UGENE

User-interface GENeration Engine

IMPLEMENTATION USER MANUAL

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.
- **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.
- **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.
- **MVC:** Model-View-Controller.
- **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.
- **STL:** Standard Template Library.
- **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 Implementation

This section will discuss UGENE's detailed design, its implementation strategy, and offer information useful to UGENE's maintainers. This section will *not* cover problems encountered and their solutions—that will be dealt with in **Section 2.6**, on page 9.

2.1 Implementation Strategy

From the very first day that we started working on UGENE, we had a prototype. We began with an old computer graphics project, stripped out the quick & dirty code (which was a good 80% of the code), and then restructured what was left. This became UGENE's initial prototype. At this point in time, it wasn't able to do much more than open a window and display a textured rectangle.

Our approach to implementing UGENE could best be described as a critical-first process. It was neither top-down, bottom-up, nor middle-out. What we did was implement whatever seemed the most important (*i.e.* critical) at any given moment. An advantage of this approach is that whatever is developed first gets tested the most. Hence, our project's critical code was getting the most testing, without any extra effort needed.

This critical-first approach was driven by an iterative process. We would modify the prototype to add new functionality, to refactor our design, or (rarely) to correct a defect. Once we made a change (and if possible, while we were making the change) we would rebuild the project and run it. These frequent system tests were an excellent opportunity to shake out obvious bugs and code deficiencies. We felt that in light of the ease of performing system tests, we wouldn't bother with unit tests under most circumstances. We felt that a pass on the system test meant a pass on the unit test.

To ensure that our code was correct, we instead performed code reviews. Not only did these allow us to find programming errors, but they also helped us increase our understanding of the system. Periodically, whenever we were confused, we would assess our architecture from a high-level perspective in order to understand where we were at and where we were going.

Altogether, the implementation approach and process were a very successful combination. In spite of the fact that most of our programmers were very busy people, we still managed to nearly meet our (admittedly ambitious) goal of reaching Beta by the end of April, 2004. Most importantly, throughout the process we had constant feedback of our progress from our prototype. Even though we're a little behind schedule, we still have an interesting program to hand in.

2.2 View

In this section, we will take a look under the hood of the View, and see how it is constructed. To get a high-level understanding of the View, consult the Architecture Document.

The View contains three subcomponents, as well as a handful of miscellaneous classes. The subcomponents are:

- User Input-Output Subcomponent
- Widget Subcomponent
- File Input Subcomponent

All of these subcomponents, and those classes that didn't fit into any group, are illustrated in **Figure 1**, and described below. Remember that one of UGENE's architectural constraints is that only top-level classes (the `MODEL`, `VIEW`, and `CONTROLLER`) are allowed to communicate with each other. Hence, all of the classes inside the View are not allowed to access any class outside of it. Many of them are connected to the `VIEW`, however, and thus can communicate indirectly with the rest of the Application.

- **User Input-Output Subcomponent:** This group of classes is concerned with obtaining input from the user, and providing output. The `CAMERA`, `KEYBOARD` and `MOUSE` classes all use the Singleton design pattern, meaning that there is only one instance of each of them. `FONT` and `DISPLAY_LIST` are used to display text and images/shapes, respectively.
- **Widget Subcomponent:** This group of classes is interested in the *states* of, *transitions* of, and *feedback* provided by UGENE's user-interface. Each derived class of `UI_WIDGET` is a state machine with very specific requirements, depending on its nature (*e.g.* a button has different behavioral requirements from a pop-up list).
- **File Input Subcomponent:** This class group is concerned with loading and parsing Panel configuration files.
- **`FPS` (Frame Per Second) Class:** Another Singleton class that keeps track of the Application's current frame rate and frame interval (the time between frames). This information is of more than academic interest, as it is used by the `CAMERA` to create uniform motion (otherwise motion would be faster on a more powerful computer).
- **`MATH_VECTOR`:** This is a simple custom 3D vector class, used by the `CAMERA` to manage positions in 3D space. It could easily, and perhaps should, be replaced by a library implementation.

