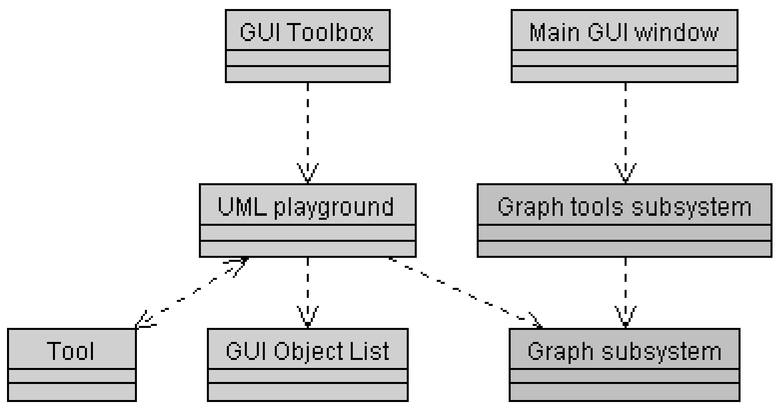
helloUML Top-Level Design

Lisa Cozzens (ljc)

February 25, 2000

1. High-level component diagrams

1.1. GUI subsystem



Main GUI window: This window will be located at the top of the screen and will consist of simply a title bar, a menu bar, and a toolbar. The menu bar will provide access to tools such as open/save file, the Layout Wizard and Design Wizard, the code generator, and online help. The toolbar will have buttons for the most frequently used tools. This is primarily a GUI component; it delegates actual work to other components.

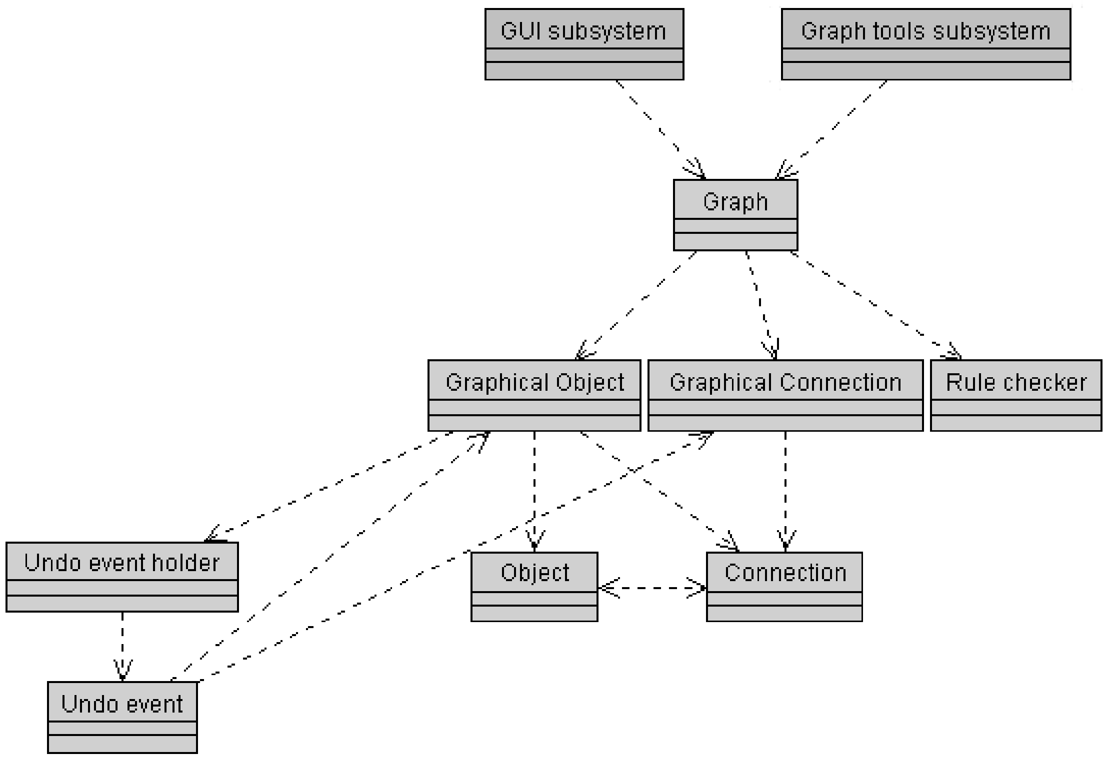
GUI toolbox: This is a small toolbox providing buttons for the connection tools (containment, inheritance, association, implementation), text box tool, lasso tool, etc. When the user clicks on a button, the appropriate tool is sent to the UML playground.

