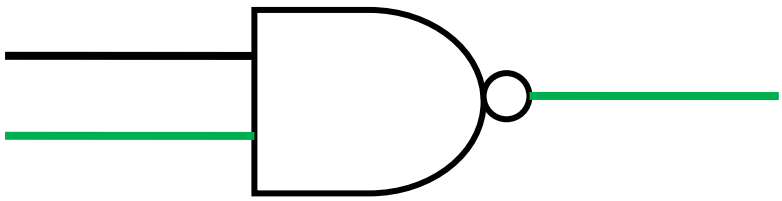
Logisim Technical Manual

December 6, 2009



Chris Stewart

John Turgeson

Alex Whitney

Contents

Introduction

In the Fall semester of 2009, the Computer Science department of Georgia Tech was looking to replace LogicWorks, a logic circuit simulator for Windows used in coursework. LogicWorks was used by students to build simple circuits using basic components such as AND and OR gates, and slowly work their way up to entire processor datapaths. LogicWorks became an integral part of the course curriculum, as students often used it to complete assignments. With many students moving away from computer clusters towards personal laptops running Linux or Mac OS, the department was interested in a multi-platform solution.

The Computer Science department employed a senior design team to create a replacement for LogicWorks. After receiving the requirements, Logisim was discovered as a suitable replacement. Logisim is an open source logic circuit simulator developed by Carl Burch, a professor at Hendrix College. It was developed for educational use and is currently being employed by a number of schools across the world. Logisim is currently one of the most mature and developed applications of its kind freely available on the Internet.

While it is open source, Logisim lacks any significant documentation. The lack of any description of the architecture makes development of new features or bug fixes by third parties difficult. Furthermore, the complexity and size of the application makes learning the system an arduous task. To make it easier for developers to improve the application, the senior design group was tasked with providing sufficient documentation for the code.

This document is the product of the senior design project. It contains a discussion on various aspects of the Logisim codebase. It is not intended to be read from front-to-back; instead, readers should focus on portions of the document that apply to a potential change to the program.

This document is divided into three parts. The first part describes some common developer operations. Specifically, these operations include details on how to write custom components. The next section describes a system-level view of the application. Subsections explain the workings of a particular component of the system, such as file management or circuit representation. The final section lists a number of problems in the code, most of which include suggestions for fixes.

Common Operations

Creating Custom Components

Preferred method is using Instance package.

How to write libraries, import.

Using the Instance Package

Creating custom components using the Instance package is fairly straightforward. The classes involved with this process handle most of the heavy lifting, so the only code that needs to be written is that which controls the component logic and drawing. For components performing relatively simple operations, this package should be sufficient.

To begin, create a class which extends InstanceFactory. InstanceFactory creates a default Component that implements some standard behavior. For the two behaviors which are non-standard, the created Component calls methods in the factory of origin. These methods are marked abstract in the InstanceFactory definition and must be implemented in any descendants. These methods will be described later in this section. There are a number of optional methods that can be overwritten that will further control the behavior of the Component. These are also described later.

First, the mechanisms involving the setup of the factory will be described. The factory's default constructor should be overwritten. At a minimum, it needs to establish a name for the component. There are two names attached to each component: an internal name and a display name. The display name is the one used in the component list on the left pane of the GUI. In existing components, the display name is selected dynamically based on the application's locale, allowing for integration of multiple languages. The names should be set by making a call to one of the super class's constructors, shown below. InstanceFactory ("component") is equivalent to InstanceFactory ("component", "component").

```
InstanceFactory(String internalName);
InstanceFactory(String internalName, string displayName);
```

There are a few other items that can be adjusted in the factory's constructor.

- Attributes

  A component's attribute list appears in the properties pane on the left side of the UI when it is selected. Attributes are one way for the user to modify a component's behavior during run-time. One common attribute is facing, which modifies the orientation of the component. By default, components created by InstanceFactory do not have any attributes.

  No attribute values are used in the default component behavior. Any added attributes should be used by the additional code in the InstanceFactory. All attributes will be visible and changeable on the main UI, so be sure not to add any attributes that should not be visible to the user. To set the attribute list, use InstanceFactory's setAttributes method.

