Computing Write Up

# Analysis

## The problem

The person I designed the program for is myself as an Electronics student. As such, there was no particular communication between programmer and client. I wished to practice my programming, and a topic that caught my interest was logic circuit simulations – I had the start of an idea of how I would design it, and so I began to program it. I am advanced enough in my programming and was sufficiently progressed through the project, that when it came time to focus on the coursework, I decided to keep going with the circuit sim.

## The current system

As an Electronics student, we use ISIS for our circuit simulations, to design circuits before, or in lieu of, building them in real life. ISIS has a large library of existing components, as well as supporting analogue signals. However, it does not support custom components.

### Problems

1. ISIS does not support custom components. There is no way to create a component that can be placed and reused like any other unless it is already within the library, possibly excluding writing code to control it.
2. ISIS has very little customisability of components. Only components such as resistors and capacitors have any form of customisability.
3. I have no understanding of ISIS under the hood. I merely know how to use it as any consumer would. This is not a true problem with the program, but a source of my original motivation to create an alternative.

## Objectives

1. A way to design and simulate a circuit.
2. A way to create a circuit, then later reuse it as a single component.
3. A way to customise components.

# Design

## Namespaces

* CircuitMaker is the namespace for the whole program.
  + Basics contains the basic classes such as Pin, Pos, Board, and others, and also contains the interfaces implemented by the Components.
  + Components contains all my predefined components.
  + GUI contains all the GUI code, including GUIForm, ComponentSelectionForm, and RenameBoardForm
    - Settings contains the whole component settings system, including the GUI, found in SettingsForm.
    - ExtApp contains the whole GUI for editing the external appearance of a board, found in ExtAppEditorForm.

## Forms

### GUIForm

GUIForm is the primary form, containing a selection of options in the drop down menu found at the top of the window that many apps have, and the editor itself (the class known as Builder).

### ComponentSelectionForm

ComponentSelectionForm is a simple form that contains a list of all hard-coded components (found in the CircuitMaker.Components namespace). It allows for selecting one of them and placing them in the editor.

### ExtAppEditorForm

ExtAppEditorForm is a form for designing the external appearance of the component that will be created when selecting a file from the Create Board Component option. It allows for resizing of the component, positioning of all the pins, and positioning and scaling of the graphics of all the IGraphicalComponents within.

### RenameBoardForm

RenameBoardForm is a simple form with a text box to enter the new name of the board.

### SettingsForm

SettingsForm is a dynamic form created either when double clicking on a component, or choosing the Open Settings option when left clicking on a component, as long as the component has settings to edit. It creates a label and asks the component for an input control for each setting, and then arranges them vertically. The component retains a reference to all of these input controls, and then alters its own fields according to the value in the controls if the OK button is used to close the SettingsForm.

## Classes

[demonstrate the inheritance of all the components, the relationship between Board and all the BoardContainerComponents, and other significant systems]

## File Save

Files are written straight to binary. The encoding is as follows:

### File

* Top level Board.
* Sub-Boards (4-bit length followed by boards end to end).

### Board

* Name (string: 1 bit length followed by characters).
* External Size (two 4-bit numbers, width then height).
* Components (4-bit length followed by components end to end).
* Wires (4-bit length followed by wires end to end).

### Component

* ID (string).
* Details (string).
* Position.
* Rotation (member of Rotation enumeration converted into string format and stored as such)

### Wire

* Position of one end.
* Position of the other end.

### Position

* X value (4-bit integer).
* Y value (4-bit integer).

# Testing

Any feature was tested before fully moving on to any other feature. Several bugs are likely to still exist.

# Evaluation

1. Boards can be designed and simulated perfectly.
2. Custom components can be created as a board, saved, and reused in any board by selecting the file.
3. Programmed components can be customised through the use of the SettingsForm.

There are features of C# I was not aware of when beginning the project that I would have used, but in order to identify them I would have to start over, as it is likely some design decisions would be different. Additionally, due to the complexity of the program, there are many bugs that I never got the chance to fix – hopefully there are no breaking issues hiding. I haven’t had the chance to get someone to try and use it.

Thinking about it as an Electronics student instead of the programmer, I see it as fully functional for my purposes.

# Code

## Github Links

<https://github.com/LastedApple3/CircuitMaker/tree/master/CircuitMakerBasics>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/Basics.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ComponentSelectionForm.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ComponentSelectionForm.Designer.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/Components.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ExtAppEditor.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ExtAppEditor.Designer.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ExtAppEditorForm.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/ExtAppEditorForm.Designer.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/GUIForm.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/GUIForm.Designer.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/Program.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/RenameBoardForm.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/RenameBoardForm.Designer.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/SettingsForm.cs>

<https://github.com/LastedApple3/CircuitMaker/blob/master/CircuitMakerBasics/SettingsForm.Designer.cs>

## CircuitMaker

### Program

internal class Program

{

[STAThread]

static void Main(string[] args)

{

ComponentRegisterer.RegisterComponents();

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new GUIForm());

}

}

### Basics

#### TransformRestorer

class TransformRestorer : IDisposable

{

private Graphics SavedGraphics;

private Matrix SavedMatrix;

public TransformRestorer(Graphics graphics)

{

SavedGraphics = graphics;

SavedMatrix = graphics.Transform;

}

public void Dispose()

{

SavedGraphics.Transform = SavedMatrix;

}

}

#### RectangleFExtensions

static class RectangleFExtensions

{

public static int dp = 5;

private static float Round(float val)

{

return (float)Math.Round(val, dp);

}

public static RectangleF Round(this RectangleF rect)

{

return new RectangleF(Round(rect.X), Round(rect.Y), Round(rect.Width), Round(rect.Height));

}

}

#### ByteEncoding

public class ByteEncoding : Encoding

{

public override int GetByteCount(char[] chars, int index, int count)

{

return count;

}

public override int GetBytes(char[] chars, int charIndex, int charCount, byte[] bytes, int byteIndex)

{

for (int i = 0; i < charCount; i++)

{

bytes[byteIndex + i] = (byte)chars[charIndex + i];

}

return charCount;

}

public override int GetCharCount(byte[] bytes, int index, int count)

{

return count;

}

public override int GetChars(byte[] bytes, int byteIndex, int byteCount, char[] chars, int charIndex)

{

for (int i = 0; i < byteCount; i++)

{

chars[charIndex + i] = (char)bytes[byteIndex + i];

}

return byteCount;

}

public override int GetMaxByteCount(int charCount)

{

return charCount;

}

public override int GetMaxCharCount(int byteCount)

{

return byteCount;

}

private static Encoding byteEncoding = null;

public static Encoding Byte

{

get

{

if (byteEncoding == null)

{

byteEncoding = new ByteEncoding();

}

return byteEncoding;

}

}

}

#### PlacementException

public class PlacementException : Exception

{

public PlacementException(string desc) : base(desc) { }

}

#### ReadWriteImplementation

public static class ReadWriteImplementation

{

public static void Write<T>(this BinaryWriter bw, T enumVal) where T : Enum

{

bw.Write(Enum.GetName(typeof(T), enumVal));

}

public static T ReadEnum<T>(this BinaryReader br) where T : Enum

{

return (T)Enum.Parse(typeof(T), br.ReadString());

