
</UMLClassDiagram>
```

Figure 10: XML file structure

According to above figure, our XML file is a just plain text. So the content is widely opened for everyone comparing to binary file which is not. And moreover, it can be used in another purpose; for example it can be used with XSLT (XML stylesheet transformation) to display the UML diagram as HTML or plain text.

In loading mechanism, XML file is read and translated back to our data objects without losing any information. Firstly all the entity elements from XML

file are read and translated into instances representing the elements and then added them to the collection in controller. In order to prevent undefined objects, we applied the same action for loading relationship after all the classifiers had been loaded. After adding all the elements and relationship into the class-collection and relationship-collection in the controller, it triggers the view port start drawing the objects and its relationships on the scene based on their saved position.

5.2. Round Trip Engineering

Round trip engineering in UML tool refers to code translation from models and reverse engineering which is a reversed process of code translation.

5.2.1. Code Translation or Export

Exporting the UML class diagram into C++ skeleton code was included as one of the functionality of our plug-in. Our goal was to give users a possible way to export their diagram into C++ code by separating a class into two different files, a header file (.h) and a source file (.cpp).

By splitting a class into two files, it makes the exported code be well organized. Technically speaking, the reason of this division is reducing dependency. That means if the implementation of a method of one class is changed, the compiler is not necessary to compile all the codes in the whole program but only the affected units. If there is only the source file and it is included in another source file, the including will be compiling again when the included file changes. In C++, #include is a straight textual substitute. If everything is defined in the header file, the preprocessor end up creating an enormous copy paste of every source files in the project and feeding that into the compiler.

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include <QMap>
#include <QObject>

class Controller : public QObject
{
public:
  Controller();
  ~Controller();
  UmlElement * getItem(const QString & pName);
  bool insertItem(const QString & pName);
private:
  QMap<QString, int> itemList;
  QMap<QString, int> relationList;
};

#endif // CONTROLLER_H
```

Figure 11: Exported C++ Header File

Referring to the figure 11, the exported C++ headers begins with `#ifndef CONTROLLER_H` and `#define CONTROLLER_H`, together with the last statement `#endif` it is called the include-guard which is used to avoid the double inclusion when dealing with the `#include` directive. After the include-guard, it's followed by the list of `#include`; for instance `#include <QObject>` and `<QMap>`. And the next line is the class definition. According to this figure, class' name is `Controller` and it inherits from the class `QObject`. Inside the class, there are constructor, destructor and all the attributes and operations which are grouped by their visibility. Additionally, by default, the constructor and destructor are placed in the public section like the above figure. Lastly it would be ended with the inner classes if it's available.

In order to speed up the process and to provide well organized code, we used template-based technique for exporting header files. There are two different template files, class and package template which is plain text document. The class template file contains keywords which will be replaced by the actual content later. The body of the class template consists of four sections. Firstly the public section which will be replaced by the public operations, attributes, constructor and destructor. Similar to the public section, the protected section and private section will be substituted by only the protected operations and attributes, private operations and attributes of the class. And the last part is the inner-class section which is for the child class.

Particularly, all the relationships between the objects are also translated into C++ codes. So corresponding to the association, aggregation, and composition relationship, we designed the dialog window for user to select their desired collection to declare those external members. However this also depends on the number of multiplicity of the relationship between two classes; for the relationship with multiplicity 1, the collection will not be applied. Currently, because of STL's popularity and motivating programmers to use Qt framework, we only provided the STL and Qt collections.

With STL collections as the standard C++ library, the exported code can be compiled by a C++ default compiler. Technically, STL collection frameworks are quite powerful, desirable for code re-usage and providing high performance which gains its popularity from programmers. In particularly, both Java and C# collection frameworks are directly patterned after the STL^[1].

Regarding Qt collections, by providing Qt collections as options, it might motivate the programmer to use the Qt framework. In addition, Qt collection frameworks have been carefully designed to provide convenience, minimal memory usage and minimal code expansion^[7] whereas STL's containers

were designed focusing on performance. For instance, Qt's containers are implicitly shared meaning that they can be passed around by values without forcing a deep copy to occur. And if the container is empty, there is no memory allocation taking place. Moreover Qt collection libraries are well documented which is similar to STL but more extensive.

Hence, by providing only STL and Qt Collection, it gives programmer the choice between the performance and memory for their application. Furthermore, the number of choice is few for modeler to choose, however this also makes this function simple for the beginners.

Moreover, when declaring the associated members, there're two types of declaration needed to be considered, by pointer and by reference. In the drawing stage, when a relationship is made, excepting dependency and inheritance, the modeler needs to specify the *Preference Type* which is pointer or reference. This indicates that the associated members will be declared as collection of pointer or reference when exporting.

In addition, to comply with the definition of the aggregation relationship, we had to declare the *part* attribute as a collection of pointer. So when the *whole* class's lifecycle ends, the *part* class is not. For composition case, if it is declared as the collection of pointer, we added codes to destroy that said collection in the destructor implementation. Above all, in the exported codes, the declarations of association and aggregation relationship are the same due to no single accepted definition of the difference between aggregation and association used by the methodologists^[3].

