# CS190 FORCE Top-level Design Document

Christine Waggoner (cmw)

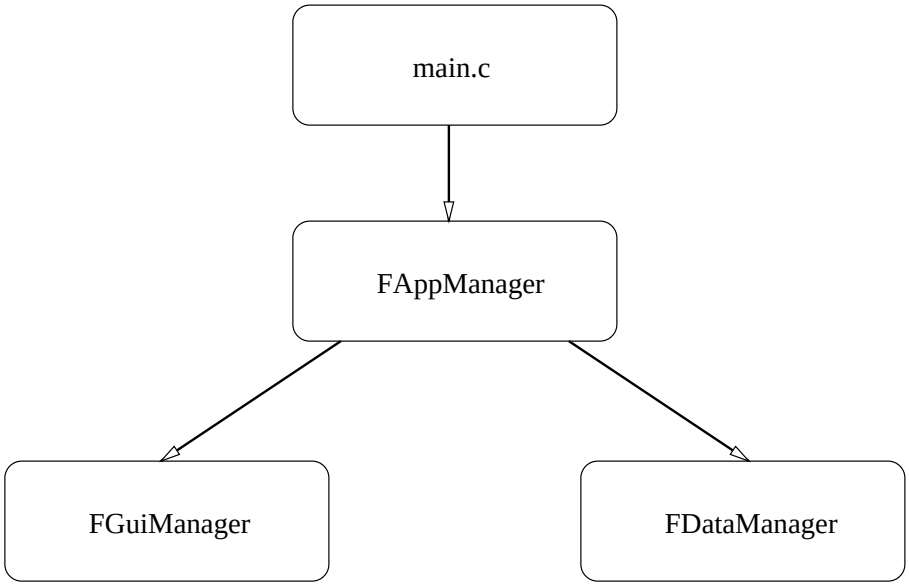
Feb. 26, 1999

## 1 Document Overview

### 1.1 Introduction

This document contains a top-level design for the FORCE, the Flow of Regulated C Expressions, a graphical programming environment for beginners in programming who are learning the C language. It should also be noted that this document attempts to encapsulate some design ideas from previous design proposals for the FORCE.

### 1.2 Top Level Components

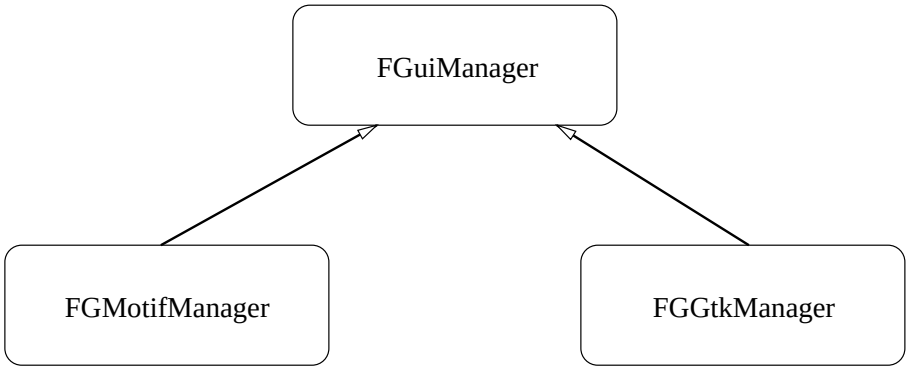



In order to devise a possible design for the FORCE, the design can first be divided into three primary components: the application manager, the GUI manager, i.e., the front end, and the data manager, i.e., the back end.

Specifically, the FORCE requires a CAD-like visual environment in which the user can construct small C programs by placing and manipulating visual programming widgets. The FORCE also requires a robust underlying data structure to accurately track the user's actions and to translate the user's visual programs to C code. The application manager serves as a controller over the FORCE application and a communicator between the front end and the back end. Of course, these three main components of the program must also be divided into smaller design components.

# 2 FORCE GUI Manager, FGuiManager

This class provides an interface to the visual components of the FORCE and encapsulates the front end of the FORCE. Ideally, this class will help reduce dependencies between the front end and the back end. For example, different implementations of the GUI could easily be added simply by by exploiting polymorphism to create multiple GUI managers which subclass from a parent GUI manager and use a common interface.