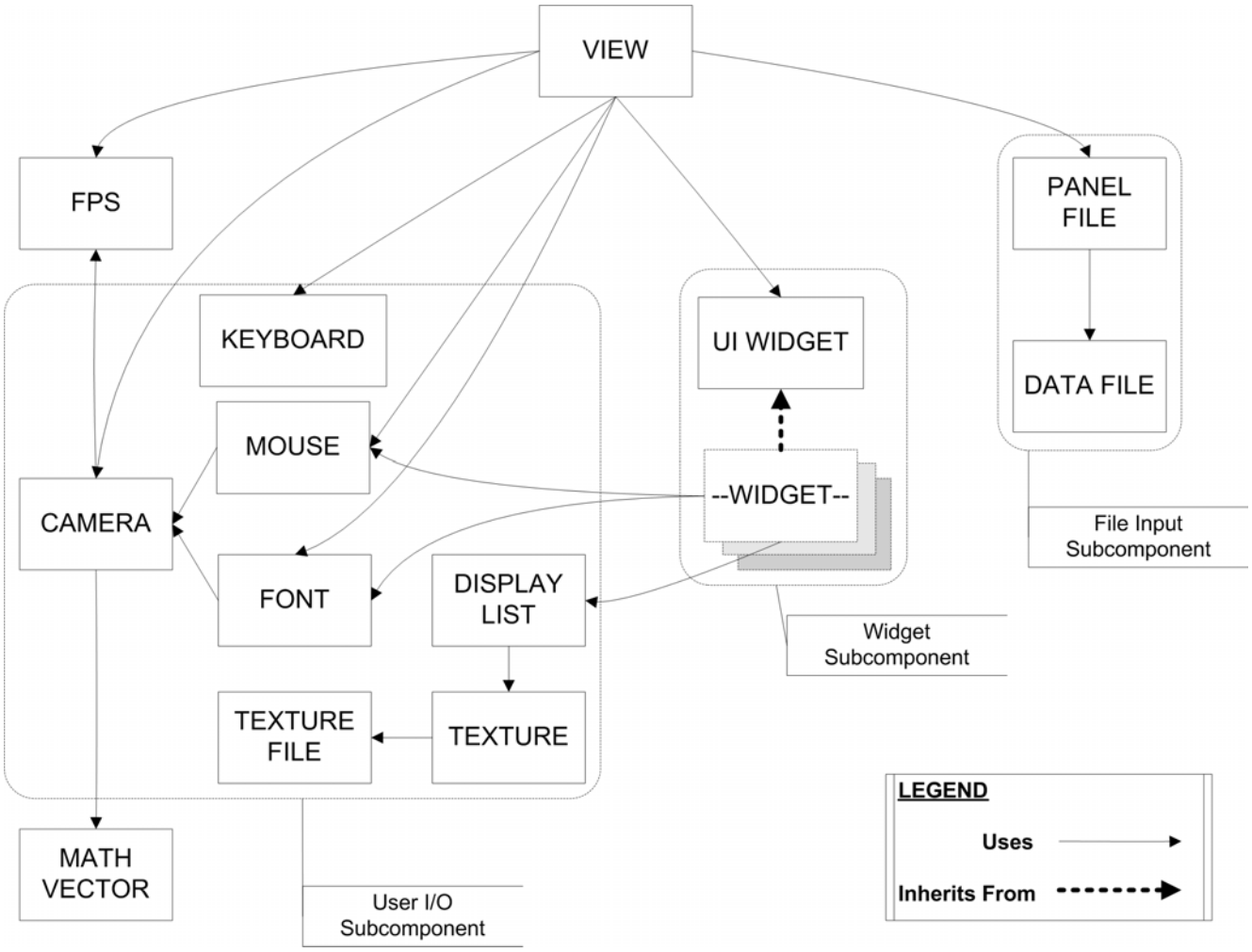

Figure 1: View Class Diagram

2.2.1 View Design Considerations

There was a lot of thinking, work and rework that went into the View's design. Following is a rough listing of the ideas that went into it:

- The `VIEW` class must be completely independent of the rendering engine (*e.g.* OpenGL). This means that there shouldn't be any system-dependant code (*e.g.* `glClear(GL_COLOR_BUFFER_BIT);`) inside it. This limits the number of classes that will need to be modified if we use a new graphics API like Direct3D. Besides which, the `VIEW` is very high-level, and doesn't need to be cluttered by low-level graphics calls anyway.
- The `VIEW` should access the `FPS` class only for displaying the frame per second rate to the screen.
- The `VIEW` class should *not* use `DISPLAY_LIST` class directly when displaying images—instead, it should access an `IMAGE` class, which is a class derived from

UI_WIDGET. Why? Because IMAGE is non-trivial: amongst other things, it can load data from a file just like any other widget. Using DISPLAY_LIST directly inside the VIEW would hard-code the images, which is unacceptable.

