E552 Final Project: Circuit Simulator

Running the Project

This project uses Maven as a build system. To run the project, use the exec plugin like so:

```
mvn compile exec:java -Dexec.args="circuit_name inputs..."
```

`-Dexec.args=""` is used to pass CLI arguments to the program. The first argument should be the name of a circuit in the `saved_circuits` folder (not including the `.txt` extension). The second and any subsequent arguments should be strings containing only the characters '01xX ', representing the signal(s) to be passed to the circuit.

Circuit File Format

Note: there is currently no ability to make comments in circuit files, I'm just using "//" here to make the explanation easier.

```
IMPORT notnot   // defines what circuits we are importing in
                // must be contained in the saved_circuits folder

temp orary -> out   // defines the contacts of a circuit, ie. the
                    // names of the inputs and the outputs
                    // inputs are on the left of the arrow
                    // outputs are on the right

notnot temp -> temp1    // defines an internal component of type "notnot"
                        // all arguments after the component type and to
                        // the left of the arrow are input wires of the
                        // component, arguments after the arrow are
                        // output wires
notnot orary -> temp2   // same as above
AND temp1 temp2 -> out  // defines an internal component of type AND
                        // (the built-in AND gate), supplying two input
                        // wires and one output wire.
```

Names of the built-in gates:

- `AND`
- `NOT`
- `OR`
- `NOR`
- `NAND`
- `XOR`
- `XNOR`

Project Goals

The primary goal of this project is to create a working logic gate simulator which allows the composition of logic gates to create more complex circuits. These circuits should also be able to contain other circuits, as well as possible feedback loops that lead to the circuit stabilizing.

Additionally, the project should have a way to save and load circuits designs to/from a file, as well as a way to provide input signals and visualize the output of circuits over time.

Project Structure

Type Hierarchy Overview

Enumerations:

- Signal

Interfaces:

- Logic

Classes:

- Wire
- Contact
- Gate
  - GateAnd
  - GateOr
  - GateNot
  - GateXor
  - GateNand
  - GateNor
  - GateXnor
- Circuit
  - FeedbackCircuit

Exceptions:

- MalformedSignalException
- InvalidLogicParametersException
- FeedbackCircuitDetectedException

`enum Signal`

Signals represent the current value on a wire. They can be high (HI), low (LO), or unknown (X). The signal enum also has other support methods defined on it which make the conversion to/from strings easier.

`class Wire`

Wires represent the connectors between different gates and circuits. They hold the Signal they contain, as well as a name to make it easier for us as humans to check which wires are where, etc. The name isn't necessary for calculations, however.

`interface Logic`

Logic represents something that can accept inputs, performs some transformation, and yields some outputs. This will be implemented by the base Gate classes, as well as the circuit classes.

`abstract class Gate implements Logic`

Represents the smallest units of logic. Gate implements all the methods of Logic, but remains abstract as it leaves the `propagate()` method to its children to implement. Gates are assumed to have 1+ input wires, and a single output wire (except for the NOT gate).

`class GateXXX extends Gate`

There are six child classes of gate that all have the same structure, a simple constructor which calls the constructor of their parent, and overriding the `propagate` method to perform the logic from `inputs` to `output`. The only special case is GateNot, which has a constructor which takes a single `Wire` as input, as opposed to a list like the other Gates.

`class Contact implements Logic`

Circuits can contain many things internally, gates, inner wires, and entire other sub-circuits. Once a circuit is created we'd like to only deal with the points of contact the separate the outside world from the interior of the circuit. To ensure this functionality circuits will always have a Contact object attached to every of its input and output wires.

A contact has two Wire references, one which goes into the contact, and the other which goes out. The circuit's own wires (and their names) are preserved, but we can still hook them up to the outside world. The `in` wire always enters the contact, and the `out` wire always exits it. The way to know whether this is an input or output connection is via the `inbound` boolean.

`class Circuit implements Logic`

A `Logic` structure which contains other `Logic` components that are wired together. This includes basic `Gate`s as well as other possible sub-circuits. The one key limitation of `Circuit`s is that they cannot contain feedback loops. If a feedback loop is detected during construction, an exception will be thrown. This

limitation allows us to guarantee that a single call to the `propagate` / `inspect` methods will fully update all outputs, and they won't change until the inputs are modified.

`class FeedbackCircuit extends Circuit`

Removes the limitation of the `Circuit` class of having feedback loops. A single call to `propagate` is not guaranteed to stabilize the outputs, so the `propagate` function will recursively call itself until the outputs do not change. This approach was the simplest way to implement the desired functionality, but it comes with a few downsides.

1. Circuits which don't stabilize
    - This will lead to infinite recursive calls of the `propagate` until the JVM runs out of memory and terminates. A possible solution would be to allow the user to define a recursion depth limit, and we stop evaluating the circuit past that point, either keeping the potential garbage value or throwing an exception stating that evaluation failed.
2. Worse performance
    - The need to call `propagate` multiple times will cause performance drops compared to standard circuits for massive circuits, which is why they've been separated into two different classes. At runtime when a circuit file is being read in, the program will first try to construct the circuit as a normal `Circuit`, and only if that fails will it use a `FeedbackCircuit`. This is also the case when constructing sub-circuits inside a `Circuit`.

`class XXXException extends Exception`

There are three custom exceptions defined, all of which extend the Exception class (so they are checked by the compiler). They're all fairly self-explanatory based on the names, however the user should only ever see the `MalformedSignalException` and `InvalidLogicParametersException`.