UGENE
User-interface GENeration Engine

ARCHITECTURE DOCUMENT

Authors

Jonathan Benn
Leon McKernan-Milon

TABLE OF CONTENTS

1 Reading This Document

1.1 Audience

We assume that the reader is familiar with common software engineering terminology and concepts.

1.2 Definitions, Abbreviations and Acronyms

- **Application:** A software program created using UGENE as a framework. For example, a computerized card game. Composed of a Back-End, Front-End and configuration files.
- **Back-End:** the Application's underlying logic and data; created by the Developer.
- **Client:** one of UGENE's actors; uses the Application.
- **Command Pattern:** a software design pattern characterized by the separation of the recognition of the need for a function call, from its execution.
- **Commander:** a concept from the Command Pattern; the module that recognizes the need for a function call.
- **Database:** a concept from the Observer-Subscriber Pattern; updates its Observer whenever the Database's contents change.
- **Developer:** one of UGENE's actors; uses UGENE to create an Application.
- **Front-End:** the Application's human-computer interface. This part of the Application is largely the responsibility of UGENE.
- **Functor:** a function object; a class that overloads the `operator()` so that its objects can behave just like functions.
- **High-Level Command:** a Controller-level function call, encapsulated and sent by the View, that can invoke functionality inside any of the three MVC modules. Part of the Command Pattern.
- **HLC:** High-Level Command.
- **Invoker:** a concept from the Command Pattern; the module that executes the function.
- **MVC:** Model-View-Controller.
- **Observer:** a concept from the Observer-Subscriber Pattern; subscribes itself to the Database in order to receive updates.
- **Observer-Subscriber Pattern:** a software design pattern whereby an Observer subscribes itself to a Database in order to updated whenever it changes.
- **Panel:** a distinct view of the model; a screen configuration. In part, it is composed of a collection of Widgets.
- **UGENE:** User-interface GENeration Engine.
- **Vector:** a dynamic array whose size can change at runtime.
- **Widget:** a user-interface component that the Client interacts with as part of the user experience. For example, a button.

1.3 A Note About Fonts

Throughout this document's text, concepts and/or class collections will be referred to in capital case (*e.g.* View), whereas individual classes will be written in all capitals and with a special font (*e.g.* `VIEW`), and functions will be written in a special font (*e.g.* `SetCurrentPanel`).

2 Architecture

For UGENE, we’re using an enhanced version of the MVC architecture. In this improved adaptation, the Command Pattern is used for communication between the View and the Controller. For exchanges between the View and Model, the Observer-Subscriber pattern is used, as per the normal MVC.

MVC, and the software engineering patterns we are using for communication, will be described in detail in the sub-sections to follow.

2.1 MVC

In the MVC structural design, the Application is split into three modules: the Model, View and Controller. Each of these components has different responsibilities. The Model holds the state of the Application, the View interacts with the user, and the Controller directs the Application.

UGENE is primarily located inside the View, but has a few important segments extending into the other two modules. These segments allow the three modules to communicate with each other in a standardized and effective manner. They are a direct result of the application of the Command and Observer-Subscriber Patterns.

It’s important to note that while the Model, View and Controller each contain many sub-classes, *only* the topmost classes (the `VIEW`, `CONTROLLER` and `MODEL`) are allowed to access each other directly. This greatly reduces the inter-class coupling across the entire Application. However, because each of these classes has clear and separate responsibilities, performance should not suffer significantly.

We chose to only allow topmost class communication because the requirements call for a distinct separation between the presentation and the Application logic. This requirement makes sense, since the graphics library (*e.g.* OpenGL or Direct3D) and the platform (*e.g.* Windows, Unix, Macintosh) are the most variable parts of the program’s environment. If the Controller and Model (*i.e.* the Developer’s Back-End) cannot access the internals of the View, then they are effectively platform independent.

UGENE imposes the MVC on the Developer. However, this isn’t as bad as it at first sounds. Since UGENE is mostly located in the View, Developers are free to do with the Controller and Model as they will. So long as the names and main APIs of the `CONTROLLER` and `MODEL` stay the same, complete programming freedom is otherwise assured. This is a primary design goal of the architecture, in support of Use-Case 20 and UGENE’s usability requirements.

To help you understand the architecture, we will now provide you with a series of diagrams illustrating various views of UGENE’s operation.

2.2 Module View

Figure 1 illustrates a static view of UGENE's enhanced MVC architecture. In other words, it is a demonstration of the connections that exist between the three topmost classes (`MODEL`, `VIEW` and `CONTROLLER`) and the main program.

