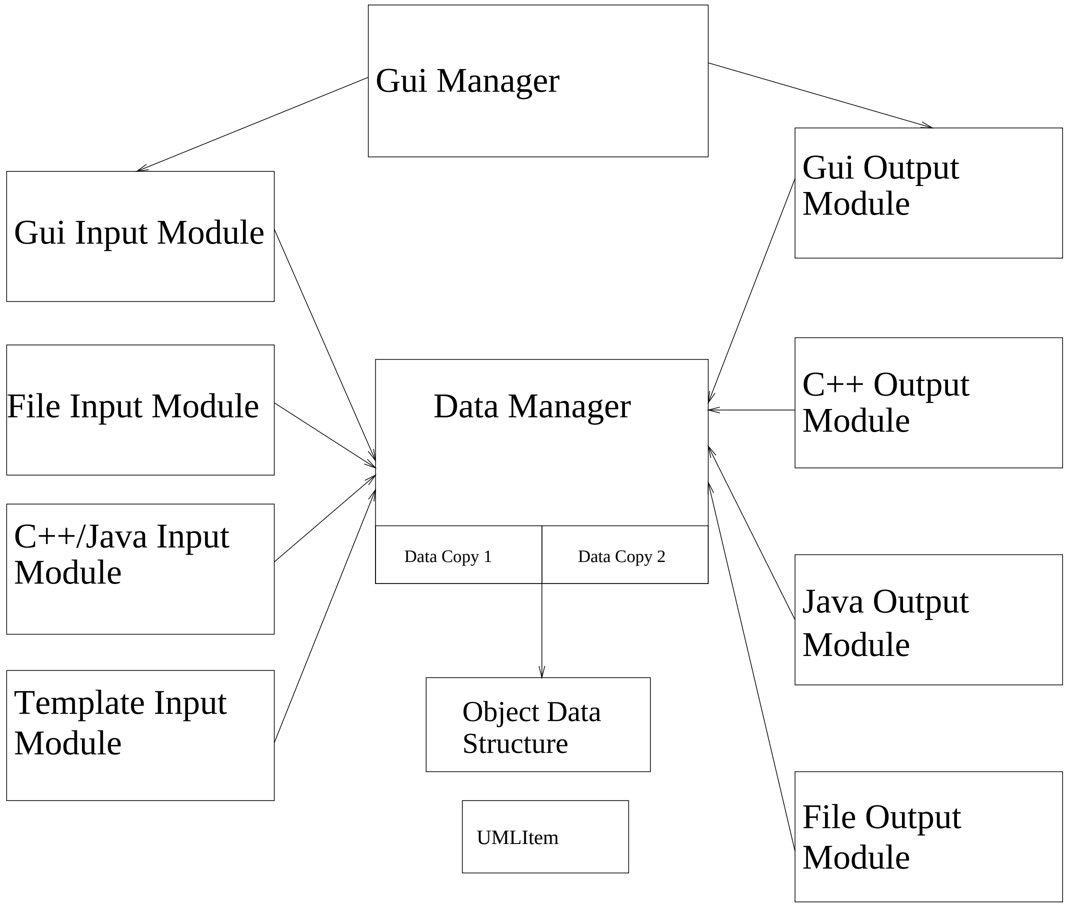
Hello UML Top Level Design

Christopher Filippis (cjf)

February 25 2000

1 Top Level Diagram

The top level view of this design for the Hello UML specification can be divided into three logical parts. The first part is the input phase, here data is collected from a number of possible sources and converted into calls upon the next part of the program, the data storage module. The storage module will contain the graph with its interdependencies, and all UML objects and their component data. The final phase is output, whether it be back to screen for further modification, out to a file for saving, or conversion to C++ or Java code.



2 Design Specifics

2.1 Gui Manager

This module will act as the coordinator for the Gui Input and Output Modules (see below), actually controlling the windows of the program, and interfacing into the windowing API used (i.e. Qt). It is provided to remove the interdependency that would have resulted if the Gui Input and Output Modules needed to speak to each other to make sure that the input received by one was consistent with the output generated by the other. This top level object can also control the operation of the other input and output modules, but the links are not shown here because they are trivial compared to the operation of the gui modules, and because they would clutter the graph.

2.2 Gui Input Module

This module will accept the input from all gui components and translate their information into calls to modify the main data structure. It will probably be responsible for most of the modifications of the main data structure in real time, so this class should be fairly efficient at data handling. But, it should not do any data processing, nor should it understand the relationships between UML objects, that is the job of the Data Manager.

2.3 File Input Module

The file input module will be responsible for handling the HelloUML file format. It will parse the file (checking for errors) and convert it into calls sent to the Data Manager which will then construct the internal representation. This is the file format that will be output through the File Output Module used for saving documents that the user is working on. The file format can be made with plain text encoding to keep it simple and human readable, I suggest using a simplified XML syntax.