}

public static void Write(this BinaryWriter bw, Pos pos)

{

bw.Write(pos.X);

bw.Write(pos.Y);

}

public static Pos ReadPos(this BinaryReader br)

{

return new Pos(br.ReadInt32(), br.ReadInt32());

}

public static void Write(this BinaryWriter bw, Wire wire)

{

bw.Write(wire.Pos1);

bw.Write(wire.Pos2);

}

public static Wire ReadWire(this BinaryReader br, Board board)

{

return new Wire(br.ReadPos(), br.ReadPos(), board);

}

public static void Write(this BinaryWriter bw, IComponent comp)

{

bw.Write(comp.GetComponentID());

bw.Write(comp.GetComponentDetails());

bw.Write(comp.GetComponentPos());

bw.Write(comp.GetComponentRotation());

if (comp is IGraphicalComponent graphicalComp)

{

bw.Write(graphicalComp.GetGraphicalElementScale());

Point? graphicalLoc = graphicalComp.GetGraphicalElementLocation();

bw.Write(graphicalLoc.HasValue);

if (graphicalLoc.HasValue)

{

bw.Write(graphicalLoc.Value.X);

bw.Write(graphicalLoc.Value.Y);

}

}

if (comp is IBoardInterfaceComponent interfaceComp)

{

Board.InterfaceLocation intLoc = interfaceComp.GetInterfaceLocation();

bw.Write(intLoc.Side);

bw.Write(intLoc.Distance);

}

}

public static IComponent ReadComponent(this BinaryReader br, Board board)

{

string compType = br.ReadString();

if (Constructors.TryGetValue(compType, out Func<string, IComponent> compFunc))

{

IComponent comp = compFunc(br.ReadString());

Pos pos = br.ReadPos();

Rotation rot = br.ReadEnum<Rotation>();

if (comp is IGraphicalComponent graphicalComp)

{

graphicalComp.SetGraphicalElementScale(br.ReadSingle());

if (br.ReadBoolean())

{

graphicalComp.SetGraphicalElementLocation(new Point(br.ReadInt32(), br.ReadInt32()));

}

}

if (comp is IBoardInterfaceComponent interfaceComp)

{

interfaceComp.SetInterfaceLocation(new Board.InterfaceLocation(br.ReadEnum<Board.InterfaceLocation.SideEnum>(), br.ReadInt32()));

}

comp.Place(pos, rot, board);

return comp;

}

throw new PlacementException($"Could not find builtin component of type {compType}");

}

public static void Write(this BinaryWriter bw, Board board)

{

board.SimplifyWires();

WriteBoardBasic(bw, board);

Board[] boards = board.GetBoardList().Where(thisBoard => thisBoard.Name != board.Name).ToArray();

bw.Write(boards.Length);

foreach (Board thisBoard in boards)

{

WriteBoardBasic(bw, thisBoard);

}

}

private static void WriteBoardBasic(BinaryWriter bw, Board board)

{

bw.Write(board.Name);

bw.Write(board.ExternalSize.Width);

bw.Write(board.ExternalSize.Height);

IComponent[] comps = board.GetComponents();

Wire[] wires = board.GetAllWires();

bw.Write(comps.Length);

foreach (IComponent comp in comps)

{

bw.Write(comp);

}

bw.Write(wires.Length);

foreach (Wire wire in wires)

{

bw.Write(wire);

}

}

public static Board ReadBoard(this BinaryReader br)

{

Board topLevelBoard = ReadBoardBasic(br);

int boardCount = br.ReadInt32();

Board[] boards = new Board[boardCount];

for (int i = 0; i < boardCount; i++)

{

boards[i] = ReadBoardBasic(br);

}

topLevelBoard.SupplyInternalBoards(boards);

return topLevelBoard;

}

private static Board ReadBoardBasic(BinaryReader br)

{

Board board = new Board(br.ReadString(), new Size(br.ReadInt32(), br.ReadInt32()));

int compCount = br.ReadInt32();

for (int i = 0; i < compCount; i++)

{

br.ReadComponent(board);

}

int wireCount = br.ReadInt32();

for (int i = 0; i < wireCount; i++)

{

br.ReadWire(board);

}

return board;

}

public static Dictionary<string, Func<string, IComponent>> Constructors = new Dictionary<string, Func<string, IComponent>>();

public static Dictionary<string, string> DefaultDetails = new Dictionary<string, string>();

}

#### DefaultDictionary

class DefaultDictionary<TKey, TValue> : Dictionary<TKey, TValue>

{

protected readonly Func<TValue> BlindGenerator;

protected readonly Func<TKey, TValue> KeyBasedGenerator;

protected readonly Func<DefaultDictionary<TKey, TValue>, TValue> DictBasedGenerator;

protected readonly Func<DefaultDictionary<TKey, TValue>, TKey, TValue> DictAndKeyBasedGenerator;

protected readonly Func<TKey, bool> KeyKeepChecker;

protected readonly Func<TValue, bool> ValKeepChecker;

public DefaultDictionary()

{

ValKeepChecker = val => val.Equals(default(TValue));

}

public DefaultDictionary(Func<TValue> generator)

{

BlindGenerator = generator;

}

public DefaultDictionary(Func<TKey, TValue> generator)

{

KeyBasedGenerator = generator;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TValue> generator)

{

DictBasedGenerator = generator;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TKey, TValue> generator)

{

DictAndKeyBasedGenerator = generator;

}

public DefaultDictionary(Func<TValue> generator, Func<TValue, bool> valKeepChecker)

{

BlindGenerator = generator;

ValKeepChecker = valKeepChecker;

}

public DefaultDictionary(Func<TKey, TValue> generator, Func<TValue, bool> valKeepChecker)

{

KeyBasedGenerator = generator;

ValKeepChecker = valKeepChecker;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TValue> generator, Func<TValue, bool> valKeepChecker)

{

DictBasedGenerator = generator;

ValKeepChecker = valKeepChecker;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TKey, TValue> generator, Func<TValue, bool> valKeepChecker)

{

DictAndKeyBasedGenerator = generator;

ValKeepChecker = valKeepChecker;

}

public DefaultDictionary(Func<TValue> generator, Func<TKey, bool> keyKeepChecker)

{

BlindGenerator = generator;

KeyKeepChecker = keyKeepChecker;

}

public DefaultDictionary(Func<TKey, TValue> generator, Func<TKey, bool> keyKeepChecker)

{

KeyBasedGenerator = generator;

KeyKeepChecker = keyKeepChecker;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TValue> generator, Func<TKey, bool> keyKeepChecker)

{

DictBasedGenerator = generator;

KeyKeepChecker = keyKeepChecker;

}

public DefaultDictionary(Func<DefaultDictionary<TKey, TValue>, TKey, TValue> generator, Func<TKey, bool> keyKeepChecker)

{

DictAndKeyBasedGenerator = generator;

KeyKeepChecker = keyKeepChecker;

}

public void TrimDown()

{

HashSet<TKey> removeKeys = new HashSet<TKey>();

if (ValKeepChecker != null)

{

foreach (TKey key in Keys)

{

if (!ValKeepChecker(base[key]))

{

removeKeys.Add(key);

}

}

} else if (KeyKeepChecker != null)

{

foreach (TKey key in Keys)