- Groups of widgets can be enabled or disabled. Disabled widgets do not receive user input
- In order to determine which widget gets input from the KEYBOARD, the VIEW keeps track of the active widget. The knowledge of whether or not an object is active is always held within that object (information expert pattern).
- The VIEW keeps track of groups of widget, to make it easier to enable or disable them all at the same time.
- UI_WIDGETs are drawn in *reverse* order, this way widgets that are placed first in the screen configuration file are drawn last, on top of the others. Of course, this is only true if the depth buffer is not being used.

2.2.2 UI Widget Architecture

As can be seen in **Figure 1** and the Architecture Document, we avoided using inheritance as much as possible. We only used it once, and even then only for a two-level hierarchy! Many C++ programmers go inheritance-crazy with their designs and inherit everything under the sun. However, experienced C++ programmers understand that inheritance is the strongest form of inter-class coupling, and that it should only be used when absolutely necessary. The preferred alternative is encapsulation (a “uses” relationship, as opposed to the “is a” relationship of inheritance).

In the case of the View’s Widgets, inheritance was absolutely necessary. The Object Factory pattern, which we used to allow dynamic object loading from a Panel configuration file at runtime, required the use of a base pointer to derived objects. At the same time, the capability to use the virtual keyword on common functions such as Update() and Draw() was of great interest. This allowed the VIEW class to deal with all of the widgets through a base pointer to a UI_WIDGET. Hence, the VIEW was protected from having any knowledge of how the Widgets functioned.

Despite these obvious benefits of using inheritance, there were still some negative side-effects. A limitation of the C++ language is that one cannot combine function templates with virtual functions. This is a silly limitation that makes writing compilers easier, but makes an implementer’s life more difficult. The result of this is that we were forced to frequently use strings as a lowest common denominator in communication between the Controller and View, and the Model and View. This is both inelegant, and potentially very inefficient. We are still investigating ways to get around this limitation, perhaps by avoiding using the virtual keyword in certain functions (especially GetAttribute and SetAttribute). It seems that an advanced programming concept called *type lists* might be of help.

2.3 Controller

From an implementation standpoint, and as far as UGENE is concerned, the Controller has only three responsibilities:

- To update the `VIEW` and `MODEL`
- To tell the `VIEW` to change Panel
- To contain an interface of HLCs

It is assumed that Developers will add whatever functionality they see fit when they create the Application's Back-End.

It's useful to mention the mechanism the `CONTROLLER` class uses for dispatching High-Level Commands. But before beginning that, we must first describe what an HLC *is*. This is a vector of strings, where the first string is the name of the function, and each string thereafter is a parameter to the function. For example:

```
["SetNumberOfPictures3i", "15", "3", "80"]
```

Notice the use of a pseudo-Hungarian notation in the function name. This is because when Developers are writing down functions in the Panel configuration file, it will help them know how many parameters to give to the function, and of what type each of them should be.

Here is where things get interesting. You might be asking, "How does this vector of strings get converted into a function?" The answer is through a concept that we invented (although others may have come up with the idea independently) called the Function Factory Pattern. In this pattern, an STL map is created that links strings to function pointers. This way, if the map is given a string, it can return a function pointer, which can then be executed. The function called is passed the vector of strings as a parameter, and since it knows both how many parameters it needs and what their types are, it is in charge of converting the parameters to the appropriate type and then using them.

There are two functions at work here: `InitializeFunctionList` and `RunCommand`. Here is the order of operations from the Developer's perspective:

1. The Developer writes custom functions and places them inside the `CONTROLLER`. These functions are the High-Level Commands that the `CONTROLLER` can understand and execute.
2. The Developer then uses the `InitializeFunctionList` function to fill in the Function Factory map's entries with pointers to the custom functions he or she wrote.
3. Developers may now write Panel configuration files, using the strings associated with the `CONTROLLER`'s functions (*e.g.* `SetNumberOfPictures3i`).
4. These strings from the Panel configuration file are then dynamically run by the `CONTROLLER` through the `RunCommand` function, which simply uses the

Function Factory map to match the string to the function pointer, and then calls the appropriate function.

