Programmering B

EKSAMENS OPGAVE

Elev: Anders Kornerup Kok Larsen
Fag: Programmering B
Klasse: 20HTXCR
Skole: Aarhus Gymnasium
Vejleder: Mirsad Kadribasic
Dato for aflevering: 2. Maj 2022

Kildekode: https://github.com/DenseOriginal/Computer-Basics
Selve programmet: https://computer-basics.netlify.app/

INDHOLDSFORTEGNELSE

ABSTRACT

Denne opgave er lavet i sammenhæng med programmerings B eksamen i 2.G. Programmet er lavet ved hjælp af p5.js, og skrevet i Typescript.

Programmet simulerer et binært kredsløb som brugeren designer. Man kan se hvordan forskellige gates interagere med hinanden. Man kan også kombinere en masse gates og få en helt ny gate.

Programmet er udviklet til undervisning. Det kan være svært at forestille sig hvordan en computer fungerer, men fordi det her er interaktivt og relativt simpelt, så giver det en hel ny måde at lære på.

På billedet ses et eksempel på hvordan et kredsløb kan laves.

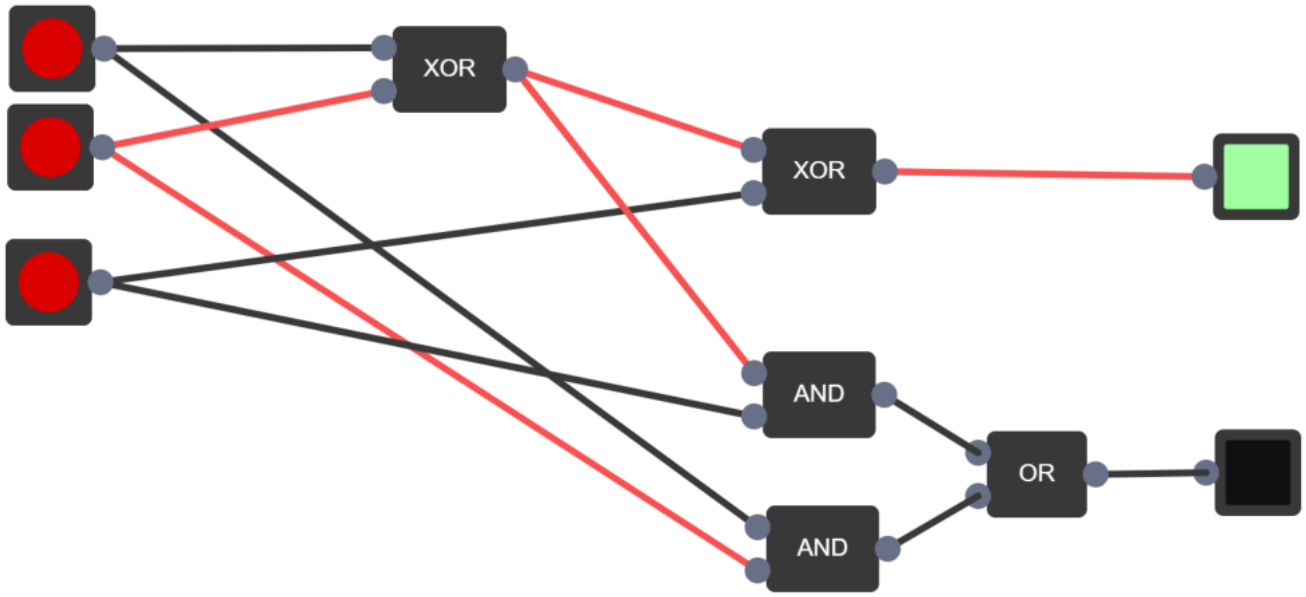


Figure 1 eksempel på kredsløb

PROBLEMFORMULERING

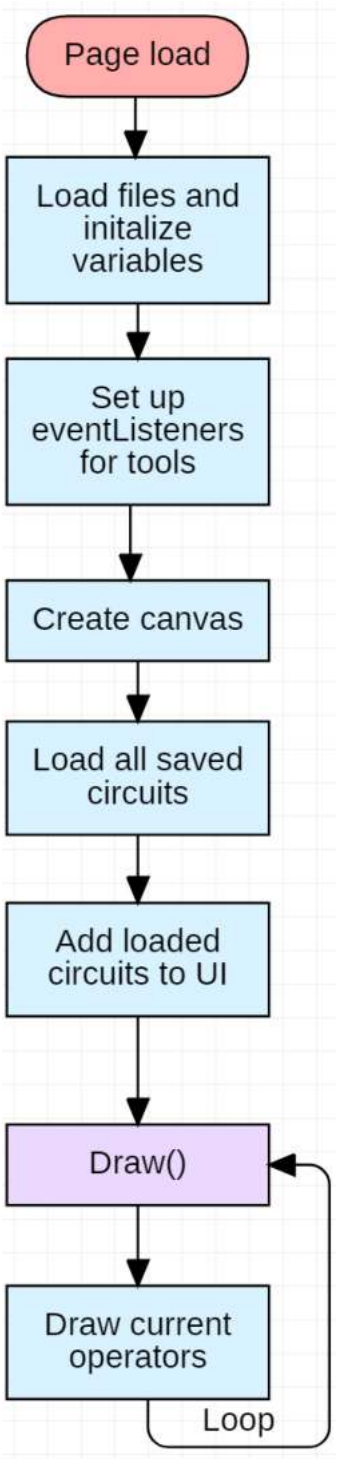
Det kan være meget svært at få et overblik hvordan en computer fungerer, når det ikke er noget man kan se med sine egne øjne. Dette gør det rigtig svært at forstå hvordan det egentligt hænger sammen. Specielt fordi alle diagrammer af kredsløb online er tekniske tegningen, som kan være svære at forstå for nogen som ikke kender dem. Programmet har nogle udfordringer der skal overkommes, så der er nogle specielle krav som programmet skal overholde, før det er helt færdigt:

- Hvordan kan relationen mellem de forskellige objekter simuleres?
- Hvordan bliver programmet interaktivt?
- Hvordan skal brugeren interagere med programmet?
  - Oprettelse af objekter.
  - Tegne sammenhæng mellem objekter.
- Hvordan skal man kunne samle alle objekterne i en samlet pakke?

PROGRAMMET

I programmet finder man to forskellige element som fylder mest. Den første er det grafiske design, programmet skulle gerne være lækkert at kigge på, og nemt at forstå. Det andet er at programmet skal være nemt at interagere med, men samtidigt være et kraftigt værktøj.