- Icon

  This controls the icon that is displayed in the component list on the left pane of the UI. If no icon is specified, a default icon will be used. There are two ways to specify an icon. If the icon is included in /logisim/resources/icons, the name of the icon file is sufficient if set using InstanceFactory's setIconName(String iconFilename) method. At run-time, the InstanceFactory will load and paint the icon.

  Alternatively, the icon can be set directly using an Icon object. This process would be necessary for anyone creating components as part of a redistributable component library. The icon can be set using InstanceFactory's setIcon(Icon componentIcon) method.

  If an icon is set using both methods, the most recent icon that has been set will be used.

- Offset Bounds

- Facing Attribute

Configure instance

Propagate and paintNewInstance

Optional handlers (destroyInstance, instanceAttributeChanged

Using ManagedComponent

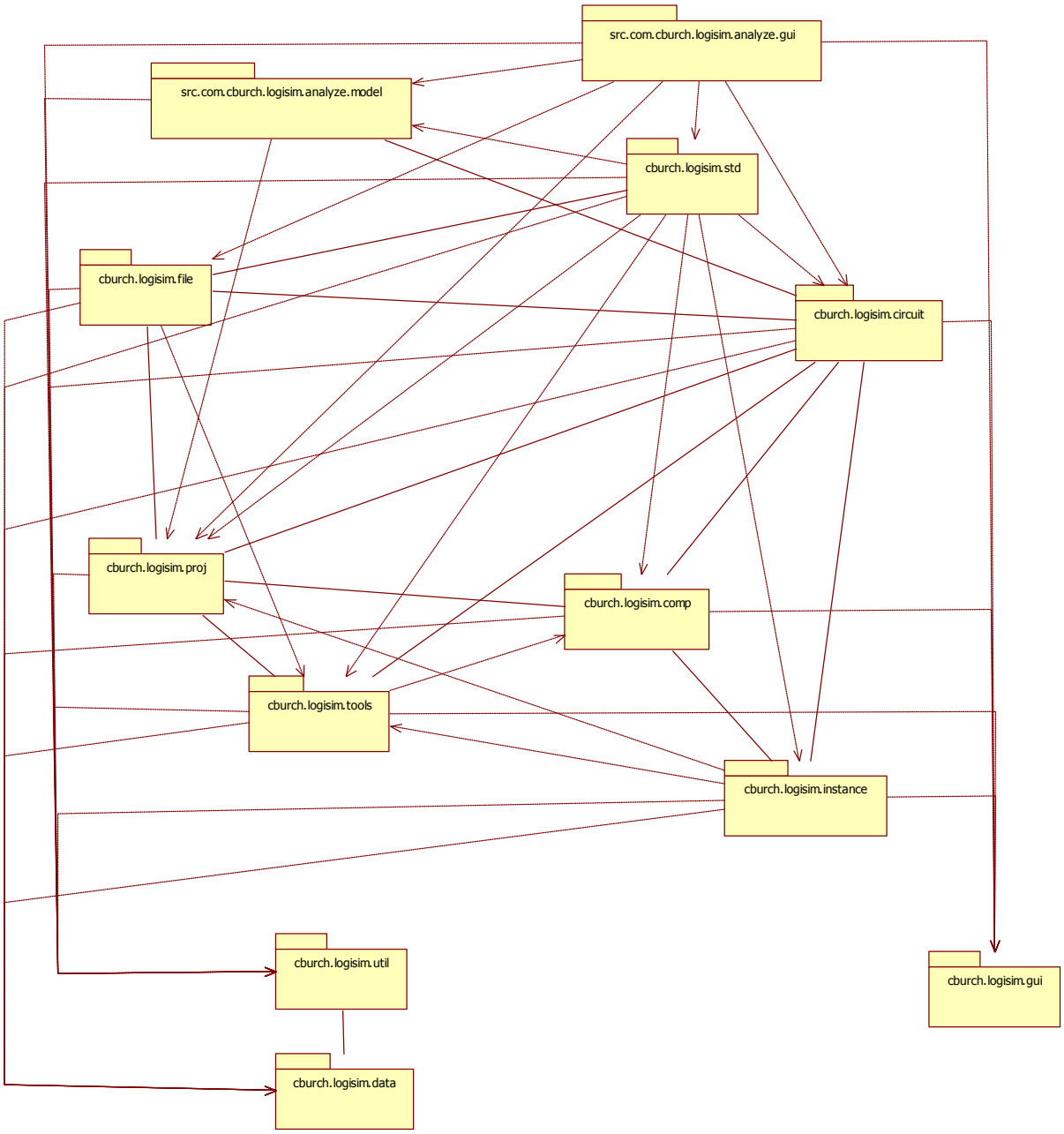
Is this any better than directly implementing Component?