UML playground: This is where the user can draw objects and make connections between them. Double clicking on the playground creates a new object in the graph. Single clicking sends the coordinates of the click to the current tool, which will then call the appropriate methods on the UML playground. For example, if the user single clicks when the containment tool is selected, the containment tool will tell the UML playground to draw a containment relationship between the objects clicked on.

Tool: This is the superclass of a bunch of tool classes. For example, there's a ContainmentTool class, an InheritanceTool class, etc. Each subclass has a useTool(.) method that takes the coordinates of a mouse click as parameters and does whatever work needs to be done to apply that tool to those mouse clicks. There is a circular dependence on the UML playground, because the UML playground needs to notify the tool of the mouse click coordinates and then the tool needs to tell the UML playground what to do. However, the tool classes are very simple, since the useTool(.) method just calls methods on other classes, particularly the UML playground. These other methods can be easily stubbed out for testing the tool classes. I feel that this design is better than the alternatives (for example, having all the tool functionality in the UML playground class, which is more of just a GUI class and shouldn't have to worry too much about what tool is currently selected).

GUI object list: A GUI component that shows all the objects in the system. When the user adds a new object by double clicking on the UML playground, the UML playground notifies the object list so it can be added there as well. The user can show and hide objects by clicking buttons in the object list.

1.2. Graph subsystem



Graph: The graph is a manager class for the graph components. When the UML playground discovers that the user wants to create a new object or connection, it hands off responsibility to the graph, which actually creates the new objects.

Rule checker: The graph hands off responsibility for checking whether a desired connection is valid to the rule checker. For example, the graph might pass in a class, a subclass of that class, and an inheritance connection to the rule checker. The rule checker would flag this as invalid, because a class cannot inherit from its subclass.

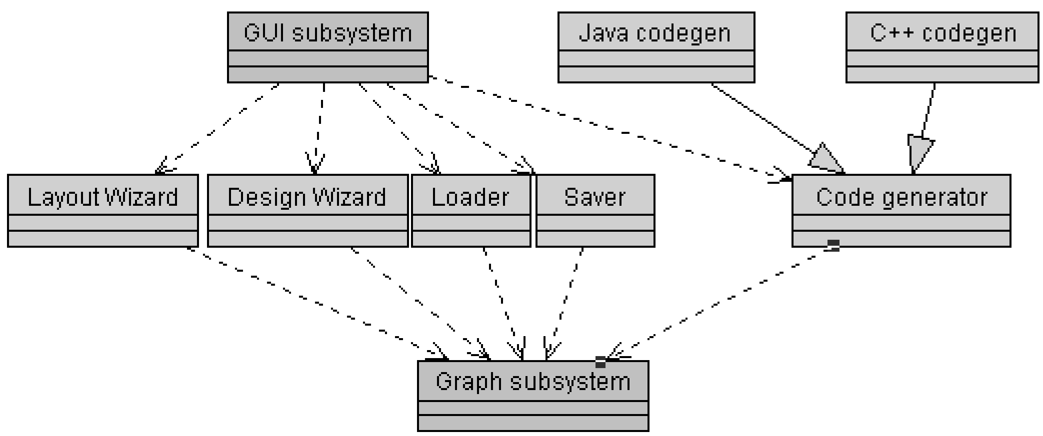
Graphical object: This component handles all the visual stuff connected to objects. For example, it actually draws the little boxes in the UML playground that correspond to the object and provides buttons and text fields to change properties of the object. It knows nothing about its connections, depending on its Object (see below) for that. It also hangs onto the various properties of the object (type of object, name of the object, methods associated with the object, etc.) It might be a good idea to have two classes in this component, one that is responsible solely for the GUI stuff and another that actually holds onto the object properties.

Graphical connection: Similar to the graphical object, this component handles all the visual stuff connected to connections. For example, it knows what type of connection it is (containment, inheritance, etc.) so it knows how it should represent itself visually (dotted line, inheritance triangle, etc.)

Object: The object just knows that it's part of a graph, so it's connected to other objects via connections. It maintains a list of all its connections. It knows nothing about that fluffy GUI stuff, or what type of object it corresponds to (superclass, interface, etc.) – none of that.

Connection: Each connection knows about the two Objects its connecting. This creates a circular relationship between Objects and Connections, but these two classes are small enough to be part of the same component. Like Objects, Connections know nothing about GUI stuff.