The main program has very few responsibilities. This follows good C++ style, where the main is considered a vestigial function to be utilized only when absolutely necessary. The main creates the three topmost classes, passes pointers to them to the appropriate classes, and then repeatedly calls the `CONTROLLER`'s `Update()` function until the program's execution ends. Aside from what is shown in the diagram above, the main program is also in charge of managing input and output with the mouse and keyboard (these weren't shown because they aren't relevant to the MVC architecture).

As the Application manager, the `CONTROLLER` has three main responsibilities:

1. It updates and draws the `VIEW`. Updates might be required because user input has been received, or because the `MODEL` has changed.
2. It receives and interprets High-Level Commands from the `VIEW` (the Command Pattern). HLCs can arrive from the `VIEW` because of user input, or because a new Panel has been loaded and its initialization script has produced them. HLC interpretation might lead the `CONTROLLER` to change the `MODEL`, or instruct the `VIEW` to display a new Panel.
3. It changes the `MODEL` (not shown since this is accomplished via the API created by the Developer), and monitors the `MODEL` to see if the change would require the `VIEW` to be updated. This is part of the Observer-Subscriber pattern.

The `VIEW`'s responsibilities are to:

1. Accurately display what is specified in the current Panel's configuration file. This will probably require that dynamic information be obtained from the `MODEL`.
2. Send HLCs to the `CONTROLLER` (Command Pattern).
3. Obtain updated information from the `MODEL` (Observer-Subscriber pattern).

The `MODEL`'s responsibilities are:

1. To hold the Application's state.
2. To change its state if asked to do so by the `CONTROLLER` (indirect result of the Command Pattern).
3. To inform the `VIEW` of its current state (Observer-Subscriber pattern).

The management of communication across different parts of a medium-sized program such as UGENE is a complex problem. The MVC architecture uses a sophisticated interplay of communication to handily solve this problem. Each of the three MVC components has three clear responsibilities and an unambiguous relationship with its peers. We believe that this results in greater cohesion across the entire application, and hence reduced coupling and improved maintainability. In addition, using the MVC architecture conceptually eases the Developer's job of creating a Back-End for the Application.