System Architecture

This section details the Logisim architecture.

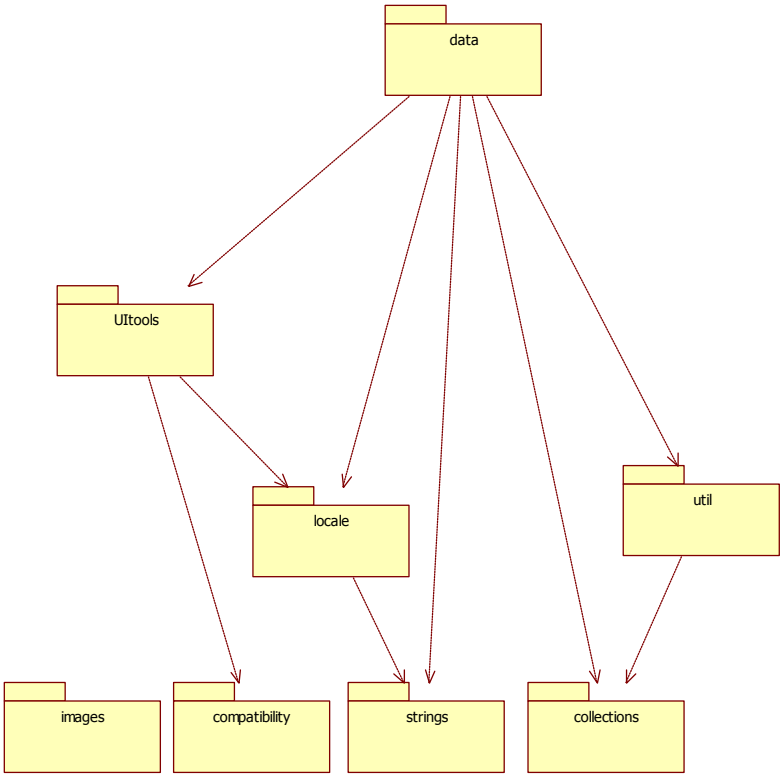
Package Design

The system-level view of the Logisim architecture is fairly grim. As apparent in the class diagram below, there is strong coupling and no apparent hierarchy. To some degree, this is a little expected given the history of the application and the desire for optimized code. Unfortunately, the complexity of this configuration makes maintenance and system changes time consuming and prone to the introduction of errors.



In general, classes are separated by function. For example, classes that manage the circuit simulation and representation are located in the logisim.circuit package, while classes relating to components have been palced in the logisim.comp package. However, the coupling between classes makes these packages more of an organizational convenience rather than a way to package and reuse components.

Some minor changes were made to the original architecture to improve the system-level design. These changes mostly involved breaking up the util package into smaller packages to improve reusability. Furthermore, a couple classes were moved from util to data, so that now no classes from the "utility" packages reference the data package. While these changes do not impact the overall complexity in a significant way, they do improve the architecture of these lower-level packages.



Circuit Representation

At a high level, the backend representation of circuits is fairly straightforward. CircuitStates are used as a top-level circuit class. Each of these classes contains a Circuit, which is composed of a number of Components and Wires. In the current implementation, there are a few different implementations of Component. A few "low-level" components have special behavior that required them to be implemented much closer to Component. Existing library classes, on the other hand, were implemented using the instance package, which allows them to function without being exposed too greatly to the underlying implementations.

