N2EMU

Software Design Document

Team 09
Alex Chmelka
Bryan Borer
Dennis Feng
Dominic Hezel
Tyler Zinsmaster

Introduction

Purpose

The purpose of the document is to explain and summarize the intricacies of our emulator and give a deep understanding of how each part of the application interacts with itself and the user.

Design Goals

- Availability: N2EMU should be useable on-demand.
- Readability: The N2EMU source code will be adequately documented so it can be more easily understood.
- Usability: By providing straightforward interaction with the user, the system should be intuitive.
- Robustness: User input outside of specified parameters should be ignore

Design Trade-Offs

This project is designed more for ease of programming and ease of use than for performance, as the system being emulated is rather old in its design, and the level of accuracy required is relatively low, performance should be sufficient on just about any relevant modern system. The largest usages of memory will be in storing the code from the text files, and the usage of an array for the emulated system memory. WIth this in mind, we are not making any active choices to store re-used values in memory for the sake of faster response time.

Another tradeoff made is that the system has been designed to interpret instructions from assembly text rather than machine code, so as to prevent the need of a full assembler subsystem. This leads to the necessity of conversion between hexadecimal and decimal at multiple points in the system runtime, so that the user always sees values in hexadecimal, while the system runs on decimal.

A design tradeoff was made early on in regards to the complexity of the processor class, which was split multiple times, arising in the instruction interpreter and behavioral core subsystems. This allows for better overall organization and implementation, but also creates more complex diagrams.

Interface Documentation Guidelines/ Coding Conventions

All interfaces should have a comment describing their general purpose and usage. Individual implementations of each interface should have descriptions on their specific usage.

All functions that are not self-explanatory should have an explanation of their purpose as a comment.

All functions, variables, and classes should be named using upperCamelCase conventions.

Variables used in multiple classes should have the other classes they are used in specified, as well as how other classes use them in regards to the current class.

Classes should be unambiguous and follow the same layout conventions. Inheritance on the top, main function next, other functions below.

Counter variables should follow standard i,j,k convention. They should be used locally instead of globally.

Avoid global variables when possible.

Errors should be caught and return exceptions.

In regards to Data Persistence, input file lines are stored as an arraylist, which is passed on piece by piece unmodified to multiple systems. Commands sent through the command prompt are not saved beyond their effects. The registry and flag variables will be stored in the processor for the duration of the program runtime, until a new file is loaded or the program is RESET.

Definitions, Acronyms, and Abbreviations

NIOS: Netware Input/Output System. A brand of processors made by Intel, whose assembly language and behavior this project will interpret and emulate the second generation of.

Emulator/Emulation: For the context of this project an emulator can be defined as a program or system which wishes to recreate (emulate) the behavior of another system. In this case, the project will be emulating the behavior of the NIOS II processor.

GUI: Graphical User Interface. What the user sees when interacting with the program.

Exception: A kind of message that shows that some error was caught.

Interface: In UML, a kind of component which allows for usage of component functionalities for other classes.

References

CSCE 361 Increment Instruction sheets

Intel NIOS II manual:
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

JNIOSEMU (Existing Java Based NIOS II Emulator): https://stpe.github.io/jniosemu/

Overview

Software Design Document overview:

Section 1: Introduction;

1.1 Purpose

1.2 Design Goals

1.3 Design Trade-Offs

1.4 Interface Documentation Guidelines/ Coding Conventions

1.5 Definitions, Acronyms, and Abbreviations

1.6 References

1.7 Overview

Section 2: Current Software Architecture;

Section 3: Proposed Software Architecture;

3.1 Overview

3.2 Subsystem Decomposition

3.3 Hardware/Software Mapping

3.4 Persistent Data Management

3.5 Access Control and Security

3.6 Global Software Control

3.7 Boundary Conditions

Section 4: Subsystem Services;

Section 5: Packages;

Section 6: Class Interfaces;

Section 7: Detailed Design;

Section 8: Glossary;

Current Software Architecture

Overview

Our program roughly matches the MVC architecture.

Our program revolves around three main systems, the Processor, GUI, and Behavioral systems. The processor does the meat of the computation, executing assembly instructions, storing registry/memory values, and having processor state control functionality. The Behavioral system largely exists to feed the processor instructions loaded from a file as well as relevant information from command line input. At its base, it gives the processor a line by line reinterpretation of assembly code to execute. However, it also takes command line inputs from the user, and accepts a program file as loaded in by the user. The GUI has two parts, Input and Output. The Input is made up of buttons and a command prompt. The buttons allow for the access of processor state control functionality by the user in GUI, for pausing, running, stepping, resetting, and for loading text files into the behavioral core. When the processor needs to be paused, resumed, or stepped, it takes input from the a button on the GUI. The command prompt allows for the input of commands which are passed to the Behavioral system. The Output allows the viewing values through a register box, memory viewing box, and miscellaneous box, which are updated by the processor as instructions are performed.