Flowchart 1 viser hvad der sker i programmet når man starter det. Groft sagt så starter programmet med at hente alle filer, så bliver der gjort nogle variable klar, herefter kører setup funktionen som opretter et canvas, og som henter alle de tidligere gemte kredsløb og tilføjer dem til UI. Der sker ikke så meget i draw loopet, her looper den bare over alle operatørerne og kalder deres draw() metode.



Flowchart 1 main

LIBRARY

Programmet fungerer ved hjælp af et library kaldet p5.js, som er at library med en masse funktionalitet til at tegne på et HTML Canvas element. p5 fungerer som et abstraktionslag ovenpå den indbyggede canvas API som findes i browsere.

I p5 findes der to funktioner som man næsten ikke kan undvære, `setup()` og `draw()`. Setup funktionen bruges til at køre et stykke kode når programmet starter, dette stykke kode bliver kun kørt en gang.

```
(window as any).setup = () => {
  createCanvas(windowWidth, windowHeight);

  loadAllCircuits().forEach((name) => addCombinedOperatorToUI(name));
};
```

Kode 1 setup

Draw er et stykke kode som bliver kørt hver frame[1], dette betyder at hvis man har noget kode som skal køres hele tiden, så er det her det skal være.

```
(window as any).draw = () => {
  background(255);
  operators.forEach((cur) => cur.draw());
};
```

Kode 2 draw loop

Udover disse to så findes der også tre andre funktioner, som bliver brugt i programmet: `mousePressed()`, `mouseDragged()` og `mouseReleased()`.

GUI

I dette afsnit beskrives bruger interfacet som er det brugeren, interagerer med når de bruger programmet. GUI'en[2] er en essentiel del af programmet, det er også derfor det skal være let og overskueligt. Dette er men som en

[1] Cirka 30 gange i sekundet
[2] Graphical User Interface

undervisnings materiale til alle aldersgrupper, det er derfor vigtigt at gøre sig nogle overvejelser om hvordan det skal struktureres.

VÆRKTØJS BAR

Når man bruger programmet, har man nogle forskellige "værktøjer" at vælge i mellem, disse består af de forskellige operatører såsom, knapper, forskellige gates og outputs. For at tilføje en ny operatør til sit kredsløb skal man trække det ud på lærredet, hvor efter den korrekte operator vil blive oprettet. Det er også her custom operatører vil blive tilføjet.

Figure 2 værktøjs bar

Udover de forskellige operatører, så er der også en knap adskilt fra resten. Denne knap bruges til at samle alle de nuværende operatører i en pakke.

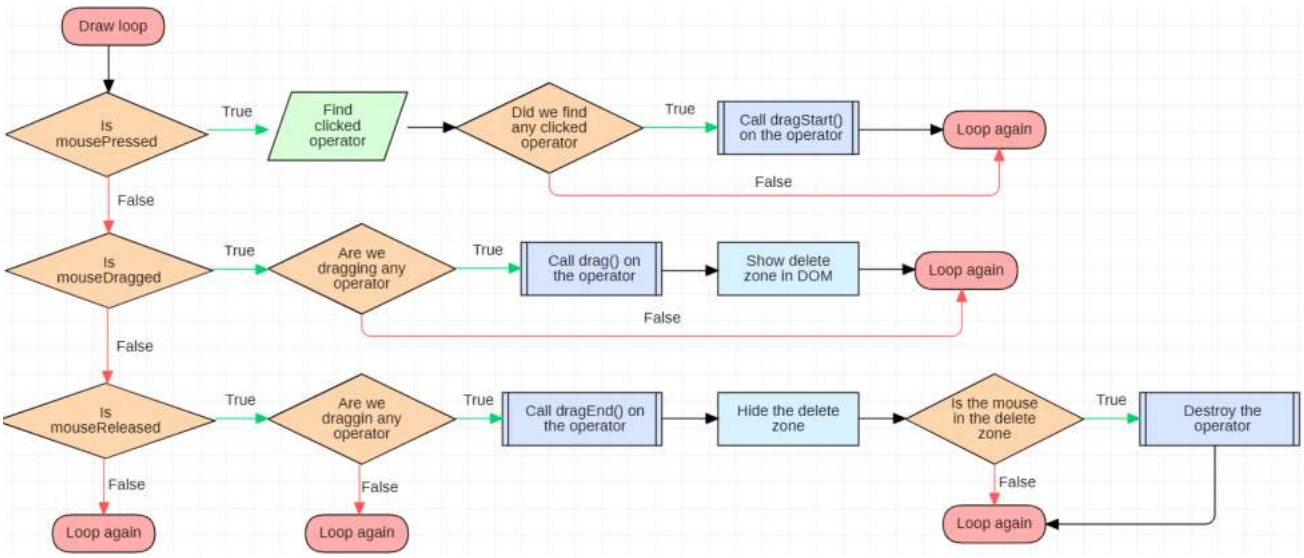
DRAG AND DROP

Både når man skal oprette en operatør og når man skal flytte på dem, så bliver der brugt drag-and-drop. Dette betyder at man bare skal bevæge musen over en ting derefter trykke på musen og trække den rundt.

Hvis man er i gang med at flytte på en operatør, så kommer det også et felt frem til at slette den nuværende ting. Hvis man slipper musen over det felt, så vil den blive slettet, og alle forbindelser til den vil blive destrueret.

KODE

Her er vist et Flowchart og en beskrivelse over hvordan drag-and-drop koden fungerer.



Flowchart 2 drag-and-drop

```
public mouseOver(): boolean {
  return mouseX > this.pos.x - (this.width / 2)
    && mouseX < this.pos.x + (this.width / 2)
    && mouseY > this.pos.y - (this.height / 2)
    && mouseY < this.pos.y + (this.height / 2);
}
```

Kode 3 check om musen er over

Alle operatører har en position, længde og bredde, dette betyder at vi kan tjekke om musen er inden for området.

Dette er en metode som alle operators har, da de arver den fra deres parent class.

```
// Loop over all operators and find the first one that is cl
// Then drag it to the mouse position
let draggingItem: GenericOperator | undefined;
(window as any).mousePressed = () => {
  const clicked = operators.find((cur) => cur.mouseOver());
  if (clicked) {
    clicked.dragStart();
    draggingItem = clicked;
  }
};
```

Kode 4 find den klikkede operatør

Det betyder at koden for at tjekke hvilken operatør der er blevet klikket på, bliver rigtig simpel. Det ses i koden at der bruges en higher-order funktion til at finde den rigtige operatør.