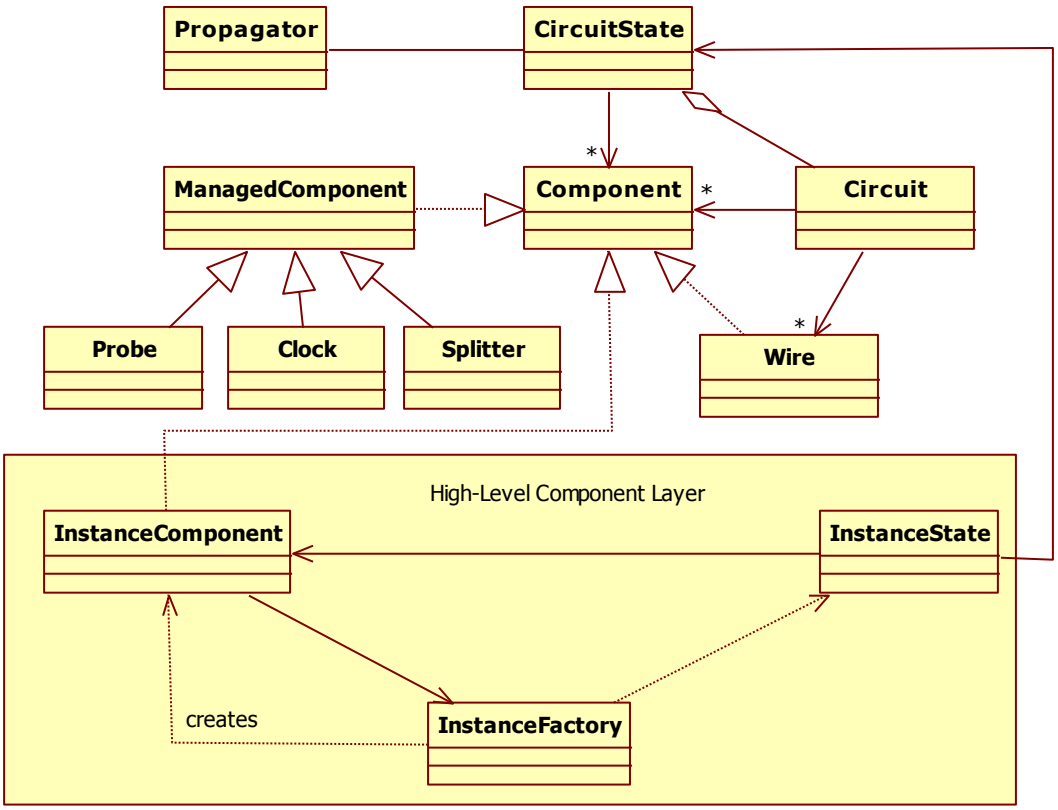


Figure 1: Core Circuit Structure

CircuitState is primarily responsible for keeping track of the Components and Points whose values have changed since the last propagation. CircuitState lets the Propagator know when a value change is requested. Curiously, CircuitState contains code to enable it to be a node in a tree. However, it does not appear that this code is heavily used.

Circuit contains a number of convenience methods for dealing with collections of Components. It is also responsible for drawing all of the components in the Circuit.

Propagation of values through the circuit can get a little confusing. Essentially, whenever a component is added to a circuit, or when a value in a circuit changes, the corresponding component is sent to the propagator. When the propagator steps, the changed values are passed forward through the circuit. The details are provided in the sequence diagram in Figure 2.

It's important to note that stepping once typically won't propagate values through the circuit completely. Step needs to be called enough times to ensure that changed values cause the circuit to reach a steady state. Generally each step would correspond to one "clock pulse." Alternatively, Propagator's propagate method will simply continue until the circuit reaches its end state. It will continue to step until the circuit's values stop changing, or until it has stepped 1,000 times. If the circuit

does not reach a steady state by the time it terminates, it is considered to be “oscillating”. The Canvas will show an error in this case.