2.4 C++/Java Input Module

This module is much like the File Input Module in that it also takes a file in and parses its contents. The difference is that the files that this module is to support are actual program files. It's logic is a little more complicated as does not have knowledge of the overall structure and dependencies, but must infer them from the way the classes are written. This will be a much more complicated parser, as the C++ language is far from simple. Perhaps, if only .H files, and their most common constructs (i.e. no complex compiler macros, etc) are considered, this would be more doable.

2.5 Template Input Module

This is a slightly different form of Input module than the previous ones in that it will send a full base description of the object structure to the Data Manager, but not through direct file input or manipulation of the GUI objects. This will ask the user a series of questions, some of which can have multiple answers, that would build up a tree internally which it then forwards to the Data Manager. For example, it should ask for object definitions, and allow the user to specify as many as are necessary. Then it should ask what kinds of procedures the user wants, etc. Special Templates are used for different types of project designs, such as client server, or generalized plug in structures. This is all presented through a series of GUI prompts, but independent of the main gui system.

2.6 Data Manager

This is where the central coordinator for information resides. The Data Manager contains the node graph which represents all objects and their connections. It will contain the objects in a list, so that more can be added easily, and objects can be taken away easily. The graph format comes in the form of the interdependencies between each of the UMLItems. These interdependencies should be represented by unique ids contained in the objects, hidden away from the user. These ids should be kept separate from the actual pointers to objects to allow easy output/input to and from files. A map should also be contained with a mapping of ids to objects. In actuality, there will be two node lists and two node maps, and two copies of each object. The reason is to contain a shadow copy of all objects to permit easy undoing of the last change to the node structure. Changes are made to one copy, and that copy is used as the main copy for output purposes. Then, when another change comes in, the previous change is also put on the second copy, to bring it up to speed with the current. If any undo is requested, the Data Manager simply swaps the working and the shadow copies.

Information is put into the Data Manager via a push philosophy, where the input modules push the information into the Data Manager for storage. The Data Manager is not aware or dependent upon any particular input system, as its API will be its window to the world, so to speak. Second, there will be methods to grab data. A particularly important design decision was to change from a push system for sending data from the Data Manager to the Output Modules to a grab system where Output Modules will have to pro-actively grab data from the Data Manager, rather than rely on the Data Manager to give them data. This avoids a cyclic dependency that would have been incurred in cases where the Input and Output mechanisms must be interrelated, as in the case of the GUI Manager. This provides a three level acyclic system design, with the top level being GUI Management and User Interfaces, followed by i/o processing in the Modules, and ending with the data storage mechanisms in the Data Manager. Thus, the Data Manager is the most important part of the project because it is the foundation for all other pieces in the project.

2.6.1 UMLItem

The UMLItem is the actual object that is contained in all of these data structures. It will contain a list of an objects attributes, such as its name, id, public or private status, its superclass, etc. It will also contain a list of dependencies that is based upon the ids used for each of the other UMLItem objects. Next, it will have a list of functions and their parameters and parameter types. Finally, it should have member variables, all of which can have an arbitrary set of attributes on them, independent of the system to which they will be implemented (Java, C++, etc). Each time a modification is made to a UMLItem, the procedure should return a code to indicate success or failure. This is how rules can be implemented. Thus, whenever the Data Manager tries to include an instance of a subclass into the superclass, a check will be made by the UMLItem being modified, and if that check fails, the method fails.

2.7 GUI Output Module

The GUI Output Module is responsible for displaying the contents of the Data Manager's working copy to screen. It will create the windows and boxes that will make up the visual class diagram, while allowing for interactivity. But all interactive user input is referred to the GUI Input Module, not the output module.

2.8 C++/Java Output Module

This module is the complement to the C++/Java Input Module, except that it is much easier to go forward than in reverse. When a translation to C++ code is requested, this output module will read in the entire node tree from the Data Manager, processing each item for the appropriate directives. With this information, it will then generate a series of .C and .H files, or .java files that have class stubs for the classes represented in the internal data structure.

2.9 File Output Module

This module is much like the C++/Java Output Module with the exception that it generates a single file, and that it writes the file out as its own proprietary format. As mentioned above, this is the format to be read in through the File Input Module, and should probably be a human readable format such as XML.

3 External Dependencies

There are basically two external dependencies incurred in this program. The first and most important is the reliance on a gui subsystem, such as Motif, X, Gtk, or Qt. While it does not matter which one is used as far as functionality goes, it would probably be more convenient to use a high level system such as Gtk or Qt for this project to ease GUI code development.

The second dependency is in the format for the C++ code. While Java has a standard implementation, C++ has many different variations that might fuddle certain compilers. Care should be taken to keep all HelloUML generated code as close to the base specs as possible to avoid any incompatibilities in different C++ compilers.