{

if (!KeyKeepChecker(key))

{

removeKeys.Add(key);

}

}

}

removeKeys.Select(Remove);

}

public new TValue this[TKey key]

{

get

{

if (!ContainsKey(key))

{

if (BlindGenerator != null)

{

Add(key, BlindGenerator());

} else if (KeyBasedGenerator != null)

{

Add(key, KeyBasedGenerator(key));

} else if (DictBasedGenerator != null)

{

Add(key, DictBasedGenerator(this));

} else if (DictAndKeyBasedGenerator != null)

{

Add(key, DictAndKeyBasedGenerator(this, key));

} else

{

Add(key, default);

}

}

return base[key];

}

set

{

base[key] = value;

TrimDown();

}

}

}

#### InstanceTracker

public class InstanceTracker<T> where T : class, InstanceTracker<T>.ITrackable

{

Dictionary<uint, WeakReference<T>> dict = new Dictionary<uint, WeakReference<T>>();

public InstanceTracker()

{

}

private uint GetFirstEmptyID()

{

TrimDown();

uint id = 0;

while (dict.ContainsKey(id))

{

id++;

}

return id;

}

public void TrimDown()

{

bool loop;

do

{

loop = false;

foreach (uint id in dict.Keys)

{

if (!dict[id].TryGetTarget(out T obj))

{

dict.Remove(id);

loop = true;

break;

}

}

} while (loop);

}

private void Add(uint id, T obj)

{

obj.SetTrackingID(id);

dict.Add(id, new WeakReference<T>(obj));

}

public void Add(T obj)

{

Add(GetFirstEmptyID(), obj);

}

public T this[uint id]

{

get

{

if (dict[id].TryGetTarget(out T retObj))

{

return retObj;

}

return null;

}

set

{

TrimDown();

if (dict.ContainsKey(id))

{

dict[id].SetTarget(value);

} else

{

Add(id, value);

}

}

}

}

##### ITrackable

public interface ITrackable

{

void SetTrackingID(uint id);

uint GetTrackingID();

}

#### Pos

public readonly struct Pos

{

public readonly int X;

public readonly int Y;

public Pos(int X, int Y)

{

this.X = X;

this.Y = Y;

}

public bool Equals(Pos other)

{

return X == other.X && Y == other.Y;

}

public Pos Add(int X, int Y)

{

return new Pos(this.X + X, this.Y + Y);

}

public Pos Add(Pos pos)

{

return new Pos(X + pos.X, Y + pos.Y);

}

public override string ToString()

{

return $"({X}, {Y})";

}

public static bool operator ==(Pos pos1, Pos pos2)

{

return pos1.X == pos2.X && pos1.Y == pos2.Y;

}

public static bool operator !=(Pos pos1, Pos pos2)

{

return pos1.X != pos2.X || pos1.Y != pos2.Y;

}

public override bool Equals(object obj)

{

if (obj is Pos pos)

{

return this == pos;

}

return base.Equals(obj);

}

public override int GetHashCode()

{

return base.GetHashCode();

}

public Pos Rotate(Rotation rotation)

{

if (rotation == Rotation.CLOCKWISE)

{

return new Pos(-Y, X);

}

else if (rotation == Rotation.HALF)

{

return new Pos(-X, -Y);

}

else if (rotation == Rotation.ANTICLOCKWISE)

{

return new Pos(Y, -X);

}

else

{

return this;

}

}

public Point ToPoint()

{

return new Point(X, Y);

}

public static Pos FromPoint(Point point)

{

return new Pos(point.X, point.Y);

}

}

#### Rotation

public enum Rotation

{

ZERO = 0, CLOCKWISE = 90, HALF = 180, ANTICLOCKWISE = 270

}

#### RotationExtensions

static class RotationExtensions

{

public static Rotation AddRotation(this Rotation rot1, Rotation rot2)

{

return (Rotation)(((int)rot1 + (int)rot2) % 360);

}

}

#### Wire

public class Wire

{

public readonly Pos Pos1;

public readonly Pos Pos2;

private readonly Board board;

public bool IsPlaced { get; private set; }

public Pin Pin1

{

get

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

return board[Pos1];

}

}

public Pin Pin2

{

get

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

return board[Pos2];

}

}

public Wire(Pos pos1, Pos pos2, Board board)

{

Pos1 = pos1;

Pos2 = pos2;

this.board = board;

board.AddWire(this);

IsPlaced = true;

}

public void OnWireUpdate()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

Pin1.SetState(Pin2.GetStateForWire());

Pin2.SetState(Pin1.GetStateForWire());

}

public bool TrySplitWire(Pos pos)

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

if (Collision(pos))

{

new Wire(Pos1, pos, board);

new Wire(Pos2, pos, board);

Remove();

return true;

}

return false;

}

public void Remove()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

board.RemoveWire(this);

IsPlaced = false;

}

public override string ToString()

{

return IsPlaced ? $"[{Pos1}, {Pos2}]" : "Removed Wire";

}

public static bool operator ==(Wire wire1, Wire wire2)

{

if (wire1 is null && wire2 is null)

{

return true;

} else if (wire1 is null || wire2 is null)

{

return false;

}

if (wire1.board != wire2.board)

{

return false;

}

return (wire1.Pos1 == wire2.Pos1 && wire1.Pos2 == wire2.Pos2) || (wire1.Pos1 == wire2.Pos2 && wire1.Pos2 == wire2.Pos1);

}

public static bool operator !=(Wire wire1, Wire wire2)

{

if (wire1 is null && wire2 is null)

{

return false;

} else if (wire1 is null || wire2 is null)

{

return true;

}

if (wire1.board != wire2.board)

{

return true;

}

return (wire1.Pos1 != wire2.Pos1 || wire1.Pos2 != wire2.Pos2) && (wire1.Pos1 != wire2.Pos2 || wire1.Pos2 != wire2.Pos1);

}

public override bool Equals(object obj)

{

if (obj is Wire wire)

{

return this == wire;

}

return base.Equals(obj);

}

public override int GetHashCode()

{

return Pos1.GetHashCode() + Pos2.GetHashCode();

}

public Rectangle Bounds()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

return Rectangle.FromLTRB(

Math.Min(Pos1.X, Pos2.X),

Math.Min(Pos1.Y, Pos2.Y),

Math.Max(Pos1.X, Pos2.X),

Math.Max(Pos1.Y, Pos2.Y));

}

public RectangleF InflatedBounds()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

RectangleF bounds = Bounds();

bounds.Inflate(0.25F, 0.25F);

return bounds;

}

public bool IsVert()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

return Pos1.X == Pos2.X;

}

public bool IsHori()

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

return Pos1.Y == Pos2.Y;

}

public bool Collision(Pos pos)

{

if (!IsPlaced)

{

throw new ObjectDisposedException("This wire has already been removed.");

}

Rectangle bounds = Bounds();

if (IsVert())

{

return bounds.Top < pos.Y && pos.Y < bounds.Bottom && pos.X == Pos1.X;

}

if (IsHori())

{

return bounds.Left < pos.X && pos.X < bounds.Right && pos.Y == Pos1.Y;

}

return bounds.Contains(new Point(pos.X, pos.Y));