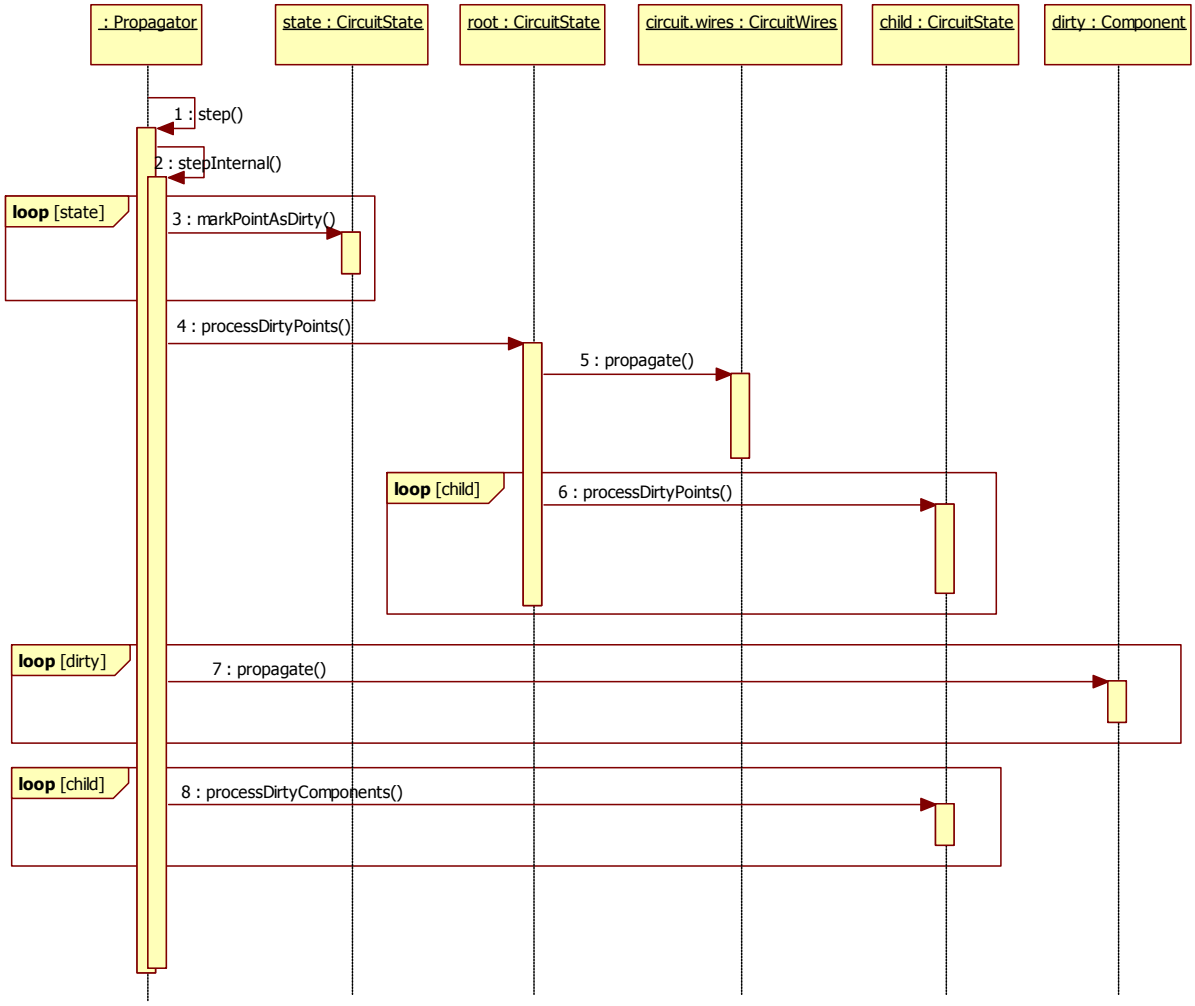


Figure 2: Sequence of calls for Propagator's Step method

All elements that can be added to a Circuit are descendents of Component. A few of these Components are atypical, which requires them to be implemented very close to Component. Wire, for example, is strange in the sense that it does not have input and output pins. Instead, values placed on either endpoint will propagate to the other. Because of this fact, Wire is the only Component that is a direct implementation of Component.