2.4 Model

Like the `CONTROLLER`, the `MODEL` class is quite simple, and is meant to be extended by the Developer. It uses the same Function Factory Pattern to communicate with the view, except instead of being called HLCs, its function calls are named Subscriptions, because they are part of the Observer-Subscriber Pattern. In addition, the `VIEW` is not allowed to modify the `MODEL`, hence all of its subscription functions are declared const. Except for these minor differences, the idea is exactly the same as for the `CONTROLLER`, see **Section 2.3** for details.

2.5 Useful Information For Maintainers

Here is a list of some tips the maintainers should keep in mind:

- We used CodeWarrior instead of Microsoft Visual C++ for a reason. CodeWarrior has a *much* better compiler, and much better STL support. The `OBJECT_FACTORY` class would not compile under MVC++, and we considered it to be a vital part of the code that had to be included in the final product.
- Using CodeWarrior made porting other libraries into UGENE more difficult, since they rarely, if ever, included CodeWarrior project files. This is a price we had to pay for a better compiler.
- Speaking of project files, take very good care of the project file (`UGENE.mcp`). It contains a lot of vital information, including all of the project's `.h`, `.cpp` and `.lib` files. Plus, it includes the OpenGL libraries and some non-obvious system library files that are needed by the `TEXTURE_FILE` class.
- The project directory contains some required `.DLL` files that must either be in the same directory as the executable, or in the Windows system folder (the exact location of which depends on the version of Windows).
- The Requirements and Architecture Documents are the result of a lot of hard work. Read them carefully, and keep them up to date.
- We've deliberately not talked about the `UI_WIDGET` and derived widgets too much. This is because they are still in great flux. We will document them as soon as we understand how best to organize them.

2.6 Problems Encountered

Our major problems were both directly related to the integration of external libraries into UGENE. This difficulty is precisely why UGENE's main vision is to not require Developers to need to do that themselves. We had a few other miscellaneous

problems along the way. The following problem list is ordered from most awful to least awful:

Integrating libpng: on January 20th, 2004, we began trying to integrate the libpng library into UGENE. This library would allow us to load the PNG image file format, and take advantage of its seamless and highly performing transparency functionality. We failed miserably, and by the month of March we hadn't made any progress at all, despite three attempts. The reason why it was so difficult was that we were using a non-standard and rarely supported compiler: CodeWarrior. Why was a whole other story (it's simply much more ANSI compliant than Visual C++). On March 5th Jonathan tasked Johnny El-Hajj with the job of compiling libpng under CodeWarrior, and by March 16th he had succeeded. Jonathan integrated his work into UGENE the next day, and we could now load PNG files.

Fonts in OpenGL: our second major problem with UGENE (which was chronologically solved before the libpng issue) was with loading text to the screen. OpenGL has no native capability for text, and in fact it is surprisingly annoying to implement. It took a lot of research to find a good OpenGL text library that could load TrueType font files (Freetype), and a middleware solution that would allow us to use it conveniently (OGLFT). This compilation and integration was accomplished on December 26th. Over the next few days, Leon was principally involved with using OGLFT to create the very usable TEXT_BOX class.

The crash bug: soon after our Alpha2 release (around the end of March) we introduced a mysterious crash bug into UGENE. We had no idea what was causing it. We (Leon and Jonathan) implemented logging to try to find the problem, and eventually the problem fixed itself when we refactored the VIEW. We moved the Panel file parsing code from the VIEW to a new class called PANEL_FILE, and the bug went away. There are still some crash bugs here and there, but this one was the worst and was taken care of on April 17th.

Virtual error: we had a major memory leak and memory corruption bug that was due to a single missing keyword in the UI_WIDGET abstract base class. When Leon had first made the base class, he had omitted the "virtual" keyword from the destructor. This meant that derived classes using the base pointer were only being partially deleted. Jonathan researched inheritance, discovered, and then corrected this problem on March 13th.

2.7 Project Assessment

Overall, the UGENE project has been a success. We initially wanted to reach Beta status by the end of May 2004, however, we have instead reached a late Alpha stage (we're nearly at Alpha3). This is only slight schedule slippage, and in our mind is very acceptable given the other pressures that each and every one of our developers has been under during this project's lifetime.