}

}

#### ColourScheme

public struct ColourScheme

{

public Color Background, ComponentBackground, ComponentEdge, Wire, WireFloating, WireLow, WirePulledLow, WireHigh, WirePulledHigh, WireIllegal, Grid, Selection;

public Color GetWireColour(Pin.State state)

{

switch (state)

{

case Pin.State.FLOATING: return WireFloating;

case Pin.State.LOW: return WireLow;

case Pin.State.PULLEDLOW: return WirePulledLow;

case Pin.State.HIGH: return WireHigh;

case Pin.State.PULLEDHIGH: return WirePulledHigh;

case Pin.State.ILLEGAL: return WireIllegal;

default: return Wire;

}

}

}

#### IComponent

public interface IComponent

{

void Place(Pos pos, Board board);

void Place(Pos pos, Rotation rotation, Board board);

void Remove();

bool IsPlaced();

void Tick();

void ResetToDefault();

Pos[] GetAllPinOffsets();

Pos[] GetAllPinPositions();

Pos[] GetAllUniquePinPositions();

Pos GetComponentPos();

Rotation GetComponentRotation();

Matrix GetRenderMatrix();

Board GetComponentBoard();

string GetComponentID();

string GetComponentDetails();

RectangleF GetComponentBounds();

RectangleF GetOffsetComponentBounds();

void Render(Graphics graphics, bool simulating, ColourScheme colourScheme);

void RenderMainShape(Graphics graphics, bool simulating, ColourScheme colourScheme);

}

#### IInteractibleComponent

public interface IInteractibleComponent : IComponent

{

void Interact();

}

#### IWireComponent

public interface IWireComponent : IComponent

{

bool TickAgain();

}

#### IBoardInterfaceComponent

public interface IBoardInterfaceComponent : IWireComponent

{

string GetComponentName();

void SetComponentName(string compName);

void SetExternalPin(Pin pin);

void RemoveExternalPin();

Board.InterfaceLocation GetInterfaceLocation();

void SetInterfaceLocation(Board.InterfaceLocation location);

}

#### IBoardInputComponent

public interface IBoardInputComponent : IBoardInterfaceComponent { }

#### IBoardOutputComponent

public interface IBoardOutputComponent : IBoardInterfaceComponent { }

#### IGraphicalComponent

public interface IGraphicalComponent : IComponent

{

bool HasGraphics();

void RenderGraphicalElement(Graphics graphics, bool simulating, ColourScheme colourScheme);

RectangleF GetGraphicalElementBounds();

Point? GetGraphicalElementLocation();

void SetGraphicalElementLocation(Point? location);

float GetGraphicalElementScale();

void SetGraphicalElementScale(float scale);

}

#### IBoardContainerComponent

public interface IBoardContainerComponent : IGraphicalComponent

{

Rectangle GetShape();

void ResetShape();

string GetInternalBoardName();

Board GetInternalBoard();

void ProvideInternalBoard(Board board);

void PromiseDetails(Action<IBoardContainerComponent> detailProvider);

}

#### ComponentExtensions

public static class ComponentExtensions

{

private static RectangleF Scale(RectangleF rect, float scale)

{

rect.X \*= scale;

rect.Y \*= scale;

rect.Width \*= scale;

rect.Height \*= scale;

return rect;

}

private static RectangleF? Offset(RectangleF rect, Point? pos)

{

if (pos.HasValue)

{

rect.Offset(pos.Value);

return rect;

}

return null;

}

public static RectangleF? GetOffsetGraphicalElementBounds<T>(this T comp) where T : IGraphicalComponent

{

return Offset(comp.GetGraphicalElementBounds(), comp.GetGraphicalElementLocation());

}

public static RectangleF GetScaledGraphicalElementBounds<T>(this T comp) where T : IGraphicalComponent

{

return Scale(comp.GetGraphicalElementBounds(), comp.GetGraphicalElementScale());

}

public static RectangleF? GetOffsetScaledGraphicalElementBounds<T>(this T comp) where T : IGraphicalComponent

{

return Offset(Scale(comp.GetGraphicalElementBounds(), comp.GetGraphicalElementScale()), comp.GetGraphicalElementLocation());

}

public static IComponent Copy<T>(this T comp) where T : IComponent

{

IComponent copy = ReadWriteImplementation.Constructors[comp.GetComponentID()](comp.GetComponentDetails());

if (comp is IGraphicalComponent graphicalComp && copy is IGraphicalComponent graphicalCopy)

{

graphicalCopy.SetGraphicalElementScale(graphicalComp.GetGraphicalElementScale());

graphicalCopy.SetGraphicalElementLocation(graphicalComp.GetGraphicalElementLocation());

}

if (comp is IBoardInterfaceComponent interfaceComp && copy is IBoardInterfaceComponent interfaceCopy)

{

interfaceCopy.SetInterfaceLocation(interfaceComp.GetInterfaceLocation());

}

if (comp is IBoardContainerComponent contComp && copy is IBoardContainerComponent contCopy)

{

Board intBoard = contComp.GetInternalBoard();

if (!(intBoard is null))

{

contCopy.ProvideInternalBoard(intBoard.Copy());

}

}

return copy;

}

}

#### Pin

public class Pin

{

private State CurrentState, OriginalState;

public State GetStateForComponent()

{

return OriginalState.Pulled();

}

public State GetStateForWireComponent()

{

return CurrentState;

}

public State GetStateForWire()

{

return CurrentState;

}

public State GetStateForDisplay()

{

return CurrentState;

}

public void SetState(State state)

{

CurrentState = CurrentState.WireJoin(state);

}

public void ResetToFloating()

{

CurrentState = State.FLOATING;

OriginalState = State.FLOATING;

}

public void SetupForTick()

{

OriginalState = CurrentState;

CurrentState = Pin.State.FLOATING;

}

}

##### State

public enum State

{

FLOATING, LOW, PULLEDLOW, HIGH, PULLEDHIGH, ILLEGAL

}

#### StateExtensions

static class StateExtensions

{

private static Dictionary<Pin.State, Pin.State> NotOpTable = new Dictionary<Pin.State, Pin.State>

{

{ Pin.State.FLOATING, Pin.State.FLOATING },

{ Pin.State.LOW, Pin.State.HIGH },

{ Pin.State.HIGH, Pin.State.LOW },

{ Pin.State.ILLEGAL, Pin.State.ILLEGAL }

};

private static Dictionary<Pin.State, Pin.State> PullTable = new Dictionary<Pin.State, Pin.State>

{

{ Pin.State.FLOATING, Pin.State.FLOATING },

{ Pin.State.LOW, Pin.State.LOW },

{ Pin.State.PULLEDLOW, Pin.State.LOW },

{ Pin.State.HIGH, Pin.State.HIGH },

{ Pin.State.PULLEDHIGH, Pin.State.HIGH },

{ Pin.State.ILLEGAL, Pin.State.ILLEGAL }

};

private static Dictionary<(Pin.State state1, Pin.State state2), Pin.State> GenericOpTable = new Dictionary<(Pin.State state1, Pin.State state2), Pin.State>

{

{ (Pin.State.FLOATING, Pin.State.FLOATING), Pin.State.FLOATING },

{ (Pin.State.FLOATING, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.FLOATING, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.FLOATING, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.ILLEGAL, Pin.State.LOW), Pin.State.ILLEGAL },