OPERATØRER

Der findes to forskellige typer operatører i programmet, den færdige operator og en custom operator. Begge typer har samme funktionalitet, i form af input, outputs og at man kan trække dem rundt.

FÆRDIGE OPERATORER

Når man først starter programmet, har man kun færdige operatører. Dette er de hardcoded som står direkte i programmet, enten fordi det er ikke logiske elementer, såsom knapper og outputs, eller fordi det er laveste gates man kan lave, såsom AND- OR- og NOT-gates. Alle disse operatører arver direkte fra `GenericOperator` classen, implementerer deres egen `logic()` metode.

```
logic() {
  this.outputs[0].setStatus(this.inputs[0].status && this.inputs[1].status);
}
```

Kode 5 logic metode fra AND-gate

CUSTOM OPERATORER

Udover de færdige operatører så kan man også lave sine egne, ved at samle de operatører som er på lærredet. Dette spiller godt sammen med at programmet er udviklet til undervisning og giver god mulighed for at eksperimenter med sine egne kreationer.

For at lave custom operatør skal man først lave et kredsløb på lærredet, herefter trykke på knappen ”Create operator”, så vil programmet samle alle operatørerne i en.

Flowchart 3 opret custom operatør

Her er koden der lytter til hvornår knappen bliver trykket. Når den bliver klikket, spørger programmet brugeren hvad den nye operatør skal hedde. Hvis ikke brugeren har indtastet noget navn, så sker der ikke noget. Herefter kalder programmet en anden funktion som, tager alle operatørerne og gemmer dem i browseren under et bestem navn. Og til sidst vil en knap blive tilføjet til UI'en.

```
// Listen for when the 'Create Operator' is pressed
// Then combine all the current operators into one
document.getElementById('new-operator')?.addEventListener('click', () => {
  // Ask the user what it should be called
  const name = prompt('What will you call this new operator');
  if (!name) return alert('You can\'t create an operator without a name');

  // Save the operators to localStorage
  saveCircuitInLocalStorage(operators, name);

  // Then erase the current operators
  operators.length = 0;

  // Add the button to UI
  addCombinedOperatorToUI(name);
});
```

Kode 6 "Create operator" event listener

KLASSER

I programmet findes der tre store klasser, og endnu flere mindre klasser. Disse klasser samler funktionalitet som let kan blive genbrugt rundt omkring i programmet. For eksempel er alle færdige operatører lavet som deres egen klasse, som arver en masse funktionalitet fra en abstrakt operatør klasse. Alle nodes og wires er også klasser, hvilket også gør det let at håndtere disse klasser.

GENERIC OPERATOR

Dette er den største klasse i hele programmet, denne klasse indeholder en masse funktionalitet som alle operatører har brug for, såsom drag-and-drop, input & output nodes og at tegne sig selv. Klassen er abstrakt hvilket betyder at man kun kan arve funktionalitet fra den, og ikke lave en instans af klassen.

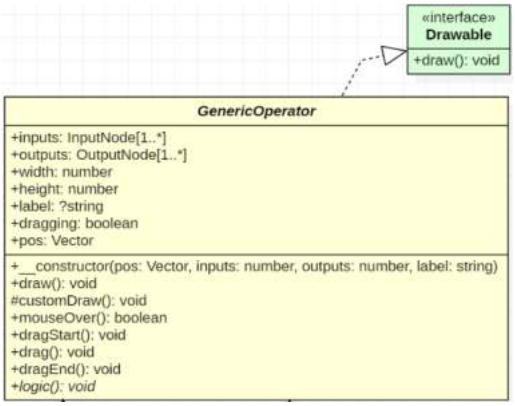


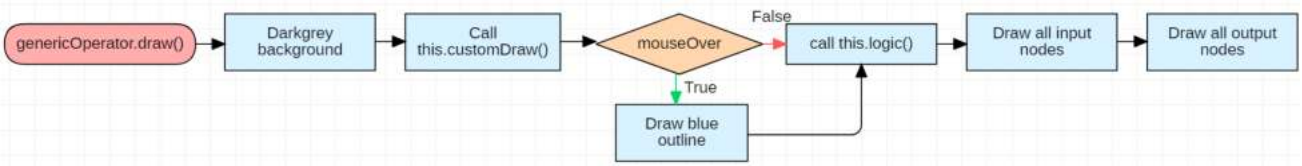
Figure 3 GenericOperator klasse diagram

Når en færdig operator udvider denne klasse, så er det eneste de skal gøre, er at kalde GenericOperator constructoren, og fortælle den hvor mange input og output nodes den skal have, samt en label. Så sørger GenericOperator for at oprette dem.

Udover inputs og outputs, så skal hver operatør også have en logic metode, denne bliver kaldet hver frame og har adgang til både inputs og outputs. Logic metodens job er at give et bestemt output ud fra en eller flere inputs.

```
// Every child class needs
abstract logic(): void;
```

Kode 7 abstract logic metode



Flowchart 4 draw metode GenericOperator

GenericOperator har også operators draw metode, dette er den metode der står for at gøre hvad der skal gøres hver frame. Det første metoden gør er at tegne selve knappens baggrund. En child-operatør kan implementere sin egen draw metode ved at override `customDraw()` metoden, som standard så bliver der skrevet operatørens label.

NODES

Der er faktisk to forskellige node klasser som begge arver funktionalitet fra en GenericNode klasse, dette er fordi at både input og output, deler en masse metode navne som har en lidt forskellig implantation. Node klassens job er at agere som en mellemmand mellem operatører og wires. Dette betyder at operatører slet ikke har direkte adgang til wires, og skal bruge nodes metoder for at hente og sætte status på wires. Operatorsene gemmer deres nodes i to arrays, fordi at de kan have flere inputs eller outputs. Derfor skal man bruge indexet på den node man gerne vil finde.

```
Get status
this.inputs[0].status;
Set status
this.outputs[0].setStatus(true);
```

Kode 8 get/set status

WIRES

Wires er den mest simple klasse, den formål at dele en boolean mellem en input- og en output-node. Derfor har den kun 3 properties som er vigtige: dens status, en reference til en input node, og en reference til en output node. Så er det nodes der henter status på den. Udover dens properties så har den også nogle metoder, her er den vigtigste `draw()`, som tegner en streg mellem to nodes, og indikere dens status.