Additionally, one should note that when reading about the design of the GUI, program animation is not directly addressed. If program animation is included in the program, a graph visualizer such as the one described in Amir's top-level design document may be needed, as well as additional menu options in the UI's menubar.

# 3 User Interface

The UI of the FORCE should be written in Motif since it is integrated in the Brown CS curriculum and familiar to most of the people working on the FORCE. In general, the UI should support windows, menus, dialogs, buttons, drawing, and possibly program animation. If program animation is included, multi-threading may be required and will have to be accounted for in the program design. Another general design note regarding the UI is that in order to implement a useful interface for the FORCE, it will be necessary to use create wrappers around Motif widgets, their callbacks, and keyboard shortcut callbacks.

## 3.1 UI Components and Features

### 3.1.1 Menu Options

Listed below are possible menu items which would need to be supported. Each menu would appear on the application menubar and each menu item would require a callback and a keyboard shortcut.

- File: Open, New, Save, SaveAs, Print, Quit
- Edit (see Actions list below)
- Code: Run, Compile

- Windows (see Windows list below)

### 3.2 Actions

In order to support all these actions and undoing these actions, a command object would be required for each action. Each action should subclass from an abstract superclass.

- Cut
- Copy
- Paste
- Move
- Edit
- Delete
- Undo
- Clear
- Select
- SelectAll

### 3.3 Windows

Windows represent a signicant component of the UI. It would be most logical to create a minimum of a single class for each of the items listed below.

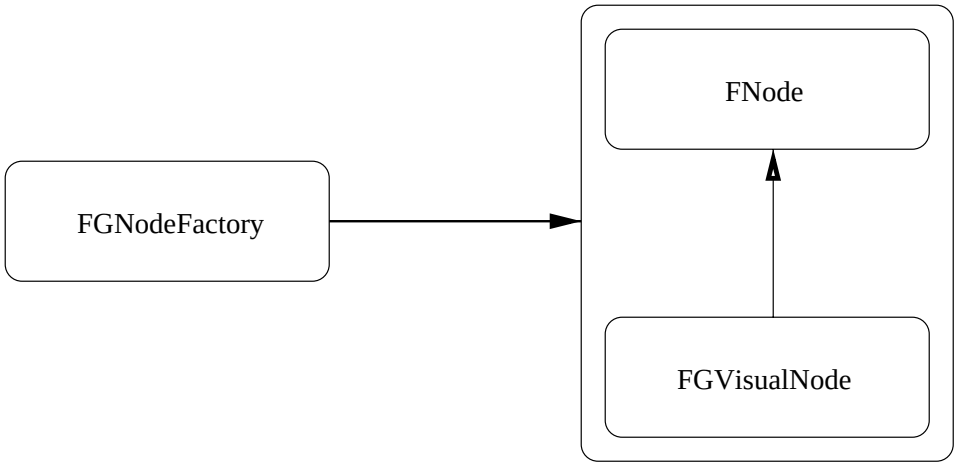
- ToolPalette, palette from which the user selects tools to manipulate the program graph
- New Graph Builder/Editor, area in which the user constructs program graphs
- GUI Builder, interface to build simple GUI's for FORCE programs
- Function Library, user-defined functions and supported library functions
- Varible List, list of variables employed in a program
- Code Builder, interface to input actual C functions
- List of graphs on which the user is working

### 3.4 The Visual Graph, FGVisualGraph

The visual graph is the primary work area/window in the user interface which contains the visual programming widgets of the user's program. The visually programming widgets will have to be built from custom code. In addition, FGVisualGraph will most likely be a subclass of a Motif drawing area.

The visual graph controls the drawing of the graph and provides an interface to visually manipulate the appearance of nodes in the graph. The visual graph also tracks mouse events within the work area and updates the the graph following editing actions.