4 Task Breakdown

The task breakdown should probably be made slightly differently than the system breakdown to exploit symmetry in the module design, as most modules have both an input and an output class, and implementation should be exactly the same on both ends. The first task should be to create the Data Manager backbone to the project. This should be assigned to one person, or a group of two people who will be responsible for generating all internal data structures, and writing the implementation of all the API functionality that the other groups will have to use. A minimal set of functionality will need to be created first in order to facilitate testing of all the other pieces.

The second task would be to create the GUI system, so again one or two people should be left with the task of writing the GUI input and output modules as well as the graphical system that will actually display everything on screen.

The third task is to create the File Parser/Writer for the internal file format. This is relatively simple, and can probably be handled by only one person. That person should design the final file format spec, and write the appropriate code to create it. As the file format is independent of the actual data model, the creation of the file format can occur separate from everything else, once the API of the Data Manager is settled upon.

The fourth task is the header file generation and parsing routines. Since reading and writing are related by the common languages, one person should do each language, from reading (where most of the work will probably need to be done) to writing of the header files.

The fifth task is to create the Template Library and Wizard Help system. This again may require one or two people to accomplish, perhaps one to create the gui elements, and one to work out the template logic.

5 Group Organization

My recommendations on group organization are mostly based upon the descriptions given on the course web page. As such, they may be far from ideal, and may misrepresent people's interests unintentionally. But, here are the recommendations none the less...

For the Data Manager team, I would assign Di Wang and myself. I would have an advantage in working in the core of the project since I came up with this design proposal and understand the paradigm that it works under. Di Wang would be a great help in this because he has reportedly worked with large complicated systems before, and this being perhaps the most complicated and crucial component to the system, it would need someone who would also like to take a leadership position in order to work out all of the module APIs.

For the GUI team, I believe that Freddie O' Connell assisted by Greg Getchell would be appropriate. Freddie would be appropriate because of his prior work in graphics systems (cs123) but more importantly for his desire to work with others to produce a product. As the GUI presents our main form of interaction with the users, it would be necessary for someone working on it to be motivated to speak and work with the potential users, as well as the other programmers in designing a GUI that meets everyone's needs. Greg would also be good for this as he has demonstrated interest in working on a graphic/GUI intensive system in his own project ideas.

The file system is a very basic, yet very integral part of the system. Without a way to save and load designs, the system is quite useless for project based designs. But, any experienced programmer should be able to program such a system, so this position assignment is completely arbitrary. Thus, arbitrarily, George Quievryn to fill this role with partial assistance from Toan Phan.

Toan Phan will also be assisting in the other file management set, the C++/Java interfaces. As both of these systems are file format management systems, such a bridging assignment is possible, I believe. In charge of this section, though, would be Joshua Tasman as his work and knowledge from taking the compiler course here would prove infinitely valuable in these modules.

Finally, in the Template/Help system category, I would leave Michelle Neuringer in charge. As she originally created the specification to make a program capable of easing the troubles in using a complex program such as Rational Rose, she would be very appropriate to head the section of the program that provides the easy user interface for beginners. Her interest in visual systems and UI is most helpful here as well, and her work in artistic areas would help provide a sense of aesthetic in a system that should be as pleasurable to use as can possibly be under the circumstances (hey, nobody said design was easy).

6 Schedule

- 3/10 - Finalization of Group Organization, Design Review
- 3/17 - Finalization of Data Manager API for push/pull operations
- 4/7 - Initial Data Manager API should be implemented (at least with stubs in non-essential features) so that integrated testing can begin. Other pieces should have preliminary work finished, ready for some initial testing with the Data Subsystem
- 4/14- Data Manager code should be complete, low level testing and debugging of Data Manager should begin to ensure API performance to specs, initial usable GUI should also be finished with partial functionality
- 4/21 - All Modules should have basic functionality implemented, test integrations should now be possible (level 2 testing begins)
- 4/28 - Feature Freeze - All basic coding for all modules should be completed, no new features not in the original spec should be added. GUI should be functional enough to begin high level testing, and testing of other modules.Full integration of module packages should now be possible (In Class Demos) - Top level testing begins

- 5/3 - Final testing of integration completed, ready for user acceptance testing. (Public Demos)
- 5/5 - Evaluation of UAT, wrap-up - Begin work on documentation by this time.
- 5/12 - Finish Documentation, minor UAT related improvements included.

7 Assumptions/Changes from Specifications

This was a major rewrite of the design as seen in the Specifications, but containing the same functionality. The idea was to remove the cyclic dependencies seen in page 5 of the specifications using a modular approach to I/O issues, and a consolidation of data into simpler classes. The template library was moved to a higher level, as I believe its nature is much more complex than the raw data level to which it was assigned. The Logic functions were split up into two different pieces and placed into the functionality of other elements. Visual and UI logic such as understanding dependencies in a visual graph was put into the GUI Manager. Low level logic dependent upon object definition relations was placed at the low data level.

The major assumption was that an extensible and general purpose attribute system was desired for the UML objects, as no specific list of desired attributes was found, thus the system is based upon a key - value engine which will allow different languages to store different properties as necessary.