As show in Figure 1, Probe, Clock, and Splitter are all subclasses of ManagedComponent. ManagedComponent is a simple abstract implementation of Component, providing some simple common functionality for most of the methods. These classes are still fairly complex.

All of the components in the libraries make use of the Instance package classes. These classes make it very easy to create a component with a custom drawing and custom behavior. In addition to providing

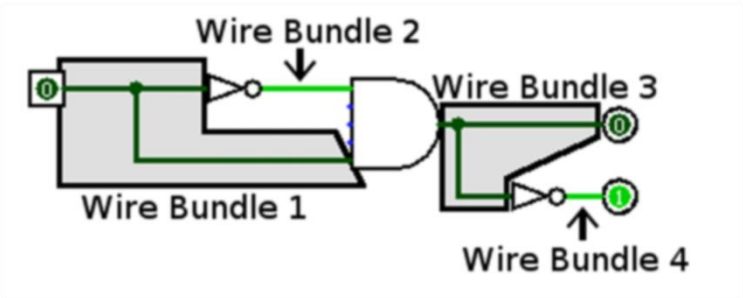
default behavior, the Instance package increases abstraction by inserting a layer between the components and the simulation code. To make a custom library component, simply create an implementation of InstanceFactory and override any methods where custom behavior is desired.

Wire Specifics

Passing data around the circuit is essential to Logisim's operation. So essential, in fact, that the Wire class is handled very differently than other components. Wires are handled so differently, that it may be hard to understand how data moves around a circuit. This section will explain these mechanics in order to increase the reader's understanding of the simulator.

Wire, perhaps surprisingly, does very little of the work involved with moving data through a circuit. In fact, the Wire class is not even responsible for drawing itself on the Canvas! CircuitWires does most of the work in this area, including propagation, leaving Wire to handle little more than keeping track of its two endpoints

When a wire is added to a circuit, it is given to the CircuitWires class. This class manages all of the wires, and propagation across those wires, for a particular circuit. CircuitWires takes wires and constructs a BundleMap, consisting of a WireBundle for each section of wire. These WireBundles are simply a tree structure describing how wire segments are connected to one another. These bundles actually don't contain any wires, but simply a list of points that are connected by wires. Note that Wires are split at junctions, and horizontal and vertical segments are always distinct Wires, so that WireBundles 1 and 3 in the image below both contain five points.



Things get a little complicated when Splitters are involved. After computing all of the WireBundles, CircuitWires determines which WireBundles are connected to all of the Splitter's endpoints (on the merge and split side). Next, all of these WireBundles are merged. The merge step is a little difficult, as the WireBundle must know how to handle each bit. To accomplish this task, WireBundle keeps separate "WireThreads" for each data bit. When connected to another splitter, these threads "merge".

CircuitWires propagates only WireBundles that are mapped to from a dirty point. The value for each WireThread is determined, and then the total wire value is computed and set on each point connected by the WireBundle (using CircuitState's setValueByWire).

Interacting With a Circuit

How to manipulate a Circuit. From creation to adding Components, changing values, etc. Should probably deal with Circuit at Circuit/CircuitState level, so make sure to cover the use of the Simulator class.

Projects and Files

All projects begin and end as a Logisim circuit file. In a way, these files allow for significant flexibility with respect to the UI layout. While the underlying menu items and sidebar locations can't be changed from the saved files, the order and contents of the Library menu can be adjusted, as well as the components placed in the toolbar. The format of these save files is XML, so they can be edited manually to customize the UI. This system is flawed, however, as only user-editable values are saved, making it impossible to retain information that needs to be hidden from the user. This limitation restricts component design options.

When Logisim is loaded, it generates a new project from a template file. This file is an internal resource file and can be located at logisim/resources/default.templ. The contents of this file setup the initial project settings. These settings include the list of libraries to include, project UI settings, mouse button mappings, and toolbar elements. Making adjustments to the default.templ file will allow changes to be made to the default UI. For example, all of the libraries except for gates and base could be removed, preventing them from being loaded at startup. Formatting for the template file and saved files are identical. The file format is XML. Details are provided below.