See general class descriptions below:

USER - The user will upload the assembly file to the main system, start and pause the emulator, and use the command line to set breakpoints and set points in memory to view.

BEHAVIORAL CORE- The behavioral core is the “brain” of the program, which determines how the different subsystems interact to scenarios. The behavioral core will take the assembly code and break it down to individual instructions. Will not work with incorrect syntax, but will accept any .txt file. The main component of the “model” portion of the MVC architecture.

INSTRUCTION INTERPRETER- The instruction Interpreter takes in instructions from the behavioral core, identifies the instruction, as well as input and output values.

PROCESSOR - The processor will run the identified instructions, updating the registry, memory values, and other relevant processor values, as long as it is in the “running” state. The main component of the “controller” portion of the MVC architecture.

THREADS - Threads are a class within the program which provide multithreading functionality. When the program is run, two main threads start, for the processor and for the GUI.

LOADFILE- This class exists purely to break an input text file into individual lines for use by the other classes.

GUI://

COMMAND PROMPT- The command prompt accepts user input from keyboard, and allows for setting up to 16 memory locations to view in the memory box viewing window, In addition, breakpoints can be set per line so that the system pauses when desired. Breakpoints can also be removed per line.

BUTTONS:

Load File: Brings up a file select prompt, accepts .txt files, and sends them to the behavioral core.

Run: The run button will run the program only once the processor is no longer in the NOT READY state (the file has been input). The run button also starts the program again after a break or pause-induced pause.

Pause: The pause button stops the running of the program.

Reset: The reset button returns the program counter to zero, resets registers and memory.

Step Up: The step-up button is used to increment the program counter to the next instruction, for debugging purposes.

Browse: This button opens a file navigation window, where you can manually select the file to load into the program.

Execute: This button executes command line commands.

Write .txt : Creates subfolder logs/ if it doesnt exist, writes relevant logging information to a .txt file, of which an unlimited number can be made. You can clear all of the log files with rmLogs command line command.

SWITCHES:

Can be toggled on and off while the program is running, so as to set the values of corresponding locations in memory.

BOX VIEWING WINDOWS:

Memory: This window will keep track of up to sixteen memory addresses and show the hexadecimal value contained at each memory address.

Registry: The registry box window lists each register and its corresponding hexadecimal value.

Text File: This window shows the code which has been loaded as a text file within the emulator.

Misc.: Includes viewing for the Program Counter, Return Address, Processor State, and Breakpoint Lines.

File Path:This box shows the path to the file we will load. Can be edited to specify path via keyboard input.

Message Box: Contains errors and Messages that will pop up during program execution, defined within the programs. Can be cleared via the command line cmes command.

7 SEGMENT HEX DISPLAYS:

These exist to display the values of the last 4 hex digits at the specified memory location 1000, in 4 individual 7 segment implementations.

Subsystem decomposition

The emulator can largely be broken down into Processor, GUI, and Behavioral systems.

The Processor System can be broken down into functions regarding instruction execution, registry/memory storage, and processor state control functionality.

The instruction execution functionality uses a general instruction format created by the Instruction Interpreter subsystem to determine individual instruction execution behavior for all instructions. This functionality can change the values of the registry, memory, and various processor state values.

The processor stores the registry and memory values, which are displayed using Box subsystems and which can be specified using the Command Prompt in regards to memory.

The processor features functionality and values regarding its state, such as program counter, processor status, return address, and current line. There are various functions to change the running status of the processor, which are implemented on a user control level using the input subsystems.

The GUI can be broken down into Inputs and Outputs, which are broken down into individual subsystems.

Inputs can be broken down into Button, Switch, and Command Prompt subsystems, and Button can be further decomposed into its individual implementations.

Button implementations can manipulate processor state and functionality, as well as loading an assembly file to be used.

Command Prompt subsystem takes in user commands and sends them to the Command prompt interpretation section of the Behavioral Core.

Outputs can be decomposed into the individual implementations of the Box subsystem, and the 4 7-Segment Hex displays.

All Box implementations display text which is used to represent various informations that are desired to be displayed.

The 4 7-Segment Hex displays take the value of a location in memory, and display the hex digits with 7 individual segments per display. an emulated I/O device.