### 3.5 Node Factory, FGNodeFactory




The node factory is reponsible for generating the visual programming widgets, termed "visual nodes" in this design document. The node factory's design is based on the factory design pattern, a creational design pattern which abstracts the instantiation process. (p. 81, Gamma, et al) Furthermore, by using the factory design pattern with polymorphism, it will be possible to easily add new visual nodes to the program.

### 3.6 Visual Nodes, FGVisualNodes

The node factory is responsible for generating the visual nodes. The visual nodes are visual components which represent functions, variables, data, operations, arrays, conditionals,and loops. These nodes appear in the visual graph. The visual nodes provide interfaces which enable them to be visually manipulated within the visual graph. All visual nodes will subclass from a single parent class, FNode. If program animation is supported, each visual node should also have an animate method which can be called when the program runs.

#### 3.6.1 Types of Visual Nodes

- Value node

- Variable node
- Function node
- Operator node
- Array node
- if/else and endif nodes
- while and endwhile nodes
- Comment node

### 3.7 GUI Builder, FGGuiBuilder

The GUI builder will be an application window which will enable the user to create extremely simple GUIs to be used with the programs that the user creates. It may be instructive to explore DTBuilder as a model for the design of the FORCE's GUI builder. As the user constructs a GUI in the GUI builder, the callback functions needed by the GUI should be added to the user's visual graph of the overall program.

### 3.8 Code Builder, FGCodeBuilder

The code builder is not a code generator. The UI will provide a "code-building" interface, i.e., the user will be able to input actual C code into the code builder. The code builder will provide an interface for the user to enter the code in an organized fashion and then a graph of the code will be created to be used as the user wishes. This seems like an extremely difficult component to design since the FORCE has an extremely limited feature set that it supports.

### 3.9 GUI Nodes, FGGuiNode

GUI nodes are somehwhat related to visual nodes; however, GUI nodes are specifically used for event-driven programming and represent GUI components in the user's final program. For example, GUI nodes provide interfaces to widgets such as push buttons, menus, and radio buttons. GUI nodes contrast visual nodes in that they provide interfaces for events, rather than representing a single programming construct. When the user wishes to create a GUI, a window in which the user can construct a GUI will be required. In addition, the insertion of new event-related functions into the visual graph should be provided.

### 3.10 States, FGStates

The inclusion of components such as the tool palette in the user interface seems to make the UI into a modal user interface. In order to deal with the modes of the program, it seems logical to use state objects (p. 305, Gamma, et al) to simplify the complexity which may arise in the program without the use of state objects.

For example, it might be useful to define a state for each tool. For example, the GUI's paint method might simply tell the current state object to draw which would in turn display the appropriate menu options. This state design is extremely extensible in that many, many states could be created.

### 3.11 UndoManager, FGUndoManager

In modern software, atleast one level of "undo" is absolutely fundamental in a program. In order to support undo-ing a command, it will be necessary to use the command objects (p. 233, Gamma, et al).

For example, a cut command object would store the visual node that was cut from the visual graph. The cut command's undo method would simply re-insert the visual node into the visual graph. While there is indeed a cost to supporting undo, it is important to design for this ability even if "undo" capabilities must be eliminated later in the development process.

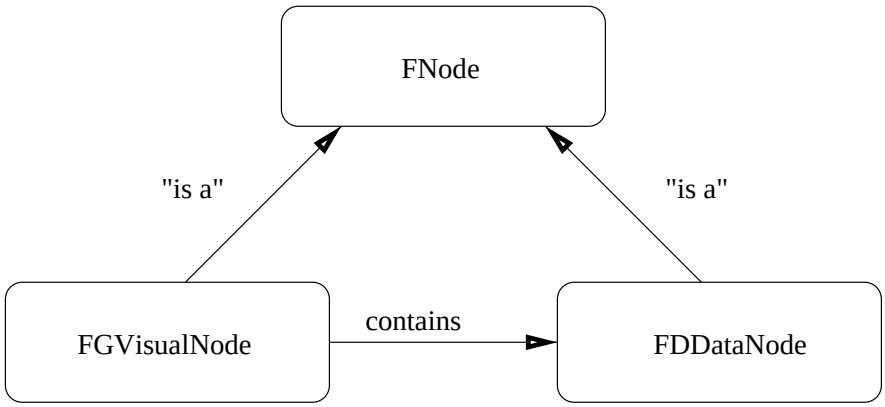
## 4 FORCE Data Manager, FDataManager

This class provides an interface to the data manipulation components of the FORCE. This class simplifies the design of the program by requiring that the other components of the program engage in "contact" with only a single object for data manipulation routines. The added benefit of using a data manager object is clear when one imagines the potential confusion of passing pointers to all the data manipulation objects among the front end's classes.