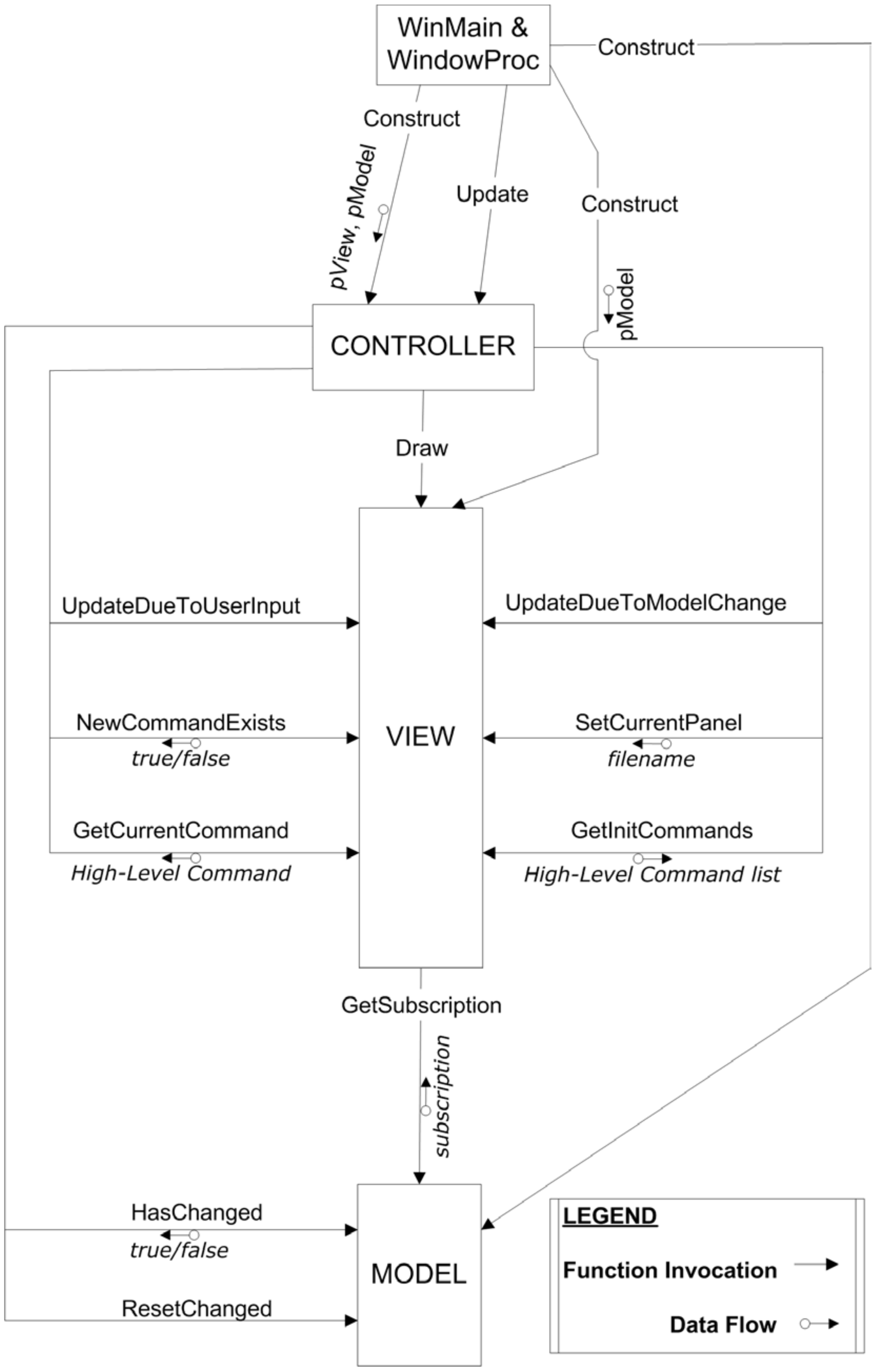

Figure 1: Module View of UGENE Architecture

2.3 Execution View

Where **Figure 1** illustrated a static view of the MVC architecture, **Figure 2** illustrates a dynamic view, as if UGENE were currently executing. This latter diagram shows one important element that couldn't be described statically: the order of operations. In addition, the interactions with the user, input devices (mouse and keyboard) and Panel configuration file have been considered.

As can be seen in the diagram, when the View is told to update itself based on user input, it queries the `MOUSE` and `KEYBOARD` in order to discover their current state. It uses this information, combined with its knowledge of its own current state, to interpret which High-Level Command to send to the Controller (if any).

During runtime, the View will occasionally be told by the Controller to change its current Panel. When this happens, it must load the indicated filename, and parse the file into information that it can understand. The View's behavior with regard to which HLCs it sends to the Controller, when it sends them, and what data it subscribes for in the Model, is dependant on the contents of the Panel file.

In **Figure 2** we've shown the most complex runtime example, where the user has entered input that calls for a new Panel to be displayed. Most of the time, since the user is very slow relative to the computer, the user will not have given any input at all. In this case, steps 1 and 2 don't occur. The View will have no HLC to transmit in step 6 (because step 5 returns nothing of interest), and so steps 6 to 10 are skipped.

Even if there is user input, it doesn't necessarily call for a Panel switch or Model change. If the current Panel is kept, steps 7a and 8 are skipped. If the HLC doesn't call for Model change, then steps 7b, 9 and 10 are skipped.

Consequently, steps 3, 4, 5, 11 and 12 are the only obligatory steps. Step 3 is invoked by the main program loop, and the remaining obligatory steps are executed as a result. It would seem that for best efficiency steps 4 and 5 could be eliminated as well, however, this is not the case. In fact, when the user presses and holds down a keyboard key (or mouse button) a message is only sent to the WindowProc *once*. Failing to run steps 4 and 5 every frame would result in the View not recognizing that the key is still pressed.