{ (Pin.State.ILLEGAL, Pin.State.HIGH), Pin.State.ILLEGAL },

{ (Pin.State.ILLEGAL, Pin.State.ILLEGAL), Pin.State.ILLEGAL }

};

private static BinOpTable AndOpTable = new BinOpTable(new Dictionary<(Pin.State state1, Pin.State state2), Pin.State>

{

{ (Pin.State.LOW, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.HIGH), Pin.State.LOW },

{ (Pin.State.HIGH, Pin.State.HIGH), Pin.State.HIGH }

}.Concat(GenericOpTable));

private static BinOpTable OrOpTable = new BinOpTable(new Dictionary<(Pin.State state1, Pin.State state2), Pin.State>

{

{ (Pin.State.LOW, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.HIGH, Pin.State.HIGH), Pin.State.HIGH }

}.Concat(GenericOpTable));

private static BinOpTable XorOpTable = new BinOpTable(new Dictionary<(Pin.State state1, Pin.State state2), Pin.State>

{

{ (Pin.State.LOW, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.HIGH, Pin.State.HIGH), Pin.State.LOW }

}.Concat(GenericOpTable));

private static BinOpTable WireJoinTable = new BinOpTable

{

{ (Pin.State.FLOATING, Pin.State.FLOATING), Pin.State.FLOATING },

{ (Pin.State.FLOATING, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.FLOATING, Pin.State.PULLEDLOW), Pin.State.PULLEDLOW },

{ (Pin.State.FLOATING, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.FLOATING, Pin.State.PULLEDHIGH), Pin.State.PULLEDHIGH },

{ (Pin.State.FLOATING, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.LOW, Pin.State.LOW), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.PULLEDLOW), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.HIGH), Pin.State.ILLEGAL },

{ (Pin.State.LOW, Pin.State.PULLEDHIGH), Pin.State.LOW },

{ (Pin.State.LOW, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.PULLEDLOW, Pin.State.PULLEDLOW), Pin.State.PULLEDLOW },

{ (Pin.State.PULLEDLOW, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.PULLEDLOW, Pin.State.PULLEDHIGH), Pin.State.ILLEGAL },

{ (Pin.State.PULLEDLOW, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.HIGH, Pin.State.HIGH), Pin.State.HIGH },

{ (Pin.State.HIGH, Pin.State.PULLEDHIGH), Pin.State.HIGH },

{ (Pin.State.HIGH, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.PULLEDHIGH, Pin.State.PULLEDHIGH), Pin.State.PULLEDHIGH },

{ (Pin.State.PULLEDHIGH, Pin.State.ILLEGAL), Pin.State.ILLEGAL },

{ (Pin.State.ILLEGAL, Pin.State.ILLEGAL), Pin.State.ILLEGAL }

};

public static Pin.State WireJoin(this Pin.State state1, Pin.State state2)

{

return WireJoinTable[state1, state2];

}

public static Pin.State Not(this Pin.State state)

{

return NotOpTable[state];

}

public static Pin.State And(this Pin.State state1, Pin.State state2)

{

return AndOpTable[state1, state2];

}

public static Pin.State Or(this Pin.State state1, Pin.State state2)

{

return OrOpTable[state1, state2];

}

public static Pin.State Xor(this Pin.State state1, Pin.State state2)

{

return XorOpTable[state1, state2];

}

public static Pin.State Pulled(this Pin.State state)

{

return PullTable[state];

}

}

##### BinOpTable

private class BinOpTable : Dictionary<(Pin.State state1, Pin.State state2), Pin.State>

{

private static (Pin.State state1, Pin.State state2) simplifyStates((Pin.State state1, Pin.State state2) states)

{

if (states.state1 > states.state2)

{

return (states.state2, states.state1);

}

return states;

}

public new Pin.State this[(Pin.State state1, Pin.State state2) states]

{

get => base[simplifyStates(states)];

set => base[simplifyStates(states)] = value;

}

public Pin.State this[Pin.State state1, Pin.State state2]

{

get => base[simplifyStates((state1, state2))];

set => base[simplifyStates((state1, state2))] = value;

}

public BinOpTable(IEnumerable<KeyValuePair<(Pin.State state1, Pin.State state2), Pin.State>> kvps) : base(kvps.ToDictionary(kvp => simplifyStates(kvp.Key), kvp => kvp.Value)) { }

public BinOpTable(IDictionary<(Pin.State state1, Pin.State state2), Pin.State> dict) : base(dict.ToDictionary(kvp => simplifyStates(kvp.Key), kvp => kvp.Value)) { }

public BinOpTable() : base() { }

public new void Add((Pin.State state1, Pin.State state2) key, Pin.State val)

{

base.Add(simplifyStates(key), val);

}

}

#### SideEnumExtensions

public static class SideEnumExtensions

{

public static bool IsLeftRight(this Board.InterfaceLocation.SideEnum side)

{

return ((byte)side & 0b010) != 0;

}

public static bool IsBottomRight(this Board.InterfaceLocation.SideEnum side)

{

return ((byte)side & 0b001) != 0;

}

public static bool IsTop(this Board.InterfaceLocation.SideEnum side)

{

return !side.IsLeftRight() && !side.IsBottomRight();

}

public static bool IsBottom(this Board.InterfaceLocation.SideEnum side)

{

return !side.IsLeftRight() && side.IsBottomRight();

}

public static bool IsLeft(this Board.InterfaceLocation.SideEnum side)

{

return side.IsLeftRight() && !side.IsBottomRight();

}

public static bool IsRight(this Board.InterfaceLocation.SideEnum side)

{

return side.IsLeftRight() && side.IsBottomRight();

}

public static Board.InterfaceLocation.SideEnum ToggleLeftRight(this Board.InterfaceLocation.SideEnum side)

{

return (Board.InterfaceLocation.SideEnum)((byte)side ^ 0b10);

}

public static Board.InterfaceLocation.SideEnum ToggleBottomRight(this Board.InterfaceLocation.SideEnum side)

{

return (Board.InterfaceLocation.SideEnum)((byte)side ^ 0b01);

}

public static Board.InterfaceLocation.SideEnum ToggleLeftRightIf(this Board.InterfaceLocation.SideEnum side, bool cond)

{

return cond ? side.ToggleLeftRight() : side;

}

public static Board.InterfaceLocation.SideEnum ToggleBottomRightIf(this Board.InterfaceLocation.SideEnum side, bool cond)

{

return cond ? side.ToggleBottomRight() : side;

}

public static Board.InterfaceLocation.SideEnum WithLeftRightAs(this Board.InterfaceLocation.SideEnum side, bool cond)

{

return (Board.InterfaceLocation.SideEnum)((cond ? 0b10 : 0b00) + ((byte)side & 0b01));

}

public static Board.InterfaceLocation.SideEnum WithBottomRightAs(this Board.InterfaceLocation.SideEnum side, bool cond)

{

return (Board.InterfaceLocation.SideEnum)((cond ? 0b01 : 0b00) + ((byte)side & 0b10));

}

}

#### Board

public class Board : InstanceTracker<Board>.ITrackable

{

public static InstanceTracker<Board> AllBoards = new InstanceTracker<Board>();

private uint trackingID;

public void SetTrackingID(uint id)