We have a working prototype, which illustrates a lot of the functionality that UGENE will have as a finished product. The prototype is reasonably stable, and functions on all of the modern Windows platforms. It has taken over a year of part-time effort to reach this point, and given the project's complexity, it will probably take at least another year to finish UGENE. At this time, our confidence is very high that UGENE will meet with success at the end of the road, as we've crossed paths with and eliminated all of our major risks.

There are several factors that contributed to our success:

Our process. The creation of an early prototype made it easy for us to constantly system test UGENE and prevent many major errors. Unit testing was hardly ever required. We were able to learn as we went along, which allowed us to start the project and accomplish a lot, even though at the beginning we had a very low understanding of the problem space.

Our teamwork. Working side-by-side on most of the code, Leon and Jonathan were able to produce code of a higher quality than either of us could have done alone. I believe that this saved us a lot of rework. Working together also helped keep us both motivated, a factor that is not to be underestimated. Also, SourceForge made it easier to keep track of what needed to be done, and who was doing it. Even though other team members did not contribute a lot to UGENE, the contributions that there were happened because of SourceForge.

Our organisation. This goes beyond using SourceForge to organise team members. Throughout the development process, we always had a good high-level understanding of UGENE. This led to more focused development efforts, and better expenditure of our very valuable and scarce resources. Documentation was a strength of our project, rather than an annoyance and a weakness (as it is in so many other projects).

2.8 Coding Convention

We attempted to use a consistent coding convention throughout UGENE's development. Its specification can be found in the sub-sections to follow.

2.8.1 Indentation

The code should be indented according to its nesting level. An indent is four spaces. The opening brace begins on the line after the control structure. Braces should always be matched on the same indentation level.

Example:

```
if (x > 1)
{
    if (y > 1)
    {
        cout << y;
    }
    else
    {
        cout << x;
    }
}
```

2.8.2 Comments

Comments should contain information that is relevant to the code and that is not trivial or obvious from the code itself. For example, adding a comment that mentions a function's parameter types and return value types is totally worthless (and yet we've seen this numerous times in professional C code…).

In particular, functions and classes should have comments indicated *what* they do, and *how* they work.

Function implementations should have comments indicating *why* they have been programmed in a particular way. Comments must be high-level descriptions of the code and the programmer's thought process.

2.8.3 Header Files

The header file is the specification for a class and therefore needs to be well commented.

Each header file should begin with comments using the following structure:

```
/*************************************************************
  ----------------
  CLASS_NAME CLASS
  ----------------

    [[Description of a class's purpose and capabilities. ]]
-------------------------------------------------------------------

  -----
  USAGE
  -----

   [[Examples of using the class's main functionality. This code
   should be able to compile.]]
-------------------------------------------------------------------

  ------------------
  EXCEPTION HANDLING
  ------------------

[[Mentions the errors that can occur and which exceptions will be
thrown.]]
-------------------------------------------------------------------

  ----------------
  REVISION HISTORY
  ----------------

  Date             Author                   Change
  ----             ------                   ------

[[This is very important, as it gives us someone to blame, and
it's also very, very useful for tracking progress]]
-------------------------------------------------------------------

  -----
  TO DO
  -----

    [[Point form list of things left to change or add in this
class. It also contains ideas for this class design that need to
be considered.]]

*************************************************************/
```

2.8.4 Temporary Code

In an iterative process where a working prototype is being modified, it often becomes necessary to add temporary "debug" code in places where it's not necessarily supposed to be. This can make it easier to perform an effective system test, for example.