OBJEKT ORIENTERET PROGRAMMERING

Eftersom at det program bruger mange forskellige kopier af det samme projekt, er det oplagt at bruge designmønstret objekt orienteret programmering. OOP[3] handler om at pakke lignende funktion en i en samlet klasse, som kan genbruges flere steder. Under OOP findes der fire grundprincipper:

- Indkapsling
- Abstraktion
- Nedarvning
- Polymorfi

INDKAPSLING

Indkapsling handler om at begrænse adgangen til en klasses interne data, så andre dele af programmet kun har adgang til de ting som de har brug for. Dette ses blandt andet i programmet i input og output klasserne, her er det kun det enkelte objekt der har adgang til dets ledning(er). Dette betyder at hvis ledningerne bliver pillet ved, så kan vi være sikre på at det er sket gennem en node.

```
private wires: Wire[] = [];
private wire?: Wire;
```

Kode 9 private properties

ABSTRAKTION

Nogle gange kan det være rart at tage noget avanceret logik ud af sin kode og gemme det væk i en klasse, så at man kun skal kalde en metode på klassen når man gerne vil have noget funktionalitet. Dette er konceptet bar abstraktion. Dette kunne for eksempel være at oprette de nødvendige input og output nodes hos en operator, derfor er denne funktionalitet pakket sammen inde i constructoren på GenericOperator. Så når en child-class gerne vil have nogle nodes, så kalder de bare deres parent constructor[4] med de nødvendige argumenter, så sørger den for det.
Andre eksempel på dette i koden, kunne også være drag-and-drop funktionaliteten, eller at tegne selve operatøren.

```
super(2, 1, 'AND');
```

Kode 10 AND-gate constructor

[3] Objekt orienteret programmering
[4] Et "super" call

NEDARVNING

En ”child”-klasse kan arve funktionalitet og data fra en anden klasse. Dette er kerneprincippet i der gør det helt vildt let at oprette nye operatører, da de bare kan arve al den nødvendige funktionalitet fra GenericOperator.

POLYMORFI

I objekorienteret programmer betyder polymorfi, at to klasser kan nedarve fra den samme forældreklasse, men implementere dem forskelligt. Et eksempel på dette er node klasserne, input og output nodes minder rigtig meget om hinanden, med den ene forskel at inputs kun har en wire, og outputs kan have flere. Dette betyder at man ikke kan lave en klasse til begge nodes, da de har lidt forskelligt funktionalitet, men vi kan lave en klasse med en masse abstrakte[5] metoder som hver node arver fra, hvorefter de implementerer dem hver for sig.

KONKLUSION

Målet med dette program var at skabe et program der overskueligt giver en ide om hvordan computere fungerer i det fysiske lag. For at opnå dette mål har det været nogle undermål, som også ses i det færdige produkt. Relationerne mellem de forskellige objekter kan simuleres ved hjælp af nodes og wires, som forbinder alle operatørerne, dette gør det overskueligt og nemt at simulere et komplekst system af forskellige gates og operatører. Programmet er blevet gjort interaktivt ved at lade brugeren kreere deres egne system, og kombinere dem så de kan genbruges. Programmet kan dog gøres en smule mere overskueligt hvis man tegnede forbindelserne mellem de forskellige operatører på en pænere måde, for eksempel ved at lade brugeren trække deres egne wires i stedet for at lave en lige linje fra punkt A til punkt B. Funktionaliteten der gør det muligt at samle et kredsløb i en operatør fungerer som skal, og gemmer samtidig den nye operatør på brugerens computer så den kan genbruges næste gang brugeren starter programmet.

Kildekode: https://github.com/DenseOriginal/Computer-Basics
Selve programmet: https://computer-basics.netlify.app/

[5] Metoder som forældreklassen ikke implementerer, men som alle under klasser skal implementere.

BILAG

Main.ts

```ts
/// <reference path="../node_modules/@types/p5/global.d.ts"/>

import { AndGate } from './classes/and-gate';
import { Input } from './classes/input';
import { Clock } from './classes/clock';
import { CombinedOperators } from './classes/combined-operators';
import { GenericOperator } from './classes/generic-operators';
import { NotGate } from './classes/not-gate';
import { OrGate } from './classes/or-gate';
import { Output } from './classes/output';
import { PulseButton } from './classes/pulse-button';
import { loadAllCircuits, loadCircuitFromLocalStorage, saveCircuitInLocalStorage } from './save-load';
import { BitAdder } from './classes/1bit-adder';

const operators: GenericOperator[] = [];

(window as any).setup = () => {
  createCanvas(windowWidth, windowHeight);

  loadAllCircuits().forEach((name) => addCombinedOperatorToUI(name));
};

(window as any).draw = () => {
  background(255);
  operators.forEach((cur) => cur.draw());
};

// Add event listener for dragend event on all list events with data-tool value
document.querySelectorAll('[data-tool]').forEach((cur) => {
  cur.addEventListener('dragend', () => {
    // When an HTML Element is dragged and then dropped
    // Then find out what tool it is, and create a new tool
    const tool = cur.getAttribute('data-tool');
    createOperator(tool as Tools);
  });
});

// Helper for creating a new operator on the screen
// It instantiates a new class corosponding to the tool that was passed in
// And then it appends the newly created operator to the operators array
type Tools = 'pulse' | 'clock' | 'input' | 'output' | 'andGate' | 'orGate' | 'notGate' | '1bitAdder';
function createOperator(tool: Tools): void {
  let newOperator: GenericOperator | undefined;

  switch (tool) {
  case 'andGate':
    newOperator = new AndGate();
    break;
  case 'pulse':
    newOperator = new PulseButton();
    break;
  case 'clock':
    newOperator = new Clock();
    break;
  case 'output':
    newOperator = new Output();
    break;
  case 'input':
    newOperator = new Input();
    break;
  case 'notGate':
    newOperator = new NotGate();
    break;
  case 'orGate':
    newOperator = new OrGate();
    break;
  case '1bitAdder':
    newOperator = new BitAdder();
    break;
  default:
    const exhaustiveCheck: never = tool;
```

```
72        throw new Error(`Unhandled tool case: ${exhaustiveCheck}`);
73      }
74
75      newOperator.pos.set(createVector(mouseX, mouseY));
76
77      operators.push(newOperator);
78    }
79
80    // Listen for when the 'Create Operator' is pressed
81    // Then combine all the current operators into one
82    document.getElementById('new-operator')?.addEventListener('click', () => {
83      // Ask the user what it should be called
84      const name = prompt('What will you call this new operator');
85      if (!name) return alert('You can\'t create an operator without a name');
86
87      // Save the operators to localStorage
88      saveCircuitInLocalStorage(operators, name);
89
90      // Then erase the current operators
91      operators.length = 0;
92
93      // Add the button to UI
94      addCombinedOperatorToUI(name);
95    });
96
97    // Get the deletion zone element, and show it when the user is dragging an operator
      around
98    const deletionZone = document.getElementById('deletion-zone');
99
100   // Loop over all operators and find the first one that is clicked
101   // Then drag it to the mouse position
102   let draggingItem: GenericOperator | undefined;
103   (window as any).mousePressed = () => {
104     const clicked = operators.find((cur) => cur.mouseOver());
105     if (clicked) {
106       clicked.dragStart();
107       draggingItem = clicked;
108     }
109   };
110
111   (window as any).mouseDragged = () => {
112     // If there's an item being dragged, then call the drag() method on them
113     if (draggingItem) {
114       draggingItem.drag();
115
116       // Show the deletion zone
117       deletionZone?.classList.remove('hidden');
118     }
119   };
120
121   (window as any).mouseReleased = () => {
122     // When ever the mouse is released, call the dragEnd() method on the item
123     // Wether or not we're actually dragging an item, using the optional chaining
124     draggingItem?.dragEnd();
125
126     // Hide the deletion zone, even if it wasn't show
127     deletionZone?.classList.add('hidden');
128
129     // If the user dropped the an operator in the deletionZone
130     // Then destroy it
131     if (draggingItem && mouseY < 100) {
132       // Tell the operator to destroy all it's connections
133       draggingItem.destroy();
134
135       // Find an remove the operator from the array of operators
136       const indexOfOperator = operators.findIndex((op) => op == draggingItem);
137       operators.splice(indexOfOperator, 1);
138     }
139
140     draggingItem = undefined;
141   };
142
143   // Takes in the name of a combinedOperator
```

```
144   // Adds it to the UI
145   // And setup eventlistner
146   function addCombinedOperatorToUI(name: string): void {
147     const stringifiedCircuit = loadCircuitFromLocalStorage(name);
148     if (!stringifiedCircuit) throw new Error(`Unknown circuit: '${name}'`);
149
150     // Create the tool button in the UI
151     const toolButton = document.createElement('li');
152     toolButton.draggable = true;
153     toolButton.innerText = name;
154
155     // Insert the button before the spacer
156
      document.getElementById('insert-before-here')?.insertAdjacentElement('beforebegin',
      toolButton);
157
158     // Setup ondragend handler
159     toolButton.addEventListener('dragend', () => {
160       const newOperator = CombinedOperators.fromString(stringifiedCircuit, name);
161       newOperator.pos.set(createVector(mouseX, mouseY));
162       operators.push(newOperator);
163     });
164   }
165
```