1.3. Graph tools subsystem



Layout Wizard, Design Wizard: Both of these components employ some AI magic that I'm not really clear on at this point. They need to know about the graph because they need to take into account all the objects and connections to do their magic. The Layout Wizard should lay out all the objects in some sort of visually appealing manner. For example, it might place all subclasses directly under their superclass. The Design Wizard helps the user by watching what they do and making suggestions about design, sort of like (dare I say it) the stupid paperclip in Microsoft Word (but much less annoying).

Loader, Saver: These components are responsible for loading (and parsing) files and saving files. The loader needs to know about the graph so it can add objects defined in the file to the graph. The saver needs to know about the graph so it can save the objects in the graph to the file.

Code generator: This component is responsible for generating code based on the objects and connections that the user has drawn. Consequently, it needs to know about the graph so it can get access to the objects and connections. There are subclasses of the code generator to handle different programming languages (Java, C++….)

Undo event holder: The undo event holder hangs onto undo events in chronological order. It provides methods to get the most recent event so it can be undone.

Undo event: The undo event knows about the state of the system prior to an event, so it can restore the system to that state. For example, suppose the user changes the name of an object. An undo event would be generated, and it would have a copy of the object before the name change. If the user wanted to undo that event, this copy would be moved back into the graph. This creates a cyclic dependency between Graphical Objects and undo events, but I don't see how it can be avoided. The undo event class should be small and simple enough so that this isn't too much of a problem.

2. External dependencies

This project does not have any dependencies on external programs.

3. Group organization

The group should be organized so as to fill the following roles:

- **Project leader:** responsible for general organization (e.g. Person X should work on Part Y), reviewing group members' work and suggesting improvements if necessary, keeping things on schedule and figuring out what to do if the schedule slips, resolving any problems that arise within the team, etc.
- **Design master:** responsible for the overall design of the program. This person must approve all proposed additions and changes to the list of required features, keeping in mind the "conceptual integrity" of the program. Although all group members can and should contribute to the design, this person has final say.
- **Lead programmer** and **co-pilot:** the person most responsible for the actual implementation of the program. How does this differ from the design master? First of all, the chief programmer will be responsible for more programming. Secondly, the design master is more concerned with high-level questions (e.g. will we need a database?) while the chief programmer can focus on more nitty-gritty questions (which database management software will we use?) The co-pilot helps the chief programmer make these decisions, along with the rest of the team, but the chief programmer's say is final.
- **Editor:** responsible for the program's documentation. As with design, all team members can and should contribute to the documentation, particularly for the parts of the program that they are directly involved with. The editor's job is to delegate documentation to different team members, to give feedback to team members on the documentation that they wrote, then to collate the documentation into a single coherent and consistent document.
- **Tester:** responsible for overseeing testing of the integrated system. Each team member should test his/her individual components, but the tester has primary

responsibility for finding bugs that arise after integration. The tester is also responsible for coordinating user testing.
- **User interface manager:** the equivalent of the design master for the user interface, this person is responsible for ensuring that the user interface is consistent and easy to use. S/he may set guidelines, e.g. say that every error message should be displayed in a small dialog box that includes the error number and a clear description of what the error means, along with an OK button.
- **Toolsmith:** responsible for learning about and setting up tools that may be of use to the programmers. For example, this person should select the configuration management software that the group will use (with input from the rest of the group), set it up, and help the other group members learn how to use it.

All team members will be involved in programming to some extent, but the chief programmer and co-pilot will take on a larger share of the programming than other team members.

4. Task breakdown

- **GUI subsystem:** Two people (one of whom should be the user interface manager) will be assigned to this section. One person will work on implementing the UML playground and the various tools, while the other will work on the actual GUI (e.g. this button should look like this and go in this place)
- **Graph tools subsystem:** Three people will be assigned to this section. One person will work on loading and saving. One person will work on code generation. One person will work on the Layout Wizard and/or Design Wizard.
- **Graph subsystem:** The remaining three people will work on this section. The lead programmer should be one of these people. One person will work on the rule checker. The other two people will work on the graph itself and undo functionality.

5. Schedule

February 25: Top-level design proposals
March 3: Discussion of top-level design proposals
March 10: Finalization of top-level design
March 15: Public interface proposals
March 20: Discussion of public interface proposals
March 24: Finalization of public interfaces
April 5: Detailed designs of components finalized, begin coding.
April 10: Initial integration, with much of the functionality still stubbed out
April 17: Integration, with much of the functionality and the entire user interface implemented
April 28: Full system integration, with all functionality implemented. No additional features will be added after this date.
April 28-May 4: testing and debugging.
May 4: Public demo. First draft of documentation.
May 4-12: any additional testing and debugging, finalization of documentation.
May 12: Final submission.