All such temporary code should be bracketed as follows:

```
// ##################################################
TemporaryCommand();
// ##################################################
```

2.8.5 Naming Conventions

Class Names

Class names should be nouns and should be descriptive.
All letters in a class name should be uppercase with words separated by underscores.

```
Example: UPPERCASE_WITH_UNDERSCORES
```

Constants

All letters in constants should be uppercase with words separated by underscores.

```
Example: UPPERCASE_WITH_UNDERSCORES
```

Variable Names

- This includes all types of variables.
- Variable names should be meaningful and descriptive.
- One-letter variables should be reserved for extremely short-lived variables such as counters within loops.
- Variable names should start with a lower case letter and the first letter of internal words should be capitalized.

```
Example: firstWordLowercaseButInternalWordsCapitalized
```

Pointers

These variables follow the standard variable naming convention except they start with "p". Note: The first word after the "p" is uppercase, since this is considered the first word.

```
Example: pSameAsOtherVariables
```

Class Member Variables

These variables follow the standard variable naming convention except they start with “m_”. Note: The first word after the “m_” is lowercase.

```
Example: m_sameAsOtherVariables
```

Class Member Pointers

These variables follow the standard variable naming convention except they start with “p_”. Note: The first word after the “p_” is lowercase.

```
Example: p_sameAsOtherMemberVariables
```

Class Member Functions

- The name of a method should be or start with a verb that is descriptive of the action performed by the method.
- Method names should start with an uppercase letter and the first letter of internal words should be uppercase.
- If a member function accepts zero parameters, do not add a `void` between the function’s parentheses.

```
Example: AllWordsUppercase()
```

3 User Manual

UGENE's primary user, the Developer, is expected to have technical expertise with C++. Therefore, this user manual will be written with that in mind. It will be difficult to describe UGENE's user interface, given that it doesn't really have one. However, we will do our best. Understanding UGENE's UI means understanding its Developer Use-Cases. There are three that are applicable: Create Application, Create an API, and Construct Panels.

3.1 Create Application

In particular, the first step of this Use-Case, "Create a Back-End for the Application", needs to be discussed. We expect that Developers will:

- Not modify the View in any way whatsoever.
- Add whatever functionality they feel is necessary to the `CONTROLLER` and `MODEL`, without modifying any of the pre-included functions. This of course includes the ability to add new sub-classes.

We can't say any more than that about this Use-Case, except that Developers have complete freedom to develop whatever sort of Application logic and data representation they wish.

3.2 Create an API

The procedure is slightly different for the `CONTROLLER` and `MODEL`. Each will be described in turn.

3.2.1 Create a CONTROLLER API

Once a new function is created, the Developer will be able to write HLCs or the same name directly into the Panel configuration files. There are three steps involved (refer to **Figure 2**):

1. Create a declaration in the `CONTROLLER` declaration for the function that will dispatch a High-Level Command.
2. Write the function, accessing the `MODEL` or `VIEW` as necessary.
3. Add the newly written function to the Function Factory.

```
//declaration in CONTROLLER header file
void AddPicture1i(const COMMAND& parameters);

//implementation
void CONTROLLER::AddPicture1i(const COMMAND& parameters)
{
    //First, make sure that our vector contains the correct number
    of parameters
    if ( (parameters.size()-1) != 1)   //expecting 1 parameter (the
     size-1 is because we're ignoring the first entry)
         throw EXCEPTION("CONTROLLER::AddPicture: Incorrect number
                                of parameters received");

    //Second, extract the parameter(s)
    int numOfPicturesToAdd = UGENE::StringToInt( parameters[1] );

    //Third, perform the action
    p_model->AddPictures(numOfPicturesToAdd);
}

//imlementation
void CONTROLLER::InitializeFunctionList()
{
    m_functionList["Setup0"] = CONTROLLER::Setup0;
    m_functionList["AddPicture1i"] = CONTROLLER::AddPicture1i;
    m_functionList["RemovePicture1i"] = CONTROLLER::RemovePicture1i;
    m_functionList["LoadNewPanel1s"] = CONTROLLER::LoadNewPanel1s;
}
```

Figure 2: Sample CONTROLLER API

3.2.2 Create a MODEL API

Once a new function has been created, Widgets inside the Panel configuration file will be able to subscribe to the `MODEL` in order to obtain dynamic information. There are three steps involved:

1. Create a **const** declaration in the `MODEL` declaration for the function that will return the subscription.
2. Write the function, accessing only const `MODEL` functions.
3. Add the newly written function to the Function Factory.