{

trackingID = id;

}

public uint GetTrackingID()

{

return trackingID;

}

private DefaultDictionary<Pos, Pin> Pins;

private HashSet<Wire> Wires = new HashSet<Wire>();

private HashSet<Wire> AllWires = new HashSet<Wire>();

private DefaultDictionary<Pos, int> ConnectionsToPin = new DefaultDictionary<Pos, int>();

private HashSet<IComponent> Components = new HashSet<IComponent>();

private HashSet<IWireComponent> WireComponents = new HashSet<IWireComponent>();

private HashSet<IWireComponent> AllWireComponents = new HashSet<IWireComponent>();

private HashSet<IComponent> NonWireComponents = new HashSet<IComponent>();

private HashSet<IBoardInterfaceComponent> InterfaceComponents = new HashSet<IBoardInterfaceComponent>();

private HashSet<IBoardInputComponent> InputComponents = new HashSet<IBoardInputComponent>();

private HashSet<IBoardOutputComponent> OutputComponents = new HashSet<IBoardOutputComponent>();

private List<IGraphicalComponent> GraphicalComponents = new List<IGraphicalComponent>();

private HashSet<IBoardContainerComponent> ContainerComponents = new HashSet<IBoardContainerComponent>();

private Board owner;

public void SetOwnerBoard(Board owner)

{

this.owner = owner;

}

public void ResetOwnerBoard()

{

owner = null;

}

public Board GetTopLevelBoard()

{

if (owner is null)

{

return this;

}

return owner.GetTopLevelBoard();

}

private Size? externalSize;

public Size ExternalSize

{

get

{

if (!externalSize.HasValue)

{

int bidirCount = InputComponents.Count() + OutputComponents.Count() - InterfaceComponents.Count();

int vertLimit = Math.Max(Math.Max(InputComponents.Count(), OutputComponents.Count()) - bidirCount, 1),

horiLimit = Math.Max(Math.Max(bidirCount, vertLimit / 2), 1);

externalSize = new Size(vertLimit \* 2, horiLimit \* 2);

SizeChanged?.Invoke();

}

return externalSize.Value;

}

set

{

externalSize = value;

SizeChanged?.Invoke();

}

}

public event Action SizeChanged;

public string Name;

public Board(string name, Size? externalSize = null)

{

Name = name;

if (externalSize.HasValue)

{

ExternalSize = externalSize.Value;

}

Pins = new DefaultDictionary<Pos, Pin>(() => new Pin(), (pos) => ConnectionsToPin[pos] != 0);

AllBoards.Add(this);

}

public void AddWire(Wire wire)

{

Wires.Add(wire);

AllWires.Add(wire);

ConnectionsToPin[wire.Pos1]++;

ConnectionsToPin[wire.Pos2]++;

}

public void RemoveWire(Wire wire)

{

Wires.Remove(wire);

AllWires.Remove(wire);

ConnectionsToPin[wire.Pos1]--;

ConnectionsToPin[wire.Pos2]--;

}

public Wire[] GetAllWires()

{

return Wires.ToArray();

}

public IComponent[] GetComponents()

{

return Components.ToArray();

}

public IWireComponent[] GetWireComponents()

{

return WireComponents.ToArray();

}

public IComponent[] GetNonWireComponents()

{

return NonWireComponents.ToArray();

}

public IBoardInterfaceComponent[] GetInterfaceComponents()

{

return InterfaceComponents.ToArray();

}

public IBoardInterfaceComponent GetInterfaceComponent(string name)

{

foreach (IBoardInterfaceComponent comp in InterfaceComponents)

{

if (comp.GetComponentName() == name)

{

return comp;

}

}

return null;

}

public IGraphicalComponent[] GetGraphicalComponents()

{

return GraphicalComponents.ToArray();

}

public IGraphicalComponent GetGraphicalComponent(int index)

{

return GraphicalComponents[index];

}

public IBoardInputComponent[] GetInputComponents()

{

return InputComponents.ToArray();

}

public IBoardInputComponent GetInputComponent(string name)

{

foreach (IBoardInputComponent comp in InputComponents)

{

if (comp.GetComponentName() == name)

{

return comp;

}

}

return null;

}

public IBoardOutputComponent[] GetOutputComponents()

{

return OutputComponents.ToArray();

}

public IBoardOutputComponent GetOutputComponent(string name)

{

foreach (IBoardOutputComponent comp in OutputComponents)

{

if (comp.GetComponentName() == name)

{

return comp;

}

}

return null;

}

public IBoardContainerComponent[] GetContainerComponents()

{

return ContainerComponents.ToArray();

}

public void SupplyInternalBoards(Board[] internalBoards)

{

Dictionary<string, Board> internalBoardsFromName = internalBoards.ToDictionary(board => board.Name);

Queue<Board> unsuppliedBoards = new Queue<Board>(); // Code Reference: Queue usage

unsuppliedBoards.Enqueue(this);

string boardName;

Board supplyingBoard, providedBoard;

while (unsuppliedBoards.Count > 0)

{

supplyingBoard = unsuppliedBoards.Dequeue();

foreach (IBoardContainerComponent contComp in supplyingBoard.ContainerComponents) // for every ContainerComponent, there is a new board needed

{

boardName = contComp.GetInternalBoardName(); // so we find out what type it is

if (internalBoardsFromName.ContainsKey(boardName)) // and if we have that type

{

providedBoard = internalBoardsFromName[boardName].Copy(supplyBoards: false); // we create one

unsuppliedBoards.Enqueue(providedBoard); // add it to the queue

contComp.ProvideInternalBoard(providedBoard); // and also give a reference to the ContainerComponent that wants it

} else // and if we don't have that type

{

throw new InvalidDataException($"{supplyingBoard.Name} board contains {boardName} board, which has not been supplied"); // throw an exception

}

}

}

}

public Board[] GetBoardList()

{

List<Board> checkedBoardList = new List<Board>(), uncheckedBoardList = new List<Board> { this };

Func<string, bool> notSeen = boardName => !checkedBoardList.Select(checkedBoard => checkedBoard.Name).Concat(uncheckedBoardList.Select(uncheckedBoard => uncheckedBoard.Name)).Contains(boardName);

while (uncheckedBoardList.Count > 0)

{

foreach (IBoardContainerComponent contComp in uncheckedBoardList[0].GetContainerComponents())

{

if (notSeen(contComp.GetInternalBoardName()))

{

Board intBoard = contComp.GetInternalBoard();

if (!(intBoard is null))

{

uncheckedBoardList.Add(intBoard);

}

}

}

checkedBoardList.Add(uncheckedBoardList[0]);

uncheckedBoardList.RemoveAt(0);

}

return checkedBoardList.ToArray();

}

public void TickSetup()

{

ClearUnusedPins();

foreach (Pin pin in Pins.Values)

{

pin.SetupForTick();

}

foreach (IBoardContainerComponent boardContainerComp in ContainerComponents)

{

boardContainerComp.GetInternalBoard().TickSetup();

}

}

public void TickComponents()

{

foreach (IComponent nonWireComp in NonWireComponents)

{

nonWireComp.Tick();

}

}

private bool SubTickWire(Wire wire)

{

Pin pin1 = wire.Pin1, pin2 = wire.Pin2;