Helpers.ts

```
1   import { GenericOperator } from './classes/generic-operators';
2
3   export const getRandID = () => [...Array(6)].map(() => Math.floor(Math.random() *
    16).toString(16)).join('');
4
5   // Operator map and stuff
6   type GenericOperatorConstructor = new (...args:any[]) => GenericOperator;
7
8   const operatorMap = new Map<string, GenericOperatorConstructor>();
9
10  export function registerOperator(constructor: GenericOperatorConstructor) {
11    const { name } = constructor;
12    operatorMap.set(name, constructor);
13  }
14
15  export function getOperator(name: string): GenericOperatorConstructor | undefined {
16    return operatorMap.get(name);
17  }
18
```

Save-load.ts

```ts
import { GenericOperator } from './classes/generic-operators';
import { Wire } from './classes/wire';
import { getOperator } from './helpers';

export type OperatorDescription = { id: string, className: string };
export type ConnectionDescription = {
  id: string,
  from: { id: string, node: number },
  to: { id: string, node: number },
}
export interface RelationMap {
  operators: OperatorDescription[],
  connections: ConnectionDescription[],
}

export function stringifyOperators(operators: GenericOperator[]): string {
  const relations: RelationMap = {
    operators: [],
    connections: [],
  };

  // Loop over all the operators that we're givin
  operators.forEach((op) => {
    if (op.constructor.name == 'CombinedOperators') throw new Error('Can\'t do CombinedOperators yet...');

    // Register the current operators
    relations.operators.push({
      id: op.id,
      className: op.constructor.name,
    });

    // We only need to loop over every input
    // Because we can be sure that every wire is between an output and an input
    // Therefore we only need to check inputs or outputs, not both of them.
    // If we do check both then we'll encounter wires we have already looked at.
    op.inputs.forEach((node) => {
      // A connection goes from Output -> Input
      const connection = node.getWireRelation();
      if (!connection) return; // If the relation is undefined, just skip it

      relations.connections.push(connection);
    });
  });

  return JSON.stringify(relations);
}

export function parseOperators(input: string): GenericOperator[] {
  const relations = <RelationMap>JSON.parse(input);
  const operators = new Map<string, GenericOperator>();

  // Loop over every operator
  // And instatiate them
  relations.operators.forEach((opDescription) => {
    // Destructure the operator description
    const { id, className } = opDescription;

    // Try to retrieve the operator class, from the operatorMap
    const operatorClass = getOperator(className);

    // If the operatorClass wasn't found then throw an error
    if (!operatorClass) throw new Error(`Unknown class '${className}'`);

    // eslint-disable-next-line new-cap
    const newOperator = new operatorClass();

    // Give the new operator the correct ID
    newOperator.setId(id);

    // Add the new operator to the Map with the id as the key
    operators.set(id, newOperator);
  });
```

```
73
74      relations.connections.forEach((connection) => {
75        // Destructure the connection
76        const { id, from, to } = connection;
77
78        // Instantiate the new wire, with the correct ID
79        const newWire = new Wire(id);
80
81        // Retriwve the correct opeators from the Map
82        const fromOperator = operators.get(from.id);
83        const toOperator = operators.get(to.id);
84
85        // Check if both operators exist
86        if (!fromOperator) throw new Error(`Incorrect id for fromOperator: ${from.id}`);
87        if (!toOperator) throw new Error(`Incorrect id for toOperator: ${to.id}`);
88
89        // Connect the wire to the correct nodes
90        newWire.connect(toOperator.inputs[to.node], fromOperator.outputs[from.node]);
91      });
92
93      return Array.from(operators.values());
94    }
95
96    export function saveCircuitInLocalStorage(operators: GenericOperator[], name:
      string): void {
97      const stringifiedOperators = stringifyOperators(operators);
98      const stringToSave = `${name}|${stringifiedOperators}`;
99
100     // Prefix the key with 'circuit' to avoid collision between other keys
101     localStorage.setItem(`circuit-${name}`, stringToSave);
102   }
103
104   export function loadCircuitFromLocalStorage(name: string): string | undefined {
105     const rawItem = localStorage.getItem(`circuit-${name}`);
106     if (!rawItem) return undefined;
107
108     // Split the raw item by the delimeter '|'
109     // And  get the second item in the array
110     const stringifiedOperators = rawItem.split('|')[1];
111
112     // Parse and return the operators
113     return stringifiedOperators;
114   }
115
116   export function loadAllCircuits(): string[] {
117     const circuits: string[] = [];
118
119     for (let idx = 0; idx < localStorage.length; idx++) {
120       const key = localStorage.key(idx);
121       if (key && key.startsWith('circuit-')) {
122         // Get pretty name from the item
123         const name = localStorage.getItem(key)?.split('|')[0];
124
125         // If the name exists, then push it to the circuits array
126         if (name) circuits.push(name);
127       }
128     }
129
130     return circuits;
131   }
132
133   (window as any).loadAllCircuits = loadAllCircuits;
134
```