Switches are emulated I/O devices that set the value of specific locations in memory based on whether they are on or off.

The switches and 7 segment displays make up the I/O subsystem.

The Behavioral system can be decomposed into the Behavioral Core and Instruction Interpreter subsystems.

The Instruction Interpreter subsystem is responsible for taking individual instructions and converting them to a format more easily used by the instruction execution subsystem of the Processor.

The Behavioral Core can be further decomposed into Command Prompt command interpretation, and breaking down the input assembly file into individual instructions and their lines.

Hardware/software mapping

In terms of outside hardware, our program will have no functionality. As merely a program that is meant to emulate an assembly file being run on a processor that supports NIOS II, there is no current need to push a program to an outside board(such as an ALTERA board).

Persistent data management

When a file is loaded, each line of code will be stored in an array list. This allows the instruction interpreter to take a line of code to interpret an instruction, and access another instruction quite easily. To add to this, when a box needs to select elements of a file, it displays it. When a command is entered by the user, it will be stored as a string for the behavioral core to interpret. Button inputs will trigger methods within the other classes of the software.

Access control and security

With little to worry about with users accessing areas of the program they shouldn't be, security implementations are largely unnecessary. A single user is the only person in this program. With access control, it is important to make sure the user doesn't attempt to go out of bounds or upload files with incorrect code. If the user attempts to view memory that is out of the range of the emulator, they will be denied and sent an error message. If a file with unusable code has been input, the program will not run.

Global software control

To ensure concurrency and synchronization, the classes for subsystems which share the same component variables will feature functionality to pass the values of those variables to other subsystems whenever they are changed.

Display boxes will update based on changes in the data that they are supposed to display, where the subsystem providing the data will send the new data to the display box, which will update automatically.

As ease of implementation was chosen over performance, button presses are detected using polling rather than interrupt calls.

We have decided that by updating the values of variables for all necessary subsystems when they are changed, this is the best approach to allow concurrency for the emulator.

When the processor is done executing an instruction, the program counter will increment, and the processor will send information to the behavioral core to send the next instruction and line information to the instruction

interpreter and processor, respectively. The processor will wait for the instruction interpreter to send the next instruction information before attempting to begin execution.

Boundary conditions

For startup the user will run the N2EMU program. This will bring up the GUI, with options for uploading, entering commands in the terminal, and other processor control buttons. For shutdown, the user can simply close the window and end the program. For error throws, if the program runs into a fatal error, it will display a window message and shutdown.

Proposed Software Architecture

No Changes

Subsystem Services

BEHAVIORAL CORE: The Behavioral Core parses text files loaded into their individual instructions, and handles the setting of memory locations for viewing as well as the setting of break points, parsed from the command prompt. The individual instructions are sent to the instruction interpreter for further action.

(Parsing the text file given by the user. Parsing the command line inputs given by the user)

BOX: The Box subsystem is responsible for displaying any text in the user interface. That includes displaying the assembly code imputed, the break points specified, the return address, the processor state, the PC, the register values, and the memory locations to be tracked.

(Displays the assembly code, break points, return address, processor state, the PC, the register values, and memory locations)

I/O: The I/O subsystem allows for the use of 4 switches, and 4 7 segment hex displays, based on and manipulating specified memory values.

BUTTON: The Button subsystem is responsible for creating all the buttons in the user interface. The reset, run, pause, step up and load file buttons are all created by button.

(Creates the reset, run, pause, step up, and load file buttons.)

COMMAND PROMPT: The Command Prompt subsystem is responsible for the command line part of the user interface. Inserting a breakpoint, removing a breakpoint and specifying a memory location to track are done through the command prompt by sending commands which are parsed by the Behavioral Core.

(Inserts breakpoints, removes breakpoints, specifies memory locations.)

INSTRUCTION INTERPRETER: The Instruction Interpreter takes the strings of assembly instruction from the behavioral core and determines what they all do. It then sends an identifying instruction number, as well as input and output locations to the processor so the instruction can be executed.

(Inputs assembly instructions and determines what they do. Relays information to the processor)

PROCESSOR: The Processor is the biggest part of the emulator. Instruction inputs are used in this subsystem and the instructions are executed. The subsystem then routes all of the outputs to the box subsystem to be outputted in the user interface. The processor also contains the functionality for the various processor control buttons.

(Executes the assembly instructions, pushes outputs from execution to the BOX, various processor control buttons)

Packages

In our current implementation, packages don't apply to any classes. With GUI taking buttons and the terminal as methods, a package doesn't really make sense to have applied.

