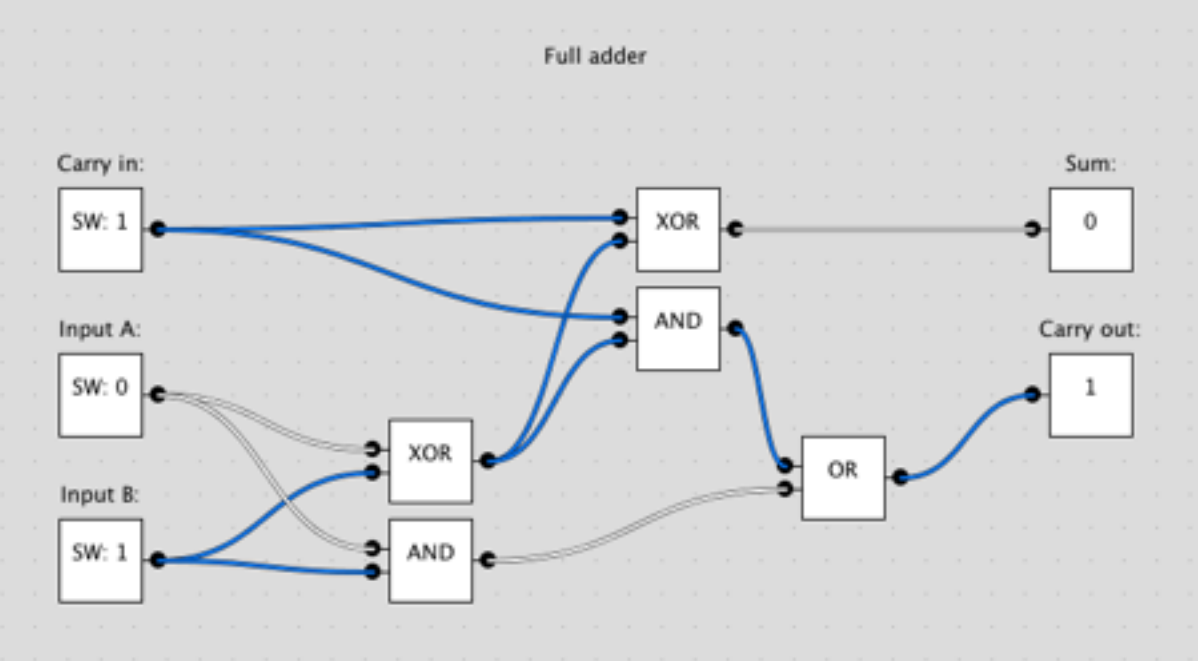
Projektbeskrivning

2016-05-01

Projektmedlemmar:
Erik Örjehag (erior950@student.liu.se)

Handledare:
Piotr Rudol (piotr.rudol@liu.se)

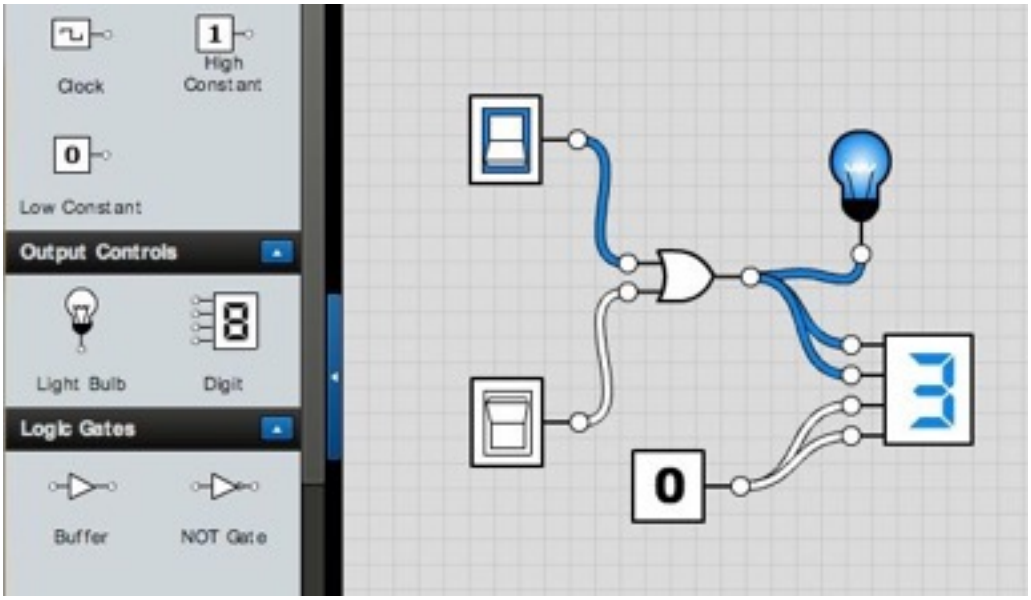


Innehåll

1. Introduktion till projektet

Jag vill utveckla en logiksimulator. I en logiksimulator kan man dra ut olika logiska komponenter och bygga en logisk krets. Ej att förväxlas med en simulator för elektriska kretsar. Denna simulator ska endast hantera ettor och nollor. Ett exempel på några logiska komponenter är grindar så som AND, NAND, OR, XOR, NOT och av/på-komponenter så som knappar, switchar eller lampor samt mer komplexa komponenter som en FlipFlops eller en LED-display.