1bit-adder.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

export class BitAdder extends GenericOperator {
  constructor() {
    super(3, 2, '1B Adder');
  }

  logic() {
    const a = this.inputs[0].status;
    const b = this.inputs[1].status;
    const c = this.inputs[2].status;

    const firstSum = xor(a, b);
    const carrySum = xor(firstSum, c);

    this.outputs[0].setStatus(carrySum);
    this.outputs[1].setStatus((a && b) || (firstSum && c));
  }
}

registerOperator(BitAdder);

const xor = (a: boolean, b: boolean) => !(a && b) && (a || b);

```

and-gate.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

export class AndGate extends GenericOperator {
  constructor() {
    super(2, 1, 'AND');
  }

  logic() {
    this.outputs[0].setStatus(this.inputs[0].status && this.inputs[1].status);
  }
}

registerOperator(AndGate);

```

clock.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';
import { Wire } from './wire';

const size = 50;
const cycle = 1000;

// This is to prevent the wire being high for 1 frame
const activationTime = 50;

export class Clock extends GenericOperator {
  private lastTrigger: number = 0;

  constructor() {
    super(0, 1);
  }

  override customDraw(): void {
    push();

    const deltaTime = millis() - this.lastTrigger;

    // Draw arc for the cycle, TWO_PI is the full circle
    const arcAngle = map(deltaTime, 0, cycle, 0, TWO_PI);
    stroke('#f9f9f9');
    strokeWeight(4);
    noFill();
    arc(this.pos.x, this.pos.y, size * 0.6, size * 0.6, 0, arcAngle);

    if (deltaTime > cycle - activationTime) { this.outputs[0].setStatus(Wire.HIGH);
    } else { this.outputs[0].setStatus(Wire.LOW); }

    if (deltaTime > cycle) this.lastTrigger = millis();

    pop();
  }

  logic(): void { }
}

registerOperator(Clock);
```

not-gate.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

export class NotGate extends GenericOperator {
  public constructor() {
    super(1, 1, 'NOT');
  }

  logic(): void {
    this.outputs[0].setStatus(!this.inputs[0].status);
  }
}

registerOperator(NotGate);
```

combined-operators.ts

```ts
import { parseOperators } from '../save-load';
import { GenericOperator } from './generic-operators';
import { Input } from './input';
import { Output } from './output';

export class CombinedOperators extends GenericOperator {
  inputOperators: Input[];
  outputOperators: Output[];
  childOperators: GenericOperator[];

  constructor(operators: GenericOperator[], name: string) {
    // Extract the inputs and outputs from all the operators
    // And the sort them in order of their pos.y component
    // This is so that the input/output nodes will match up with the internal input/output operators
    const inputOperators = operators.filter((op) => op instanceof Input).sort((a, b) => b.pos.y - a.pos.y) as Input[];
    const outputOperators = operators.filter((op) => op instanceof Output).sort((a, b) => b.pos.y - a.pos.y) as Output[];

    // Pass the number of inputs and outputs to the GenericOperator
    // So that it can create the appopriate amount of nodes
    super(inputOperators.length, outputOperators.length, name);

    this.inputOperators = inputOperators;
    this.outputOperators = outputOperators;
    this.childOperators = operators;
  }

  logic(): void {
    // Loop over every input node, and set every internal input operator to the state
    this.inputs.forEach((inp, idx) => (this.inputOperators[idx].state = inp.status));

    // Loop over all the operators and run their logic
    this.childOperators.forEach((op) => {
      op.logic();
    });

    // Loop over every output node, and set it's state the match the internal output opetators state
    this.outputs.forEach((out, idx) =>
    out.setStatus(this.outputOperators[idx].state));
  }

  static fromString(stringifiedOperators: string, name: string): CombinedOperators {
    const operators = parseOperators(stringifiedOperators);
    return new CombinedOperators(operators, name);
  }
}