Refer to **Figure 3**:

```
//declaration in MODEL header file
std::string GetNumberOfPicturesMessage0(const SUBSCRIPTION& parameters) const;

//implementation
std::string MODEL::GetNumberOfPicturesMessage0(const SUBSCRIPTION&
parameters) const
{
    return ( UGENE::IntToString(GetNumberOfPictures()) );
}

void MODEL::InitializeFunctionList()
{
    m_functionList["GetRandomInt"] = MODEL::GetRandomInt;
    m_functionList["GetRandomFloat"] = MODEL::GetRandomFloat;
    m_functionList["GetRandomFloat2i"] = MODEL::GetRandomFloat2i;
    m_functionList["GetNumberOfPicturesMessage0"] =
                          MODEL::GetNumberOfPicturesMessage0;
}
```

Figure 3: Sample MODEL API

3.3 Construct Panels

Now here is the most interesting part of the whole process. Once the appropriate API has been created inside the CONTROLLER and MODEL, then it can be used from inside the Panel configuration files. These files are loaded at runtime, and the best part is that UGENE's components don't even know or care which API functions get called from within the Panel files.

Each Panel configuration file represents one screen's worth of information. Loading a new screen means loading a new Panel file. The Developer is expected to configure the Widgets that are to be displayed, as well as any other desired characteristics of the screen. At the time of this writing the only functional part of the Panel files is widget loading and initialization scripts, thus we will concentrate on these.

3.3.1 The Guts of the Panel Configuration File

The discussion to follow is probably already out of date when you're reading this document, since Widgets and Panel configurations are a part of UGENE that is currently in great flux. However, this will give you a good idea of what is going on.

It is best to start with a sample Panel configuration file (**Figure 4**), and then describe the example.

```
UGENE_PANEL_FILE

START_INIT_SCRIPT
     Setup0
END

START_WIDGET_LIST
     widgetName = PlusButton
     widgetType = BUTTON
     width = 10.0f
     height = 10.0f
     depth = 0.0f
     x = >GetButtonX
     y = >GetButtonY
     z = >GetButtonZ
     enabledImage = Images\PlusEnabled.gif
     disabledImage = Images\PlusEnabled.gif
     mouseOverImage = Images\PlusMouseover.gif
     pressedImage   = Images\PlusPressed.gif
     releasedMessage = AddPicture1i 1
     releasedShiftMessage = AddPicture1i 10
     releasedControlMessage = AddPicture1i 100
     ;
     widgetName = GoToOtherScreen
     widgetType = BUTTON
     width = 20.0f
     height = 20.0f
     depth = 0.0f
     x = 80.0f
     y = 80.0f
     z = 0.0f
     enabledImage  =  Images\nextPanel.png
     disabledImage =  Images\nextPanel_noAccess.png
     mouseOverImage = Images\nextPanel_checked.png
     pressedImage  =  Images\nextPanel_puke.png
     releasedMessage = LoadNewPanel1s PanelDefault.txt
     ;
END
```

Figure 4: Sample Panel Configuration File

There are a lot of points that need to be covered regarding Panel configuration files:

- White space, including tabs and spaces, is ignored.
- The carriage return is used to separate unrelated statements, but excess carriage returns are ignored.
- Comments will be allowed, but have not been implemented yet.
- Panel files begin with the statement `UGENE_PANEL_FILE`. This is the Panel file's "magic number," that identifies the file type.
- In the future the files will also include a version number, to help manage UGENE's growth.
- Panel files are composed of a series of sections, which can be written in any arbitrary order, but that must not overlap. Some sections may be dependant on the existence of other sections (*e.g.* when defining Widget groups).
- Sections begin with a "start" tag, and finish with the `END` tag.
- Within sections, sub-sections are separated by a single semicolon alone on a line.
- The `START_INIT_SCRIPT` section contains a series of HLCs to run, one after the other, each placed on a new line.
- The `START_WIDGET_LIST` section contains a series of Widgets, separated by semicolons.
- Note how Widget attributes are assigned with an '=' character. White space on either side of the symbol is ignored. Entries are written one per line.
- Notice the Observer-Subscriber pattern at work in the `PlusButton` Widget, in the function calls such as `>GetButtonZ`. The > before the function call allows the parser to recognize the difference between a function call and a string.
- Notice the Command pattern at work in the various `releasedMessage` attributes.

3.3.2 Apology Regarding Widget Specifications

In the future, we will be able to specify the details of what widgets are available, as well as what attributes they have. Right now, however, anything we would write could change tomorrow; hence we don't feel that it would be productive at this time.

We hope that **Figure 4** gives you a good idea of how UGENE is meant to operate.