Pin.State state1 = pin1.GetStateForWire(), state2 = pin2.GetStateForWire();

if (state1 != state2)

{

pin1.SetState(state2);

pin2.SetState(state1);

return true;

}

return false;

}

public bool SubTickWires()

{

bool retVal = false;

foreach (Wire wire in AllWires)

{

retVal |= SubTickWire(wire);

}

foreach (IWireComponent wireComp in AllWireComponents)

{

wireComp.Tick();

retVal |= wireComp.TickAgain();

}

return retVal;

}

public void TickWires()

{

while (SubTickWires()) { }

}

public void Tick()

{

TickSetup();

TickComponents();

TickWires();

}

public Pin this[Pos pos] => Pins[pos];

private string GuaranteeUniqueName(string current, string[] existing)

{

string baseName = current.TrimEnd("0123456789".ToArray());

string baseNumberString = current.Substring(baseName.Length);

int baseNumber = 0;

int.TryParse(baseNumberString, out baseNumber);

while (existing.Contains(current))

{

baseNumber++;

current = baseName + baseNumber.ToString();

}

return current;

}

private void CountComponentPins(IComponent comp)

{

foreach (Pos pos in comp.GetAllUniquePinPositions())

{

ConnectionsToPin[pos]++;

}

}

private void AddContainedBoardWires(IBoardContainerComponent comp)

{

AllWires.UnionWith(comp.GetInternalBoard().AllWires);

AllWireComponents.UnionWith(comp.GetInternalBoard().AllWireComponents);

if (!(owner is null))

{

owner.AddContainedBoardWires(comp);

}

}

internal void AddComponent(IComponent comp)

{

foreach (IComponent otherComp in Components)

{

if (otherComp.GetOffsetComponentBounds().IntersectsWith(comp.GetOffsetComponentBounds()))

{

throw new PlacementException("cant place on another component. this error shouldn't be in the final product");

}

}

Components.Add(comp);

if (comp is IGraphicalComponent graphicalComp)

{

GraphicalComponents.Add(graphicalComp);

}

if (comp is IWireComponent wireComp)

{

WireComponents.Add(wireComp);

AllWireComponents.Add(wireComp);

} else

{

NonWireComponents.Add(comp);

}

if (comp is IBoardContainerComponent contComp)

{

ContainerComponents.Add(contComp);

contComp.PromiseDetails(CountComponentPins);

contComp.PromiseDetails(AddContainedBoardWires);

} else

{

CountComponentPins(comp);

}

if (comp is IBoardInterfaceComponent interfaceComp)

{

InterfaceLocation interfaceLoc = interfaceComp.GetInterfaceLocation();

if (interfaceLoc.Distance == 0)

{

int[] existingLocs = GetInterfaceComponents().Where(thisComp => thisComp.GetInterfaceLocation().Side == interfaceLoc.Side).Select(thisComp => thisComp.GetInterfaceLocation().Distance).ToArray();

for (int newDist = 1; true; newDist += 2)

{

if (!existingLocs.Contains(newDist))

{

interfaceLoc.Distance = newDist;

interfaceComp.SetInterfaceLocation(interfaceLoc);

break;

}

}

}

interfaceComp.SetComponentName(GuaranteeUniqueName(interfaceComp.GetComponentName(),

InputComponents.Select(thisComp => thisComp.GetComponentName()).Concat(OutputComponents.Select(thisComp => thisComp.GetComponentName())).ToArray()));

InterfaceComponents.Add(interfaceComp);

if (comp is IBoardInputComponent inpComp)

{

InputComponents.Add(inpComp);

}

if (comp is IBoardOutputComponent outpComp)

{

OutputComponents.Add(outpComp);

}

bool isSide = interfaceLoc.Side.IsLeftRight();

while (interfaceLoc.Distance >= (isSide ? ExternalSize.Height : ExternalSize.Width))

{

ExternalSize = new Size(ExternalSize.Width + (isSide ? 0 : 1), ExternalSize.Height + (isSide ? 1 : 0));

}

}

}

internal void RemoveComponent(IComponent comp)

{

comp.GetAllUniquePinPositions().Select(pos => ConnectionsToPin[pos]--);

Components.Remove(comp);

if (comp is IGraphicalComponent graphicalComp)

{

GraphicalComponents.Remove(graphicalComp);

}

if (comp is IWireComponent wireComp)

{

WireComponents.Remove(wireComp);

AllWireComponents.Remove(wireComp);

} else

{

NonWireComponents.Remove(comp);

}

if (comp is IBoardInterfaceComponent interfaceComp)

{

InterfaceComponents.Remove(interfaceComp);

}

if (comp is IBoardInputComponent inpComp)

{

InputComponents.Remove(inpComp);

}

if (comp is IBoardOutputComponent outpComp)

{

OutputComponents.Remove(outpComp);

}

if (comp is IBoardContainerComponent contComp)

{

ContainerComponents.Remove(contComp);

AllWires.ExceptWith(contComp.GetInternalBoard().AllWires);

AllWireComponents.ExceptWith(contComp.GetInternalBoard().AllWireComponents);

}

}

protected void ClearUnusedPins()

{

HashSet<Pos> keepPinPositions = new HashSet<Pos>();

foreach (IComponent comp in Components)

{

keepPinPositions.UnionWith(comp.GetAllPinPositions());

}

foreach (Wire wire in Wires)

{

keepPinPositions.Add(wire.Pos1);

keepPinPositions.Add(wire.Pos2);

}

HashSet<Pos> removePinPositions = new HashSet<Pos>();

removePinPositions.UnionWith(Pins.Keys);

removePinPositions.ExceptWith(keepPinPositions);

foreach (Pos pinPos in removePinPositions)

{

Pins.Remove(pinPos);

}

}

public void Render(Graphics graphics, bool simulating, Rectangle bounds, ColourScheme colourScheme)

{

Pen gridPen = new Pen(colourScheme.Grid, 0.005F);

for (int x = bounds.Left; x <= bounds.Right; x++)

{

graphics.DrawLine(gridPen, x, bounds.Top, x, bounds.Bottom);

}

for (int y = bounds.Top; y <= bounds.Bottom; y++)

{

graphics.DrawLine(gridPen, bounds.Left, y, bounds.Right, y);

}

Pin pin;

foreach (Pos pinPos in Pins.Keys)

{

if (bounds.Contains(pinPos.X, pinPos.Y))

{

pin = Pins[pinPos];

if (ConnectionsToPin[pinPos] == 0)

{

continue;

}

if (ConnectionsToPin[pinPos] != 2)

{

graphics.FillEllipse(new SolidBrush(colourScheme.GetWireColour(pin.GetStateForDisplay())), pinPos.X - 0.05F, pinPos.Y - 0.05F, 0.1F, 0.1F);

}

if (simulating)

{

graphics.DrawString(pin.GetStateForDisplay().ToString(), new Font("arial", 0.1F), Brushes.Black, pinPos.X, pinPos.Y);

}

}

}

graphics.DrawEllipse(new Pen(colourScheme.Grid, 0.01F), -0.1F, -0.1F, 0.2F, 0.2F);

Matrix matrix;

foreach (IComponent comp in Components)

{

if (comp.GetOffsetComponentBounds().IntersectsWith(bounds))