```

generic-operators.ts

```ts
import { Vector } from 'p5';
import { getRandID } from '../helpers';
import { Drawable, HasID } from './interfaces';
import { InputNode, OutputNode } from './node';

// Generic class for creating operators
// This parent class has functionality for creating the input/output nodes
// Aswell as other important features that is required in every operator such as:
//   Drawing itself and nodes
//   Checking if the mouse is hovering over
//   Draggin and dropping
//   Destorying this operator
//
// This class is an abstract class, meaning that you cannot create an instance of this class
// You can only extend this class, this is because every operator needs to have different logic
// Every child operator can also implement it's own draw method by overriding the customDraw() method
// By default the customDraw() method just draws the label, if the operator has one.
export abstract class GenericOperator implements Drawable, HasID {
  public inputs: InputNode[] = [];
  public outputs: OutputNode[] = [];
  public width: number;
  public height: number;
  public label?: string;

  private dragging: boolean = false;

  public pos: Vector = createVector();

  private _id = getRandID();
  public get id() { return this._id; } // eslint-disable-line no-underscore-dangle
  public setId(id: string) { this._id = id; } // eslint-disable-line no-underscore-dangle

  constructor(
    inputsN: number,
    outputsN: number,
    labelOrWidth: string | number = 50,
  ) {
    // Calculate the width and label
    // Depending on what type labelOrWidth is
    this.width = typeof labelOrWidth === 'string' ? textWidth(labelOrWidth) + 40 : labelOrWidth;
    this.label = typeof labelOrWidth === 'string' ? labelOrWidth : undefined;

    // Calculate the height of this operator
    // For every node give it 25 pixels of space
    // If there's 1 node or less, then just set the height as 50 pixels
    const most = Math.max(inputsN, outputsN);
    this.height = Math.max(most * 25, 50);

    // Generate input nodes and space evenly on the left side
    for (let i = 0; i < inputsN; i++) {
      this.inputs.push(new InputNode(createVector(
        -(this.width / 2) - 5,
        ((-this.height / 2) + (i * this.height / inputsN)) + (this.height / inputsN / 2),
      ), this));
    }

    // Generate output nodes and space evenly on the right side
    for (let i = 0; i < outputsN; i++) {
      this.outputs.push(new OutputNode(createVector(
        (this.width / 2) + 5,
        ((-this.height / 2) + (i * this.height / outputsN)) + (this.height / outputsN / 2),
      ), this));
    }
  }

  public draw(): void {
```

```
67        push();
68
69        rectMode(CENTER);
70        textAlign(CENTER, CENTER);
71
72        // Draw the darkgrey background
73        noStroke();
74        fill('#383838');
75        rect(this.pos.x, this.pos.y, this.width, this.height, 5, 5, 5, 5);
76
77        this.customDraw();
78
79        // If mouse is over the operator, draw a white border
80        if (this.mouseOver()) {
81          stroke('#95d8ff');
82          strokeWeight(3);
83          noFill();
84          rect(this.pos.x, this.pos.y, this.width * 1.15, this.height * 1.15, 5, 5, 5, 5);
85        }
86
87        pop();
88
89        this.logic();
90
91        // Draw all the nodes attached to this operator
92        this.inputs.forEach((input) => input.draw());
93        this.outputs.forEach((output) => output.draw());
94      }
95
96      protected customDraw(): void {
97        // This method just draws the label of the operator by default
98        // Child classes can overwrite this method and implement their own draw
99        if (this.label) {
100         fill('#fff');
101         textSize(14);
102         text(this.label, this.pos.x, this.pos.y);
103       }
104     }
105
106     public mouseOver(): boolean {
107       return mouseX > this.pos.x - (this.width / 2)
108         && mouseX < this.pos.x + (this.width / 2)
109         && mouseY > this.pos.y - (this.height / 2)
110         && mouseY < this.pos.y + (this.height / 2);
111     }
112
113     public dragStart(): void {
114       this.dragging = true;
115     }
116
117     public drag(): void {
118       if (this.dragging) {
119         this.pos.x = mouseX;
120         this.pos.y = mouseY;
121       }
122     }
123
124     public dragEnd(): void {
125       this.dragging = false;
126     }
127
128     public destroy(): void {
129       // This method will tell all nodes to destroy all wires
130       // So that no other operators are connected to this
131       this.inputs.forEach((cur) => cur.destroy());
132       this.outputs.forEach((cur) => cur.destroy());
133     }
134
135     // Every child class needs to write it's own logic method
136     abstract logic(): void;
137   }
138
```

input.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

const buttonSize = 50;

export class Input extends GenericOperator {
  public state: boolean = false;

  constructor() {
    super(0, 1, buttonSize);
    document.addEventListener('click', () => this.mouseClicked());
  }

  override customDraw() {
    push();

    rectMode(CENTER);

    noStroke();
    fill('#383838');
    rect(this.pos.x, this.pos.y, buttonSize, buttonSize, 5, 5, 5, 5);

    fill('#db0000');
    circle(this.pos.x, this.pos.y, buttonSize * 0.7);

    pop();
  }

  logic(): void {
    this.outputs[0].setStatus(this.state);
  }

  private mouseClicked(): void {
    const distSq = ((this.pos.x - mouseX) ** 2) + ((this.pos.y - mouseY) ** 2);
    const dist = Math.sqrt(distSq);

    if (dist < buttonSize * 0.7 / 2) {
      this.state = !this.state;
    }
  }
}

registerOperator(Input);

```

interfaces.ts

```ts
import { GenericOperator } from './generic-operators';

export interface Drawable {
  draw(): void;
}

export interface HasID {
  readonly id: string;
}

export interface SavedCombinedOperator {
  [ index: string ]: GenericOperator[]
}

```

node.ts

```ts
import { Vector } from 'p5';
import { getRandID } from '../helpers';
import { GenericOperator } from './generic-operators';
import { Drawable, HasID } from './interfaces';
import { Status, Wire } from './wire';

const radius = 15;

// This is stuff for creating a new wire between to nodes
let selectedOutputNode: OutputNode | undefined;
function selectNode(node: InputNode | OutputNode): void {
  if (node instanceof OutputNode) {
    // If the clicked node is an output
    // And we haven't selected an outputNode already
    // Then set selectedOutputNode to the node that was clicked on
    if (!selectedOutputNode) { selectedOutputNode = node; return; }

    // If the clicked node is the same node, that was already pressed
    // Then just cancel the selection
    if (selectedOutputNode.id == node.id) { selectedOutputNode = undefined; }
  } else {
    // If we have already selected an outputNode
    // And we have clicked an input node, then connect the two nodes with a wire
    if (selectedOutputNode) {
      new Wire().connect(node, selectedOutputNode);
      selectedOutputNode = undefined;
    }
  }
}

// Generic class for the two types of nodes
// The main difference between the input & ouput nodes, are that
// Input nodes only have one wire, and output nodes can have multiple wire
// This means that the methods should reflect this difference
// The class is abstract because even though the implmentation is going to be
different
// The method names should remain the same
abstract class GenericNode implements Drawable, HasID {
  readonly id = getRandID();
  public abstract readonly type: 'input' | 'output';
  get pos(): Vector { return this.parent.pos.copy().add(this.relativePos); }

  constructor(
    private relativePos: Vector,
    public readonly parent: GenericOperator,
  ) {
    document.addEventListener('click', () => this.mouseClicked());
  }

  // Abstract methods for connecting and removing a wire
  // The input and output nodes will implement a slightly different method
  // This is because the output node can have multiple wires, and the input can only
  have 1
  public abstract connectWire(wire: Wire): void
  public abstract removeWire(wire: Wire): void

  public getNodeNumber(): number {
    // Because the nodes are two different types, and exist in two different arrays
    // So we need to check what type the calling node is
    // This check is instead of making this method abstract
    if (this.type == 'input') return this.parent.inputs.findIndex((node) => node.id
    == this.id);
    // If it isn't input, then it's output
    return this.parent.outputs.findIndex((node) => node.id == this.id);
  }

  abstract draw(): void;
  protected clickHandler(): void {} // Empty handler for clicking on the node (only
  used by output node)
  private mouseClicked() {
    const distSq = ((this.pos.x - mouseX) ** 2) + ((this.pos.y - mouseY) ** 2);
    const dist = Math.sqrt(distSq);

```