- **project**
  This is the outermost node.
  "version": Currently just set to 1.0 by XmlWriter
  "source": The version of the Logisim application that created the file.
- **project.lib**
  Directs the application to load the specified library.
  "name": sets the file-specific name for the library. This is used later with the toolbar.
  "desc": specifies the library to load by name. Internal libraries are prefixed with "#".
- **project.lib.tool**
  Allows setting of default attributes.
  "name": specifies a component within the library
- **project.lib.tool.a**
  Sets a new default value for the specified attribute. Note: these attributes are the same that are displayed on the left panel for the user to edit.
  "name": the attribute name (varies by component)
  "val": the default value for the attribute
- **project.main**
  Sets the main circuit
  "name": the name of the circuit to set as main

- **project.options**
  Contains the project-specific option attributes
- **project.options.a**
  Sets the value of a specific option attribute for the project. See com.cburch.logisim.file.Options for the full list of attributes.
  "name": The attribute name
  "value": The value to set the attribute to
- **project.mappings**
  Contains a set of mappings from mouse events to tools
- **project.mappings.tool**
  Maps a tool to one specific mouse event.
  "lib": The file-specific library name that contains the tool.
  "name": The internal tool name. All tools are valid, but useful ones include "Poke Tool", "Text Tool", "Wiring Tool", and "Menu Tool"
  "map": The mouse event. These must include at least one mouse button and may optionally include a modifier. Valid buttons are "Button1", "Button2", and "Button3". Valid modifiers include "Ctrl", "Alt", and "Shift". Items should be space delimited.
- **project.toolbar**
  Details the configuration of the toolbar. Items in the toolbar node are drawn in list order.
- **project.toolbar.sep**
  This node represents a vertical separator bar.
- **project.toolbar.tool**
  Specifies a tool to add to the toolbar.
  "lib": The file-specific library name that contains the tool.
  "name": The internal tool name. Tools and Components can be specified; Components are later wrapped in an Add Tool.
- **project.toolbar.tool.a**
  Sets the attribute of the tool to a specified value when it is used
  "name": The internal attribute name. The list of valid attributes will vary based on the tool or component.
  "val": The value to set the attribute to
- **project.circuit**
  Projects can have a number of circuits. Each contains a listing of the wires and components contained in the circuit.
  "name": The name of the circuit
- **project.circuit.wire**
  Describes a wire in the circuit. Points should be specified using "(<x>, <y>)", where (0, 0) is the top left corner of the canvas. Order of the points can be significant. The "from" field should be to the left or higher up than the "to" field. In other words, a wire from (x_1, y_1) to (x_2, y_2) is valid if

and only if $x_1 <= x_2$ and $y_1 <= y_2$. Also note that Wires must be only horizontal or vertical, so either $x_1 = x_2$ or $y_1 = y_2$, but not both.

"from": A point on the canvas

"to": A point on the canvas

- **project.circuit.comp**

  Describes a component in the circuit

  "lib": The file-specific library name that contains the tool.

  "name": The internal component name.

  "loc": The (x,y) location of the component on the canvas, where "(0,0)" is the top left corner of the canvas.

- **project.circuit.comp.a**

  Sets the attribute of the component to a specified value

  "name": The internal attribute name. The list of valid attributes will vary based on the component.

  "val": The value to set the attribute to

Drawing Circuits

Cover use of Canvas for circuit painting. Also, under what conditions the circuit is redrawn. Also discuss drawing options from the POV of the Components. Probably very similar with Instance and Component.

Custom Data Structures

Attributes, AttributeSet, AttributeSets, SmallSet, ArraySet, Cache, DirectedAcyclicGraph, EventSourceWeakSupport, PriorityQueue, UnmodifiableList. Discuss purpose, use of, etc.

Subcircuits

How they are implemented, any strange things about them, and so forth. Room for customization?