{

using (new TransformRestorer(graphics))

{

matrix = comp.GetRenderMatrix();

graphics.MultiplyTransform(matrix);

comp.Render(graphics, simulating, colourScheme);

if (comp is IGraphicalComponent graphicalComp)

{

graphicalComp.RenderGraphicalElement(graphics, simulating, colourScheme);

}

RectangleF compBounds = comp.GetComponentBounds();

}

}

}

foreach (Wire wire in Wires)

{

graphics.DrawLine(new Pen(simulating ? colourScheme.GetWireColour(wire.Pin1.GetStateForDisplay()) : colourScheme.Wire, 0.01F), new Point(wire.Pos1.X, wire.Pos1.Y), new Point(wire.Pos2.X, wire.Pos2.Y));

}

}

public bool CheckAllowed(RectangleF bounds)

{

foreach (IComponent comp in Components)

{

if (comp.GetOffsetComponentBounds().IntersectsWith(bounds))

{

return false;

}

}

return true;

}

public void ResetToFloating()

{

foreach (Pin pin in Pins.Values)

{

pin.ResetToFloating();

}

}

public void ResetForSimulation()

{

ResetToFloating();

foreach (IComponent comp in Components)

{

comp.ResetToDefault();

}

}

private bool TrySimplifyWirePair(Wire wire1, Wire wire2)

{

Func<Pos, int> getCompOrd, getOtherOrd;

Func<int, int, Pos> backToPos;

int[] compOrd;

int min, max, otherOrd;

if (wire1.TrySplitWire(wire2.Pos1) ||

wire1.TrySplitWire(wire2.Pos2) ||

wire2.TrySplitWire(wire1.Pos1) ||

wire2.TrySplitWire(wire1.Pos2))

{

return true;

}

if ((ConnectionsToPin[wire1.Pos1] == 2 && (wire1.Pos1 == wire2.Pos1 || wire1.Pos1 == wire2.Pos2)) ||

(ConnectionsToPin[wire1.Pos2] == 2 && (wire1.Pos2 == wire2.Pos1 || wire1.Pos2 == wire2.Pos2)))

{

if (wire1.IsHori() && wire2.IsHori())

{

getCompOrd = pos => pos.Y;

getOtherOrd = pos => pos.X;

backToPos = (comp, other) => new Pos(other, comp);

}

else if (wire1.IsHori() && wire2.IsHori())

{

getCompOrd = pos => pos.X;

getOtherOrd = pos => pos.Y;

backToPos = (comp, other) => new Pos(comp, other);

}

else

{

return false;

}

compOrd = new int[] { getCompOrd(wire1.Pos1), getCompOrd(wire1.Pos2), getCompOrd(wire2.Pos1), getCompOrd(wire2.Pos2) };

otherOrd = getOtherOrd(wire1.Pos1);

min = compOrd.Aggregate(Math.Min);

max = compOrd.Aggregate(Math.Max);

new Wire(backToPos(min, otherOrd), backToPos(max, otherOrd), this);

wire1.Remove();

wire2.Remove();

return true;

}

return false;

}

public void SimplifyWires()

{

HashSet<Wire> removeWires = new HashSet<Wire>();

Wire[] wires;

HashSet<Wire> addWires = new HashSet<Wire>();

bool changeMade;

do

{

changeMade = false;

foreach (Wire wire in Wires)

{

if (wire.Pos1 == wire.Pos2)

{

removeWires.Add(wire);

}

}

foreach (Wire removeWire in removeWires)

{

removeWire.Remove();

}

removeWires.Clear();

wires = new Wire[Wires.Count];

Wires.CopyTo(wires);

for (int i = 0; i < wires.Length - 1; i++)

{

for (int j = i + 1; j < wires.Length; j++)

{

changeMade = TrySimplifyWirePair(wires[i], wires[j]);

if (changeMade)

{

break;

}

}

if (changeMade)

{

break;

}

}

} while (changeMade);

foreach (IBoardContainerComponent boardContainerComp in ContainerComponents)

{

boardContainerComp.GetInternalBoard().SimplifyWires();

}

}

public Dictionary<Pos, Pin.State> GetStateToCheckForChanges()

{

return Pins.ToDictionary(kvp => kvp.Key, kvp => kvp.Value.GetStateForDisplay());

}

public override string ToString()

{

return $"{Name} ({trackingID})";

}

private static string GetFilename(string boardname)

{

return $"Boards/{boardname}.brd";

}

public void Save(string filename)

{

using (FileStream file = File.Open(filename, FileMode.Create))

{

using (BinaryWriter bw = new BinaryWriter(file))

{

bw.Write(this);

}

}

}

public static Board Load(string filename)

{

using (FileStream file = File.Open(filename, FileMode.Open))

{

using (BinaryReader br = new BinaryReader(file))

{

return br.ReadBoard();

}

}

}

private Board CopySingle(string copyName = null)

{

Board copy = new Board(copyName ?? Name, new Size(ExternalSize.Width, ExternalSize.Height));

foreach (IComponent comp in Components)

{

comp.Copy().Place(comp.GetComponentPos(), comp.GetComponentRotation(), copy);

}

foreach (Wire wire in Wires)

{

new Wire(wire.Pos1, wire.Pos2, copy);

}

return copy;

}

public Board Copy(string copyName = null, bool supplyBoards = true)

{

Board copy = CopySingle(copyName);

if (supplyBoards)

{

Board[] copiedBoards = GetBoardList();

for (int i = 0; i < copiedBoards.Length; i++)

{

copiedBoards[i] = copiedBoards[i].CopySingle();

}

copy.SupplyInternalBoards(copiedBoards);

}

return copy;

}

public override bool Equals(object obj)

{

if (obj is Board board)

{

return this == board;

}

return false;

}

public override int GetHashCode()

{

return GetTrackingID().GetHashCode();

}

public static bool operator ==(Board board1, Board board2)

{

if (board1 is null) { if (board2 is null) { return true; } else { return false; } } else if (board2 is null) { return false; }

return board1.GetTrackingID() == board2.GetTrackingID();

}

public static bool operator !=(Board board1, Board board2)

{

if (board1 is null) { if (board2 is null) { return false; } else { return true; } } else if (board2 is null) { return true; }

return board1.GetTrackingID() != board2.GetTrackingID();

}

}

##### InterfaceLocation

public struct InterfaceLocation

{

public SideEnum Side;

public int Distance;

public InterfaceLocation(SideEnum side, int distance)

{

Side = side;

Distance = distance;

}

public static bool operator ==(InterfaceLocation loc1, InterfaceLocation loc2)

{

return loc1.Side == loc2.Side && loc1.Distance == loc2.Distance;

}

public static bool operator !=(InterfaceLocation loc1, InterfaceLocation loc2)

{

return loc1.Side != loc2.Side || loc1.Distance != loc2.Distance;

}

public override bool Equals(object other)

{

if (other is InterfaceLocation otherLoc)

{

return this == otherLoc;

}

return false;

}

public override int GetHashCode()

{

return (int)Side ^ Distance;

}

public override string ToString()

{

return $"({Side},{Distance})";

}

}

###### SideEnum

public enum SideEnum : byte {

Top = 0b00,

Bottom = 0b01,

Left = 0b10,

Right = 0b11,

}

### Components