```
70        if (dist < radius) {
71          // If the mouse is over the node
72          // Then call the clickHandler on this node
73          // And call the selectNode function
74          this.clickHandler();
75          selectNode(this as unknown as InputNode | OutputNode);
76        }
77      }
78
79      // Abstract method to remove all wires
80      // This is because input and output nodes handle wires differently
81      public abstract destroy(): void
82    }
83
84    export class InputNode extends GenericNode {
85      private wire?: Wire;
86      readonly type = 'input';
87
88      get status(): Status { return this.wire?.status || Wire.LOW; }
89
90      public connectWire(wire: Wire): void {
91        this.wire = wire;
92      }
93
94      public removeWire(wire: Wire): void {
95        if (this.wire?.id == wire.id) {
96          this.wire = undefined;
97        }
98      }
99
100     public draw(): void {
101       push();
102
103       noStroke();
104
105       fill('#677087');
106       circle(this.pos.x, this.pos.y, radius);
107       pop();
108     }
109
110     protected override clickHandler(): void {
111       // Shift click to delete the node
112       if (!selectedOutputNode && this.wire && keyCode == SHIFT) {
113         this.wire.destroy();
114       }
115     }
116
117     public destroy(): void {
118       this.wire?.destroy();
119     }
120
121     public getWireRelation() {
122       return this.wire?.describeRelation();
123     }
124   }
125
126   export class OutputNode extends GenericNode {
127     private wires: Wire[] = [];
128     readonly type = 'output';
129
130     public connectWire(wire: Wire): void {
131       this.wires.push(wire);
132     }
133
134     public removeWire(wire: Wire): void {
135       // When removing a wire, look through the wires array
136       // And filter out the one with a matching id
137       this.wires = this.wires.filter((w) => w.id != wire.id);
138     }
139
140     public flip(): void {
141       // Simply flip all the wires
142       this.wires.forEach((wire) => (wire.status = !wire.status));
```

```
143     }
144
145     public setStatus(status: Status): void {
146       this.wires.forEach((wire) => (wire.status = status));
147     }
148
149     public draw(): void {
150       push();
151
152       noStroke();
153       // Call the draw method on all it's wires
154       this.wires.forEach((wire) => wire.draw());
155
156       fill(selectedOutputNode?.id == this.id ? '#395699' : '#677087');
157       circle(this.pos.x, this.pos.y, radius);
158
159       // If this node is clicked, then highlight it with a different color
160       // And draw a line from the node to the mouse
161       if (selectedOutputNode?.id == this.id) {
162         strokeWeight(4);
163         stroke('#383838');
164         line(this.pos.x, this.pos.y, mouseX, mouseY);
165       }
166       pop();
167     }
168
169     public destroy(): void {
170       this.wires.forEach((wire) => wire.destroy());
171     }
172   }
173
```

or-gate.ts

```
1   import { registerOperator } from '../helpers';
2   import { GenericOperator } from './generic-operators';
3
4   export class OrGate extends GenericOperator {
5     constructor() {
6       super(2, 1, 'OR');
7     }
8
9     logic() {
10      this.outputs[0].setStatus(this.inputs[0].status || this.inputs[1].status);
11    }
12  }
13
14  registerOperator(OrGate);
15
```

output.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

export class Output extends GenericOperator {
  public state: boolean = false;

  constructor() {
    super(1, 0);
  }

  override customDraw() {
    push();

    rectMode(CENTER);
    noStroke();

    // Draw a smaller rectangle to represent the output
    // A high value is a green rectangle
    // And a low value is a darkgrey rectangle
    fill(this.inputs[0].status ? '#a0ffa0' : '#101010');
    rect(this.pos.x, this.pos.y, this.width * 0.75, this.height * 0.75, 2, 2, 2, 2);

    pop();
  }

  logic(): void {
    this.state = this.inputs[0].status;
  }
}

registerOperator(Output);

```

pulse-button.ts

```ts
import { registerOperator } from '../helpers';
import { GenericOperator } from './generic-operators';

const buttonSize = 50;
const pulse = 30;

export class PulseButton extends GenericOperator {
  constructor() {
    super(0, 1);
    document.addEventListener('mousedown', () => this.mouseClicked());
  }

  override customDraw(): void {
    push();

    noStroke();
    fill('#db0000');
    circle(this.pos.x, this.pos.y, buttonSize * 0.7);

    // Draw a little '1' in the lower right corner
    textAlign(CENTER, CENTER);
    textSize(buttonSize * 0.3);
    fill('#fff');
    text('1', this.pos.x + (buttonSize * 0.7 / 2), this.pos.y + (buttonSize * 0.7 /
    2));

    pop();
  }

  logic(): void { }

  private mouseClicked(): void {
    const distSq = ((this.pos.x - mouseX) ** 2) + ((this.pos.y - mouseY) ** 2);
    const dist = Math.sqrt(distSq);

    if (dist < buttonSize * 0.7 / 2) {
      this.outputs[0].setStatus(true);
      setTimeout(() => this.outputs[0].setStatus(false), pulse);
    }
  }
}

registerOperator(PulseButton);

```

wire.ts

```ts
import { getRandID } from '../helpers';
import { ConnectionDescription } from '../save-load';
import { Drawable, HasID } from './interfaces';
import { OutputNode, InputNode } from './node';

export type Status = boolean;

// Wire class that describes a connection between an input- and output-node
// A wire can only be connected to 1 input and 1 output
export class Wire implements Drawable, HasID {
  public status: Status = false;

  public output: OutputNode | undefined;
  public input: InputNode | undefined;

  // eslint-disable-next-line no-useless-constructor
  constructor(public readonly id = getRandID()) { /* */ }

  draw() {
    push();

    strokeWeight(4);
    stroke(this.status ? '#f55151' : '#383838');

    if (this.input && this.output) {
      line(
        this.output.pos.x,
        this.output.pos.y,
        this.input.pos.x,
        this.input.pos.y,
      );
    }

    pop();
  }

  public connect(input: InputNode, output: OutputNode) {
    this.input = input;
    this.output = output;
    input.connectWire(this);
    output.connectWire(this);
  }

  public destroy() {
    this.input?.removeWire(this);
    this.output?.removeWire(this);
  }

  public describeRelation(): ConnectionDescription | undefined {
    // If this wire isn't fully connected return undefined
    if (!this.output || !this.input) return;

    return {
      id: this.id,
      from: { id: this.output.parent.id, node: this.output.getNodeNumber() },
      to: { id: this.input.parent.id, node: this.input.getNodeNumber() },
    };
  }

  static HIGH: true = true;
  static LOW: false = false;
}
```