Class Interfaces

Instruction Interpreter - This class calls the interpret method. This method is sent each line of the source code individually and tokenizes it. Based on the instruction, it defines how to divide each line up into inputs and outputs, formed from the registers. The compareFlag is set to a certain value depending on whether or not two registers are being compared, such as in a conditional branch instruction. This class also takes the instruction name as the first token, which will be passed to the processor along with the inputs, outputs and compareFlag. This class receives the symbol table made from the behavioral core which maps labels to exact values and assigns the value as an input or output.

Behavioral Core - The Behavioral core is the brain of the program. This class has three methods. The first method will take the text file uploaded by the user, and it will parse it. This text file is the assembly code that the user has written. The second method is called load instruction. This method will load the instruction line by line as parsed from the previous method. The third method is called execute command and this method will execute the commands given by the user in the GUI.

Processor - The processor class actually breaks down the line of code and what it does. It will take in the instruction and find the case for instruction. It will then do a multitude of actions depending on the instruction. This is done with the execute Instruction method. The processor also contains the run, pause, stepup, and reset methods, which will do as the method implies: puts the emulator in the respective states.

Loadfile - This class will load the file (This file is a txt file consisting of assembly code the user uploads) for the rest of the classes to use.

GUI - The GUI (Graphical User Interface) will be the face of the emulator. It uses Swing and AWT to provide an interactive environment for the user to interact with the Emulator. All user inputs will be done through the GUI.