For the CPP file, we did not use the template technique. Commonly, the source file's structure starts with its header file (declaration file) following by the constructor's, destructor's and operations. In the constructor, there're codes to initialize the attributes which has the initialized value. And if it inherits from another class, the constructor of the base class has to be called. For the virtual destructor, there're only the codes to delete the pointer variables. For other operations, the implementation leaves blank. Additionally, the interface will not have the CPP file.

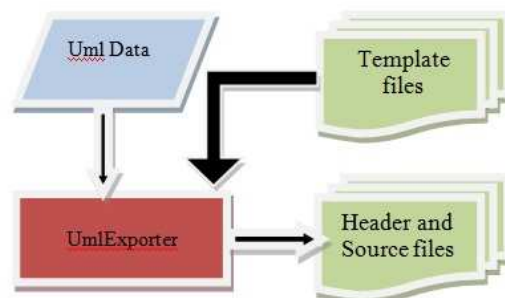


Figure 12: UML Export Diagram

5.2.2. Reverse Engineering or Import

Reverse engineering in this context means, the UML tool reads program source code as input and derives model data and corresponding graphical UML diagrams from it. To do so for our UML plug-in, we had to concern on three main issues: how could we read the whole structure of any C++ class (C++ file is either file with extension .h or .cpp) and translate it into our internal data, how can we translate relationship from code into diagram, and should we read import the C++ file that has been in project file of Qt or the selected directory.

First problem come to solution, there are two ways to read the whole structure of C++ class and translate it into our internal data: writing our own parser to check the structure of C++, and using existing C++ parser or library. Writing our own C++ parser would be beyond our scope and plus with the complexity and multiplicity features of C++ will make it even worst for us to compose one parser for our own. For example, C++ developer has possibility to separate code into two parts declaration (header file) and implementation (cpp file), to declare template, and to overload the operators. On the other hand, the second solution, using existing C++ parser, is the best choice for us. We developed Qt Creator plug-in with Qt framework which is an extension cross-platform of standard C++ compiler with a special pre-processor moc. It could ensure that our plug-in's reverse engine functionality will work on many platforms. And more importantly Qt C++ parser lessens the memory footprint after process. In addition, we wanted to develop UML tool which oriented to Qt framework so it is really the best of choices for us to take Qt C++ parser's capability for our reverse engine module that will give a lot of benefits to future research or extension like extending its feature toward signal and slots.

In order to know how to use Qt parser, we explored some of Qt Creator's plug-in including iLocator, CppTools, and CppEditor as well. As a result, Qt has declared some classes for parsing the C++ code to its editor such as Document, Symbol, Name, and etc. These classes generally read the syntax in the C++ file and give the result as Symbol objects back and they could be namespaces, classes, methods, declarations and etc. Therefore, we can use those classes to read C++ file and create responding class diagram from this code.

For translating the relationship from code to diagram, we have to solve it based on the different type of relationship. Inheritance is easiest one among others because in the Qt parser classes, we can get the base class directly from its sub-class. As a consequence of the multiplicity of C++ and similarity

of the association, aggregation, and composite relationship make it hard for us to define the way for translating them from code to diagram; therefore, we decided to create the pre-defined collection template type for matching and generating those relationships and by the default it will generate the Association relationship because it is more general than the other.

For instance, in the pre-defined type has QVector collection of Qt then if in C++ files that user import also has the attribute declared with this type then it will generate the association relationship to the type which QVector collection holding. And it is really a limitation of our plug-in for handling the associated relation; it is expected to be future development or research. We used the regular expression to get the type which that collection holds because it is able to search for string efficiently by using search pattern. Dependency relationship is much more complicated because of we could not find the good way to judge it from code point of view. For example, if we check dependency based on the include file then the whole class diagram will be full of dependency relationship or infinite loop of dependency relationship because in one header file can have many classes and in that header file also contains more included directives which could lead to infinite recursive. And if taking the declaration into account, it will step into the above path too. Moreover due to the time frame so we leaved it for the future development.

The final point is which choice is the best choice for now between imports only the files in Qt project file or the all C++ file in the selected directory. First, considering about the Qt project file if we imports only the header and source file from Qt project file, it is too specific because the other C++ applications that are not generated by Qt then they cannot be imported or need to create the project file with Qt first. Moreover, we have to write more code as well to be able to read and filter only the C++ file from Qt project file. On the other hand filter for C++ files in the selected directory is more general than import based on Qt project file which means it can import all the C++ projects. Furthermore, the user also can import just only one or two specific part of the C++ project as well and that is really convenient for understanding too. Hence we decided to use the filter for C++ file in selected directory. For this reason, it also makes the import interface looks much simpler to the user as well.