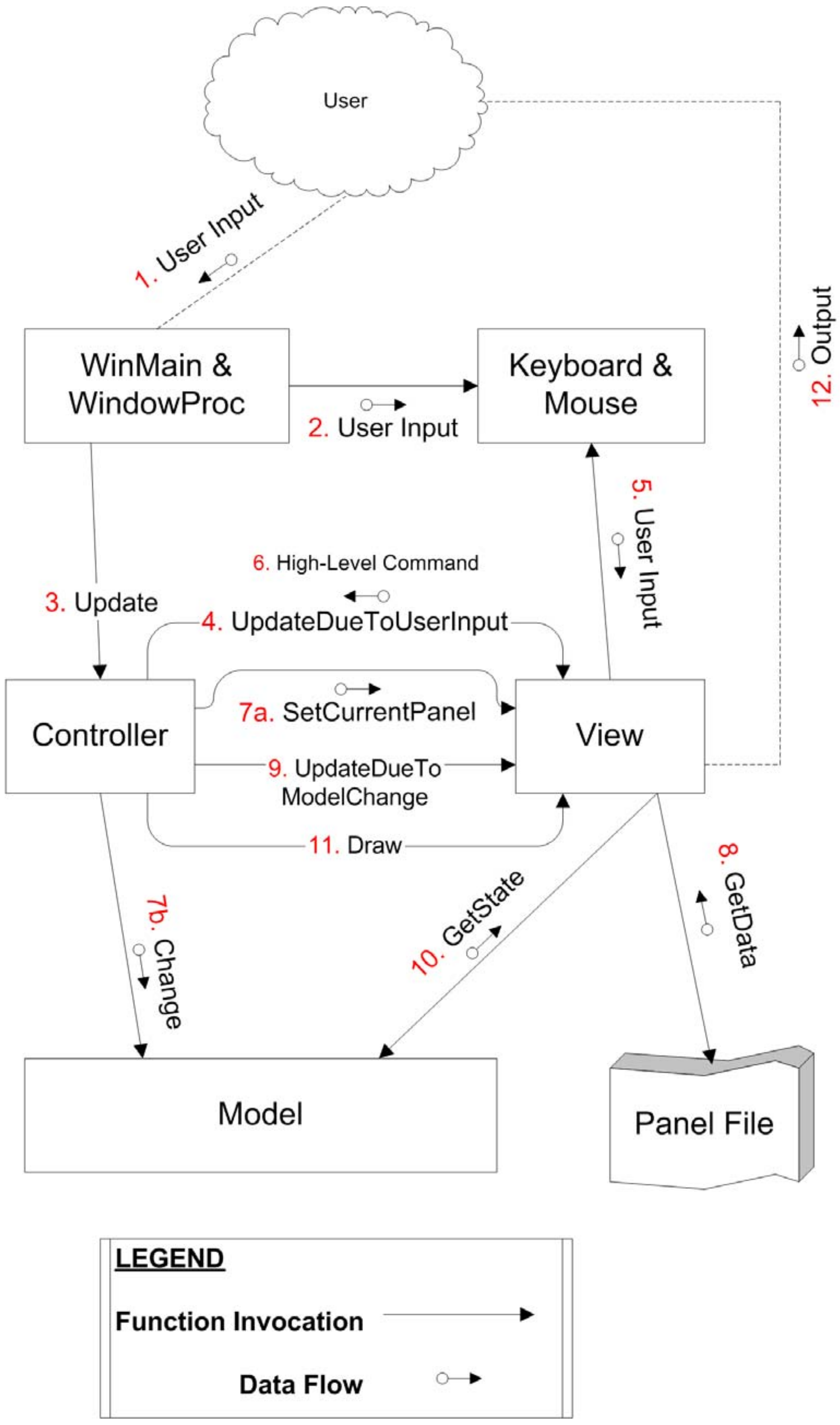

Figure 2: Execution View of UGENE Architecture

2.4 The Observer-Subscriber Pattern

The idea behind the Observer-Subscriber pattern is that a software module (called the *Observer*) *subscribes* to a second module called the *Database*. Whenever the Database changes, the Observer is notified, and then through its subscription obtains only the information that is relevant to it. Thus, the Observer-Subscriber pattern is a framework for allowing the Observer to stay up to date with the state of the Database, without the Database needing to know what information is important. This reduces coupling between the Observer and Database. If the subscription is dynamically determined at runtime (as it is in UGENE), then the coupling between the two modules is virtually eliminated, since the Observer doesn't even know what it will be observing.

The Observer-Subscriber pattern's main use is when the Observer is the software's user-interface, and the Database is the software's internal state. The pattern allows the software developer to create the software's underlying data structures without concern for how those structures might be displayed to the user. The Database developer's only responsibility to the user interface is to ensure that the Database provides a complete API for accessing its data.

The Observer-Subscriber pattern is also very efficient because it only creates an overhead when the Database changes. Of course, this assumes that the Database doesn't change constantly—if it does then the software architecture needs to revised, or the loss of efficiency needs to be accepted.

In UGENE, the View is analogous to the Observer, and the Model is analogous to the Database.

The benefits to UGENE of using the Observer-Subscriber pattern are significant. Thanks to Panel configuration files, the View's subscriptions to the Model are determined at runtime. This means that Developers are free to design the Model as they wish, and that they *never* have to modify the View's code. The Developer's interaction with the View is completely abstracted. As an added bonus, the Developer can modify the user-interface without even needing to recompile the source code.