Graphical User Interface

Details the GUI, how it interacts with Circuit, structure, etc. Talk about Frame and all those GUI classes.

Issues and Potential Fixes

Architectural Problems

ComponentAction

Currently, implementations of ComponentAction perform some functions that populate a particular property in ComponentAction. On "doIt", ComponentAction makes the calls to circuit to remove, add, or modify particular elements in circuit.

This is a poor implementation choice that violates OO principles. Implementations of ComponentAction should provide their own implementation of "doIt." Placing the Circuit calls in a central class doesn't increase maintainability and does not introduce any flexibility. In fact, there does not seem to be any common code between any of the ComponentAction implementations, other than clone.

Wire

Wires are implementations of Component, but they are often treated very differently. Not only is this a violation of OO principles, but it also makes program logic confusing and has encouraged poor class design. For example, Circuit keeps track of Wires and Components in two separate lists. By increasing abstraction, Circuit should be able to treat Wires and other Components in the same way, allowing the objects themselves to handle special situations.

Maintainability Improvements

CircuitListener

CircuitListener is often used as a local variable with the name "what." This name was selected to be clever when invoking the "circuitChanged" method.

Single-Line If Statements

For improved maintainability, it is important to use block if statements. While single-line if statements are syntactically correct, it's harder to read and is prone to errors when being maintained. These should be expanded to multiple lines and used in block form.

Generics

The lack of generics in the application makes reading the code much more difficult. Furthermore, it encourages poor design and increases the chances of failure. While generic support was not in Java when the application was first developed, that is not a reason to avoid their use. For type safety reasons, and for ease of maintenance, it is highly recommended that generic support be added in all situations where it is appropriate. For example, when using Java collection classes, the generic version should always be used. These cases are identified by most modern IDEs.

For the same reasons, support for generics should also be encouraged for collection-like custom classes, such as SmallSet.

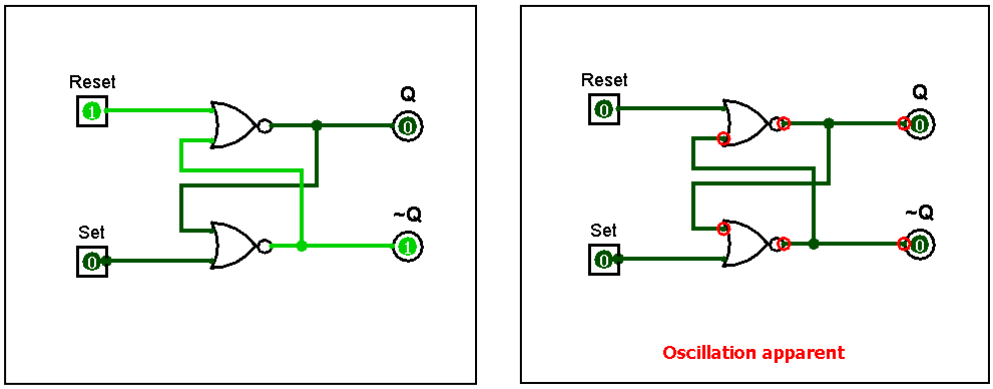
Existing Bugs

Empty Template

Setting the project settings to use empty templates (File -> Preferences) causes the application to fail when creating new circuits. Because the preference is saved when the application is closed, this problem persists until the preference file is deleted.

Recursive Propagation

Logisim fails when loading circuits with "recursive propagation." Below are screenshots of an RS latch built using Logisim. The image on the left was taken immediately after being built. The latch works as it should. After being saved and loaded, Logisim reached the state shown in the image on the right. An "oscillation" occurred because Q and $\overline{Q}$ continuously changed value.



The reason this error occurs is because the Q and ~Q lines, when the entire circuit is propagated simultaneously, are both initially 1. The NOR gates reverse the output on both lines, so in the next "clock pulse" both Q and ~Q lines switch to 0. Since the Reset and Set lines are both 0, the output of the two NOR gates is 1, which repeats the cycle.

When manually building the circuit, the wires are added sequentially, so that the circuit is able to determine the output of one NOR gate before computing the second.