6. Result

As stated in first section of our scope of making UML Plug-in, all the functionalities were implemented. Our Plug-in gives the user or modeler the possibility to draw the class diagram with drag-drop feature. With rich graphical user interface, users are able to drag the diagram round inside the scene. We also provide the

possibility to save the diagram in XML format for later editing of UML diagram and future development (XMI). For the C++ programmers, they can benefit from round trip engineering which exports the model into C++ skeleton code or imports the C++ code and translates it into class diagram back.

Comparing to other existing UML tools which are not only capable to draw class diagram but also other UML diagrams and export into different programming languages, our plug-in has only few functionalities. However, it is just a beginning. Moreover, for the C++ and Qt developer, it gives more advantages; our plug-in with simply functionalities which the user can understand and use it effectively in a short period of times plus the narrow down the complexity in C++ for them by providing choice over collection data type and the usage of reference or pointer in translating code for association, aggregation or composite relationship.

7. Discussion

Nothing is perfect. And our plug-in is the same which need to be improved in the future. The project was completed and all the set goals were completed, but it has restrictions for some functionalities. For example, our plug-in does not have the template classifier which mean user could not use the template mechanism in our class diagram. And multiplicity of the association, aggregation, and composite relationship is not well defined to handle the user input. Moreover, the reverse engineering functionality could not translate the dependency relationship from code into diagram was also known as our plug-in limitation as well.

As a result, there are still many points to be improved in the functionalities and interface parts for the future thesis or research. The important issue would be to break through the limitation such as adding C++ template classifier and solve the reverse engineering for translating the dependency relationship and etc.

Furthermore, the user interface also needs to be improved. For instance, the import feature; in our current system, after importing, all the UML's entities are on top of each other. So modeler needs to move them manually. So to make the system more intelligent, an auto-arranged method should be added. This method will automatically position the imported entities in scene based on the structure and behavior (relationship). Within this method, all the objects will be placed in scene nicely and without being on top of each other. Moreover, window's size has limitation, so when there are many UML entities drawn on the scene the modeler can't see the whole diagram without scrolling around the scene. Even though we had the zoom in and out, but the expander feature of classifier is still needed to be employed. This function will give

the modeler an option to hide or show its own properties (attributes, methods, and inner item). Beside, the user interface for expert should be implemented. The short-cut interface for inputting the attributes or methods by typing the C++ syntax will be the best example here (For more details of future development please find the appendix C).

8. Appendices

- Appendix A: Development environment
- Appendix B: How to implement Qt Creator Plug-in
- Appendix C: Future development
- Appendix D: UML class diagram data modeling
- Appendix E: How to integrate our plug-in and use it

9. Acknowledgements

We are heartily thankful to our supervisors, Ing. Thomas Fannes and Dr. Stef Desmet, whose encouragement, guidance, great advices and support from the initial to the final level enabled us to develop and understand the subject.

Lastly, we offer our regards and blessings to our family, friends and those who supported us in any aspect during the completion of the project.

10. References

- [1] Herbert Schildt. (2004) *The Art of C++*. McGraw-Hill. United States
- [2] Jakob Nielsen & Kara Pernice. (2009) *Eye Tracking Web Usability*. New Rider Press. United States
- [3] Martin Fowler & Kendall Scott. (1998) *UML Distilled Applying the Standard Object Modeling Language*. Addison Wesley Longman. Canada
- [4] Sinan Su Albir. (2003) *Learning UML*. 1st ed. O'Reilly Media. United States
- [5] Object Management Group. (2005) *UML 2.0 Specification*. OMG. United States. www.omg.org
- [6] VCreate Logic. (2009) *Writing Qt Creator Plugins (Beta)*. VCreator Logic. India
- [7] Nokia Corporation. (2009) *Qt 4.6 White paper*. Nokia Corporation
- [8] Nokia Corporation. (2009) *Qt 1.3 Creator White paper*. Nokia Corporation
- [9] Nokia Corporation. (2009) *Qt 4.6 Help Document*. Nokia Corporation
- [10] Abeele, V. Vanden (2008), *Interaction Principles, Styles & Paradigms*, HCI lecture, Presented at KU Leuven, Belgium

- [11] Altova (2010). Altova UModel Tool Integration [online]. 07 May 2010.
<http://www.altova.com/umodel.html>
- [12] StarUML 5.0. (2005). StarUML Features [online]. 07 May 2010.
<http://staruml.sourceforge.net/en/about-3.php>
- [13] *Model-view-controller*, In Wikipedia, Retrieved April 29, 2010.
<http://en.wikipedia.org/wiki/Model-view-controller>
- [14] *Unified Modeling Language*, In Wikipedia, Retrieved May 8, 2010.
http://en.wikipedia.org/wiki/Unified_Modeling_Language
- [15] *Comparison of Unified Modeling Language tools*, In Wikipedia. Retrieved May 12, 2010.
http://en.wikipedia.org/wiki/Comparison_of_Unified_Modeling_Language_tools