2.5 The Command Pattern

The essence of the Command Pattern is the separation of the recognition of the need for a function, from its invocation. There are two software modules at play here: the *Commander* and the *Invoker*. This pattern requires that a function call asked for by the Commander be encapsulated in another form (*e.g.* a string, a vector of strings, or a functor) and then passed to another module, the Invoker, where the function is actually executed. The Commander doesn't know what its commands mean. For example, a string has no intrinsic meaning to a computer unless it is associated with something, such as a function. At the same time, the Invoker only knows that it has been given a function to invoke—it doesn't know why. Hence, the "why" is contained entirely within the Commander, and the "what" is within the Invoker.

The Command Pattern's main use is for decoupling the software's user-interface from its logic. In this case the Commander would be the user-interface, and the Invoker would be the software's internal logic. The Commander determines which commands to send based on its interaction with the user. For example, if the user were to click a button labeled “Quit”, the Commander would send a “Quit” message. The Invoker then interprets and executes that command. For example, after receiving a “Quit” message, the Invoker would send a confirmation message to the user.

The Command Pattern is potentially very efficient because it only requires information flow between the Commander and Invoker when something important has happened (which, with users being so slow compared to computers, is rather infrequent). However, the Command Pattern’s efficiency is greatly affected by the function encapsulation it uses. For example, reinterpreting functions as strings may be very inefficient, especially if input variables need to be cast back and forth from string form to their native form and vice versa. Functors are a potential solution to this problem, however, they are complex and difficult to use properly, and so are no silver bullet.

In UGENE, the View is analogous to the Commander, and the Controller is analogous to the Invoker.

The benefits of using Panel configuration files are enormous when combined with the Command Pattern. Not only does the View not know what its HLCs *mean* when it sends them to the Controller, but at compile time it doesn’t even know what they *are*. The Developer has complete control over how the View will interact with the Controller solely through the creation and modification of Panel configuration files. Once again, the Developer never needs to modify the View’s code directly.

Used in conjunction with the Observer-Subscriber pattern, the Command Pattern ensures that the View is decoupled both from the Application’s underlying data representation, *and* its underlying logic. Since UGENE is primarily located inside the View, this means that UGENE is very comfortably decoupled from the Developer’s Back-End. Developers are free to create whatever Back-End they want, and all they need to do is create an API (in the Controller and Model) and Panel configuration files in order to connect their Back-End to UGENE and take advantage of all of its built-in functionality.

3 Meeting Quality Objectives

3.1 Use-Cases

Supporting UGENE's Use-Cases is the primary purpose of this architecture.

The architecture's decoupled View ensures that the Developer has complete freedom to develop whatever Back-End (Controller and Model) he or she wants for the Application. The use of the Command and Observer-Subscriber Patterns makes creating an API easy. Constructing Panels is also easy because it can be done with a simple text editor, and doesn't require any programming or knowledge of UGENE's internals. All of these factors support Use-Case 20 (and consequently Sub-Cases 2020 and 2040).

In Use-Case 40, the most important event is when the Client interacts with the Panels and their Widgets. This architecture does not attempt to meet this need directly. However, by allowing great flexibility in UGENE's Widgets, we hope that they will be able to be created in such a way that they will be very useful and usable.

Use-Case 60 has been deliberately left out of the architecture at this time. The reason is simple: we don't know how to deal with it. We believe that our architecture is flexible enough to allow the addition of a networking component to UGENE without causing serious difficulties. We anticipate that the main problem will be at the implementation level, where thread-safety will have to be guaranteed.

3.2 Features

UGENE's architecture makes the implementation of all of its functional requirements (*i.e.* features) possible. However, its detailed design and implementation will determine whether or not those goals are achieved. For example, while the architecture allows the creation of a variety of widget types, only implementation will actually see their construction.

3.3 Qualities

On the other hand, many of UGENE's quality objectives (*i.e.* non-functional requirements) are met directly by this architecture. This is obviously the case for the architecture section of the requirements. In particular, maintainability is a hallmark of our design, as evidenced by our constant efforts to reduce coupling between the different architectural modules.

Unfortunately, this architecture cannot guarantee that UGENE will be reliable—that remains an implementation quality issue.

There remains one important issue that needs to be discussed: portability. UGENE's View and main are device dependant and OpenGL dependant, whereas the Controller and the Model are device and OpenGL independent. This means that only the View and main will need to be modified in order to port this program to other platforms. Of course, that’s assuming that the compiler stays the same. A platform change that requires a compiler change will necessitate code revisions, and there’s nothing that this architecture can do to stop that.

Finally, a change in language requiring the introduction of Unicode (e.g. Chinese language support) would probably mean considerable change to many classes, and not just in the View. We’re not sure at this time how well the STL deals with Unicode, so future non-English language support is certainly a risk.