Detailed Design

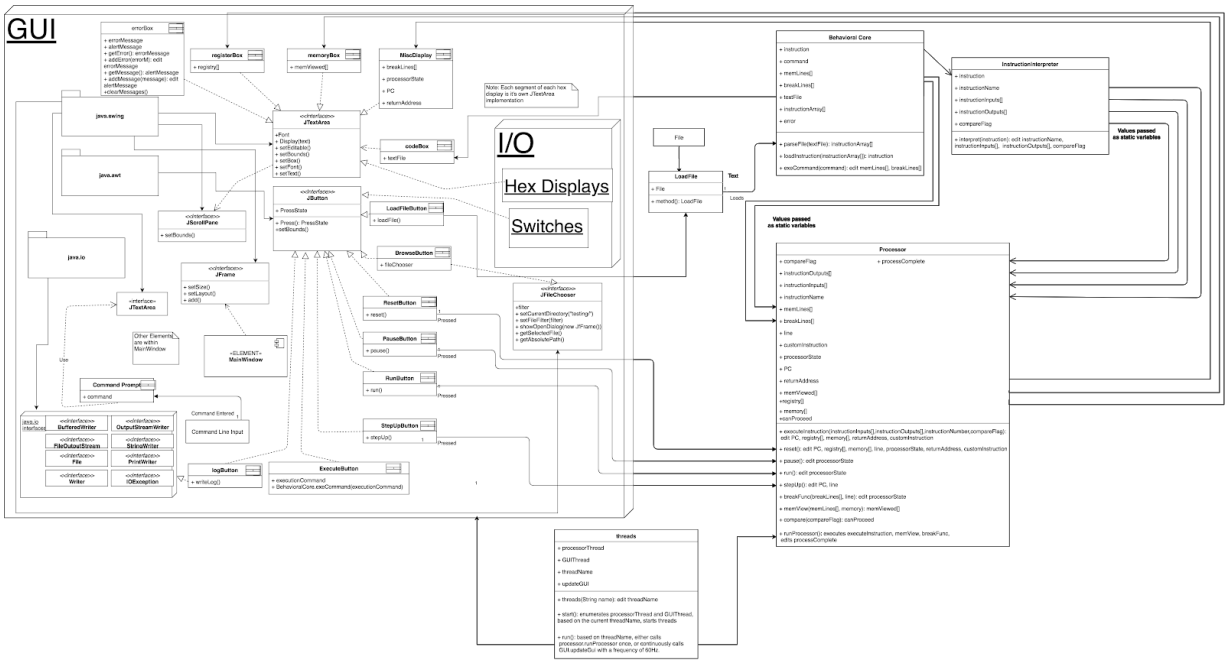

Figure 1: Total System Class Diagram

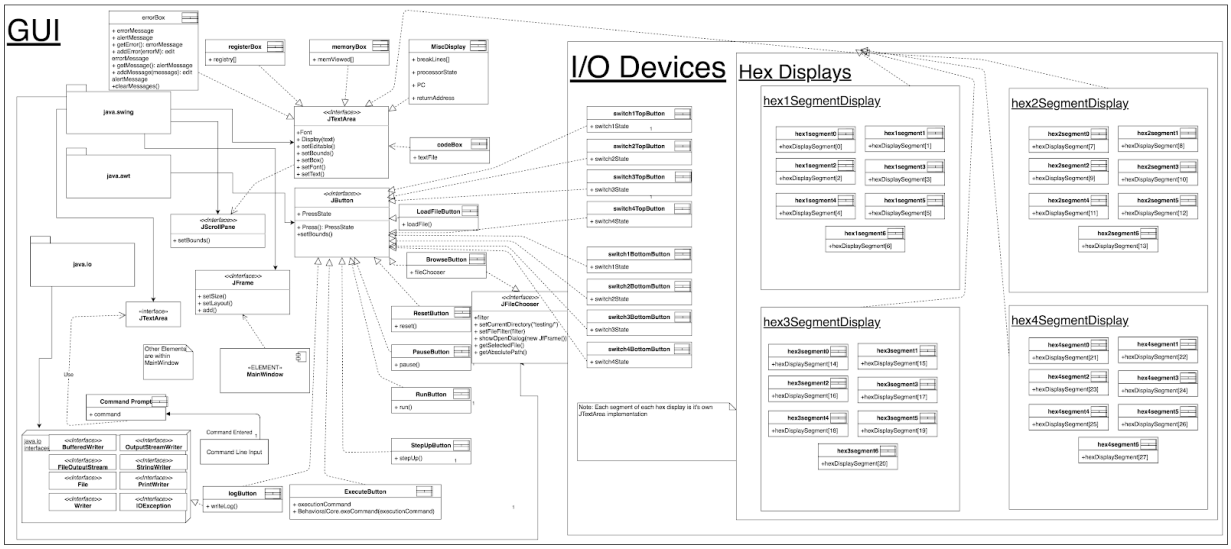

Figure 2: GUI

Figure 3: Processor

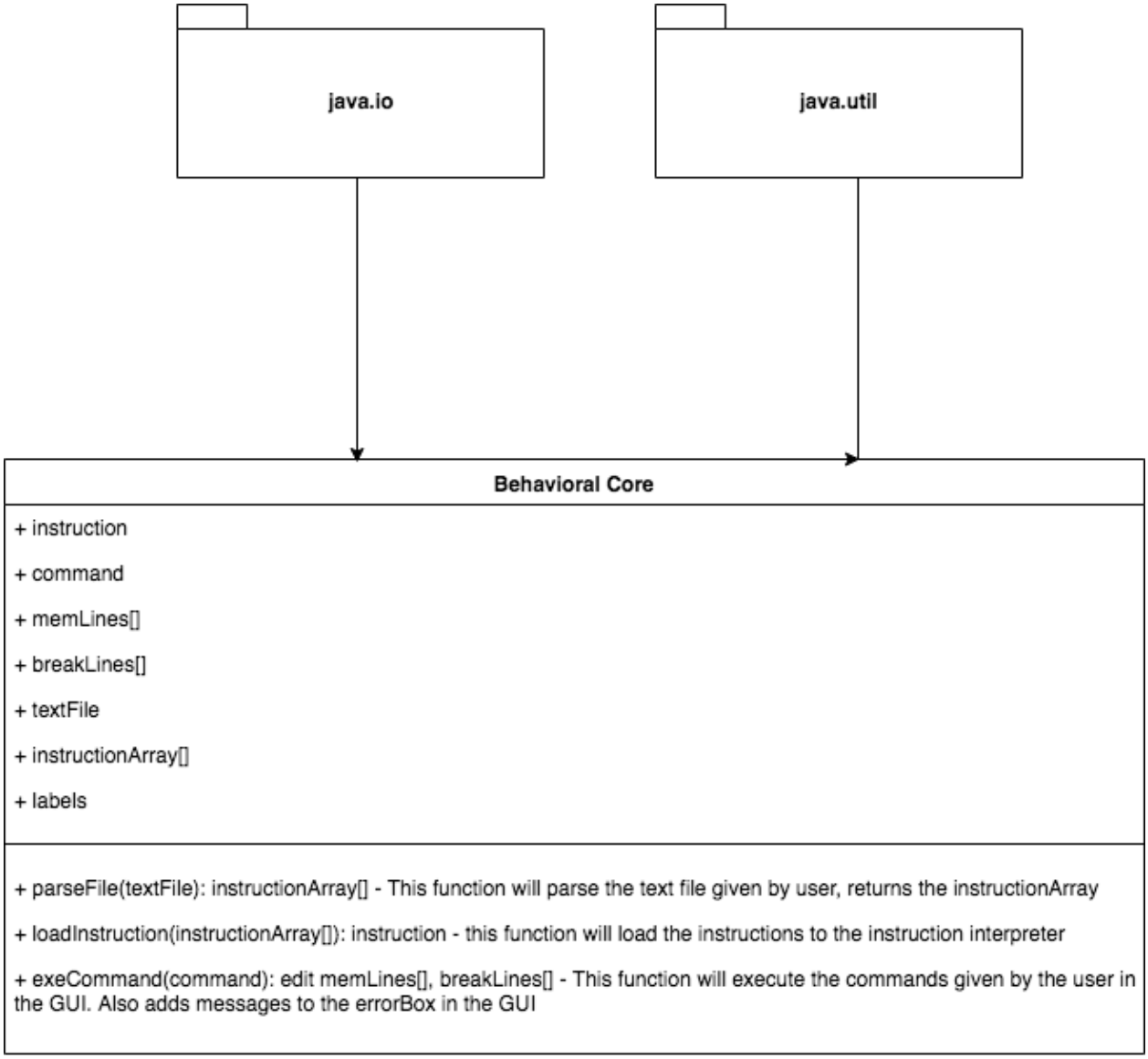

FIgure 4: BehavioralCore