### 4.1 Internal Data Graph, FDDataGraph

The internal graph tracks the nodes of the program by ordering the input from the visual nodes in a fashion which will allow the code generator to output syntactically and semantically correct C code for the user's program. The internal graph will also be used to verify that the user is creating a reasonable program.

### 4.2 Data/Semantic Nodes, FDDataNode



The data nodes will be instantiated within the visual nodes and inserted into the internal and visual graph so long as the user is carrying out a semantically correct action.

### 4.3 File Parser, FDParser

The file parser generates files in the FORCE's file format and reads files in the FORCE's file format. It may also be necessary to have a routine which generates postscript so that the user can print the visual representation of her project. As stated in the specification document, it will be important for the file format to keep track of the locations of the visual nodes.

### 4.4 Code Generator, FDCodeGenerator

This class will be responsible for reading an internal graph and generating C code for the user. The code generator should also generate Makefiles for the user. The code and the Makefiles that are generated should be extremely easy to read and to understand. For example, the C code should be indented and employ traditional coding conventions. As stated by Jeff in his design document, it may be desirable to make several subclasses of a a parent CodeGenerator and have the subclasses handle output for multiple langauges. In addition, As stated by Michael Pellauer in his design document, it may be useful to devise an algorithm to avoid storing repeated functions in memory.

### 4.5 Function Library, FDFuncLib

The function library will provide an interface to a database of functions. The function library will include user-defined functions as well as the Standard C library functions. The function library should contain fields for the true function's name, any simplification of the name, i.e., a keyword, and explanatory information regarding the function. The function library should also be organized in such a way that the user and implementors of the FORCE can categorize and recategorize functions as needed.

As stated in Jeff Alexander's top-level design, multiple edits to the function library should be avoided. Therefore, the function library should control reading and writing to the library. In addition, as proffered by Jeff, it may be desirable to further abstract the function library by including a function library interface class.

## 5 FORCE Application Manager, FAppManager

The application manager provides an interface for the main components of the FORCE to communicate. The application manager will also instantiate the major components of the program and track the number of windows open while the application is open. Specifically, the application manager will instantiate the gui manager and the data manager. The gui manager and the data manager will respectively manage their own subcomponents.

### 5.1 Help Server, FHelpServer

The help server is a quasi-top-level component because it provides the help system of the entire program. The help server will not only generate error messages, but will also generate context-sensitive help. The help server will include error messages, tool tips, and pop-up messages. The help server should also provide the ability to read and write to its database. For example, it should be relatively easy to edit a help message. Jeff suggests a possible design for the help server based on a lookup table to help messages and their respective ids. In addition, as suggested by Jeff, it may be beneficial to provide a HelpInterface class to be used in the bulk of the program to avoid confusion and overhead. Finally, note that the help server should never be mistaken for a replacement for program documentation.

## 6 Personal Notes

- I think that the Code Builder could be exceeding troublesome and is not essential to the project.
- I am against the name "CLUIBuilder" and do not believe that a separate class in needed for the CLUIBuilder. I think that all programs should be built in a single layout environment and that the GUI should be built in another environment.
- Will the program insert any code other than the user's? How do we plan on hiding UI code? Are we planning on using UML?
- Will a shell pop-up when the user runs a commandline program?
- Is there really no feedback window in the UI?
- How will printing work?
- The spec states something about adding a "make clean" option to one of the menus. Is it really reasonable to expect a user to understand the concept of "make clean" ?
- Shouldn't there be some object to indicate scope visually to the user? Perhaps, color or partitioning the screen into areas would be helpful.