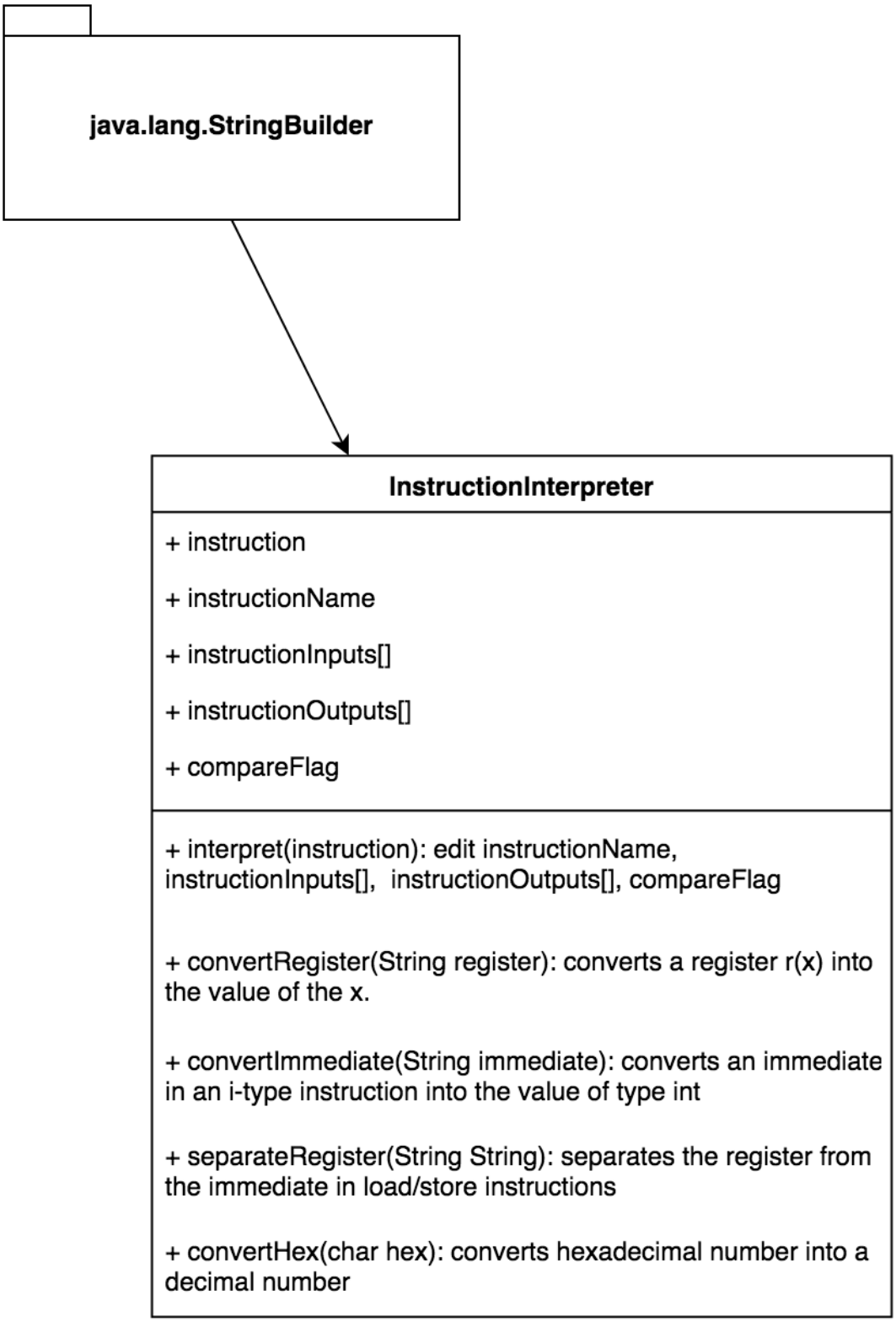

Figure 5: InstructionInterpreter

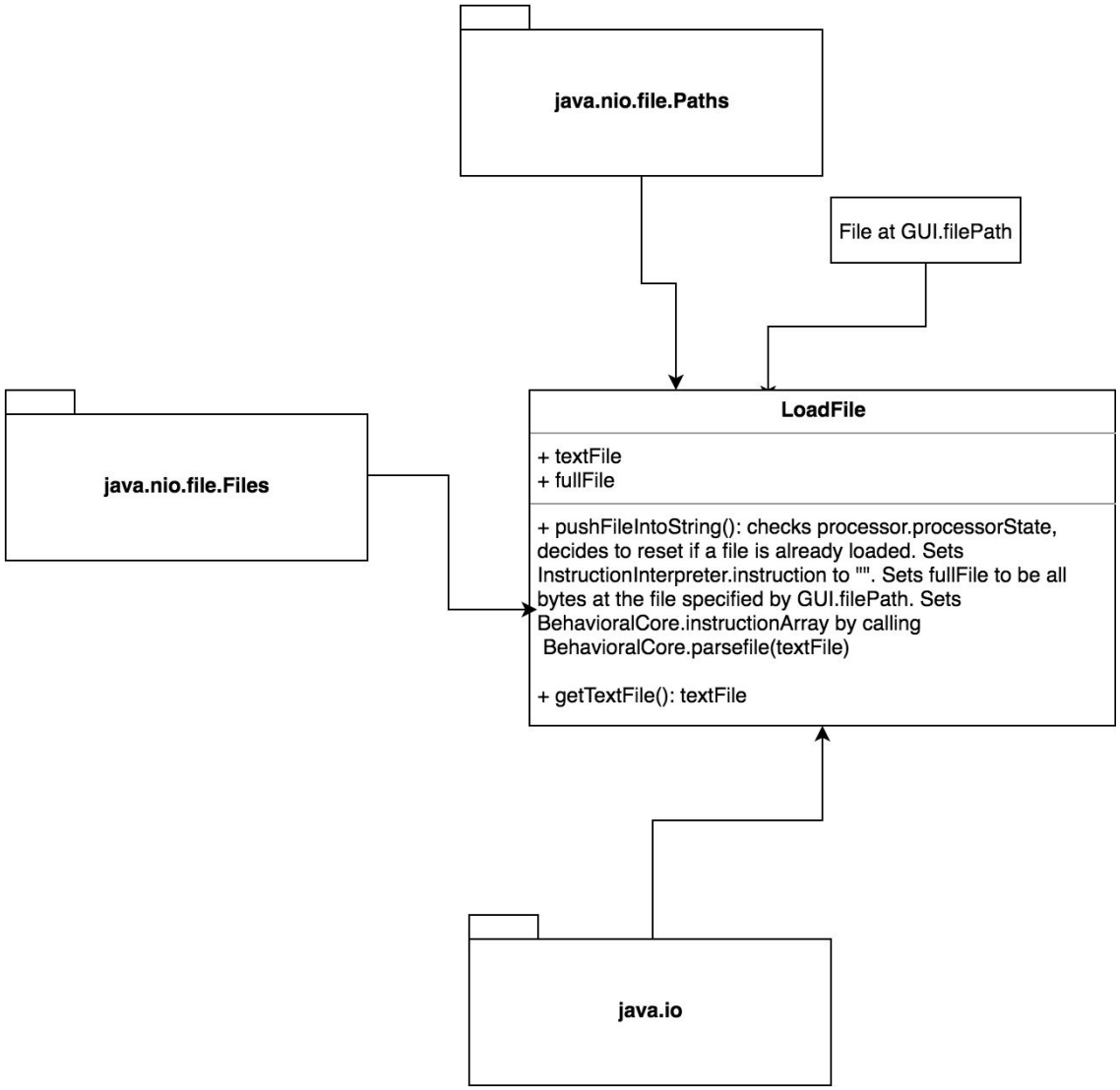

Figure 6: LoadFile

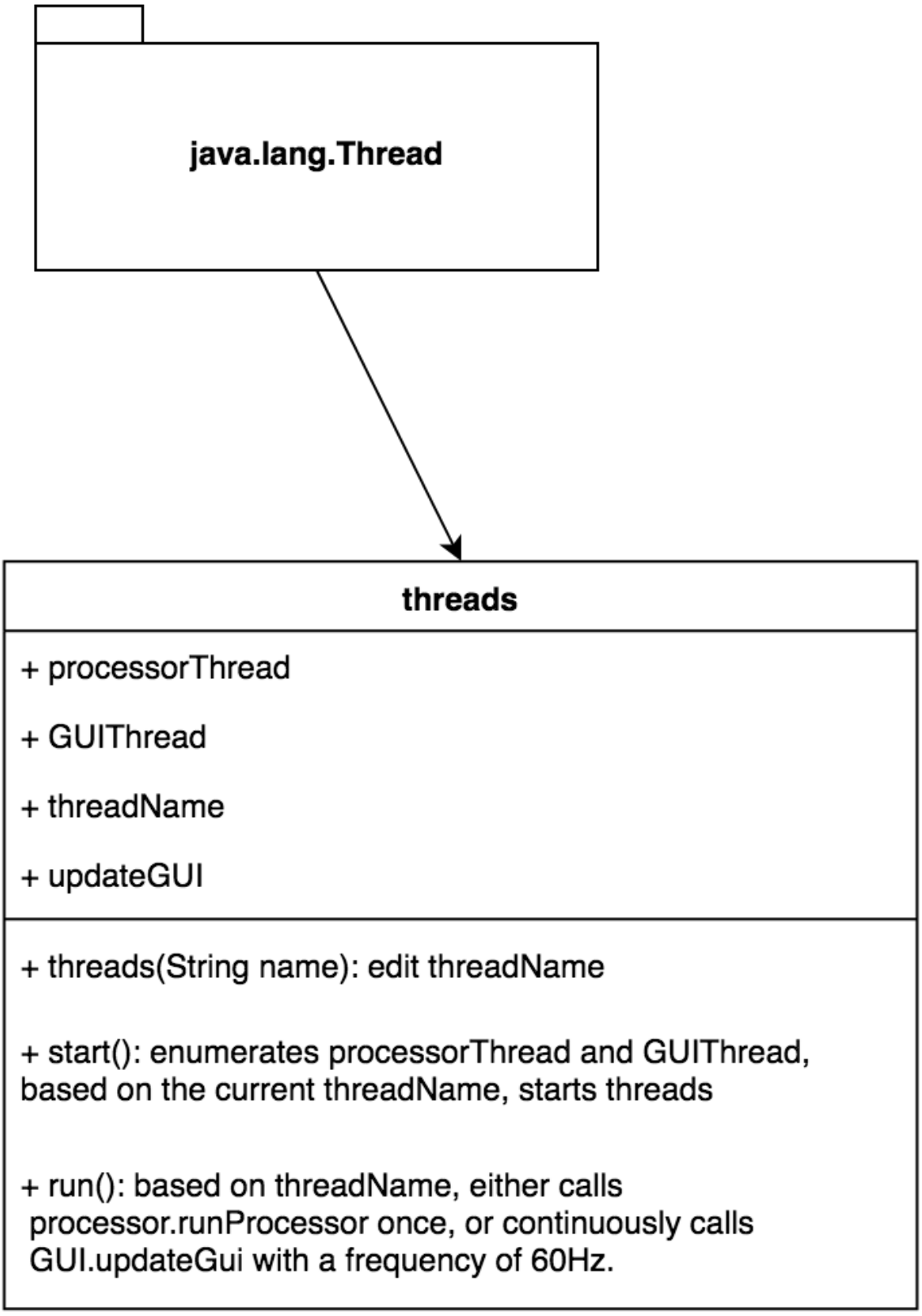

Figure 7: threads

Glossary

AWT - AWT (Abstract Windows Toolkit) is Java's original user-interface widget toolkit before Swing.

Behavioral Core - The brain of the program. This interacts will all of the other classes in the program.

GUI - GUI (Graphical User Interface) is a way that a user can visually interact with the program.

Swing - Swing is a widget toolkit in java used to produce GUIs. It provides a more sophisticated set of GUI instructions than AWT.

Thread - A thread is the program's path of execution.