Här är en bild från ett liknande projekt jag hittade online:



2. Ytterligare bakgrundsinformation

Jag kommer behöva göra en del research angående hur man ska propagera 1:or och 0:or genom kretsen när den ska uppdateras. Vad jag förstått kan man garantera ett förutsägbart beteende om kretsen inte innehåller loopar. Men om man inte tillåter loopar så försvinner många intressanta möjligheter när man ska bygga kretsar. Jag kommer därför tillåta loopar men försöka hitta en lösning, till exempel visa ett varningsmeddelande, om det skulle uppstå en paradox. Vad händer till exempel om man kopplar en 1:a till första ingången på XOR gate och sedan kopplar utgången tillbaka till den andra ingången. Beteendet blir odefinierat eftersom det är en paradox. Här är en stackoverflow-tråd som är relaterad: http://stackoverflow.com/questions/975211/creatingalogicgate-simulator

Jag tycker att detta är ett bra projekt eftersom det borde vara lätt att komma igång och bygga en liten kärna som man sedan kan bygga vidare på i princip hur långt som helst. Efter att jag simulerat en simpel krets med en enda switch och en lampa så kan jag börja lägga till en gate i taget, expandera menysystemet och göra drag-and-drop-funktionalitet, lägga till mer komplexa komponenter, lägga till funktionalitet för att designa egna komponenter, beroende på hur lång tid projektet tar.

3. Milstolpar

#	Beskrivning	
1	Ett fönster med en grafisk Compontent i. Fönstret har en menyrad och det går att stänga programmet genom att kryssa fönstret eller använda menyn.	Genomförd.
2	En icke modifierbar simulation av en 1:a som är sammankopplad med en lampa.	Genomförd.
3	En icke modifierbar simulation av en switch som är sammankopplad med en lampa.	Genomförd.
4	Det går att flytta runt komponenter i den färdigbyggda skissen och sammankoppla dem med hjälp av musen.	Genomförd.
5	Det går att lägga till nya komponenter till skissen genom att använda siffertangenterna för att välja viken komponent man vill lägga ut.	Genomförd.
6	Möjligheten att radera komponenter från skissen med DELETE-knappen.	Genomförd.
7	Det finns ett par logiska grindar så som AND, NOT, OR, XOR, NAND.	Genomförd.
8	En grafisk meny som man kan använda för att dra ut komponenter till skissen istället för att använda siffertangenterna.	Genomförd.
9	En ny inputkomponent som fungerar som en klocka som slår av/på med jämna intervall.	Genomförd.
10	Möjlighet att stänga av simulationen, återställa eller hoppa ett “steg” i simulationen med hjäp av menyn.	Ej genomförd.
11	En outputkomponent som visar en siffra baserad på binär input.	Genomförd.
12	En outputkomponent som visar antalet 1:or som har skickats till komponenten.	Ej genomförd.
13	Alla vanliga logiska grindar är implementerade.	Genomförd.
14	Möjligheten att lägga ut textkommentarer i skissen brevid de logiska komponenterana för att dokumentera skissen.	Genomförd.
15	En flipflop-komponent.	Genomförd.
16	Möjligheten att spara skisser till fil på disk och sedan öppna sparade skisser igen.	Delvis genomförd.
17	Möjligheten att bygga en egen logisk komponent med hjälp av andra komponenter, abstrahera bort implementationen, och sedan använda den i en skiss precis som en standardkomponent.	Ej genomförd.
18	Möjligheten att ångra modifikationer av skissen (Ctrl + Z)	Ej genomförd.
19	Möjligheten att zooma och flytta vyn.	Ej genomförd.
20	Möjligheten att öppna exempelskisser genom att använda menyn.	Genomförd.

4. Övriga implementationsförberedelser

Jag planerar att ha någon form av abstrakt förälderklass (Placeable) för alla komponenter som kan läggas ut i en skiss. Sedan en ytterligare abstrakt förälderklass (Logical) för alla logiska komponenter. Denna klass kan sedan ärvas från för varje logisk komponent och metoden som producerar en output kan skrivas över med unik kod för varje enskild komponent.

Jag känner mig lite osäker på hur koden som inte har med själva komponenterna ska struktureras. Exempelvis koden för att spara/öppna skisser, menyn med komponenter, ritning av komponenter samt uppdatering av logik. Men jag tror att jag kommer kunna fatta bättre beslut om jag jobbar iterativt istället för att försöka bestämma allt för mycket i förväg.

5. Utveckling och samarbete

Jag planerar att jobba ensam. Jag kommer använda Git för att versionshantera koden så att jag kommer åt den vart jag än är. Jag kommer jobba under en del av labbpassen för att kunna ställa frågor till handledaren, men även mycket hemifrån.

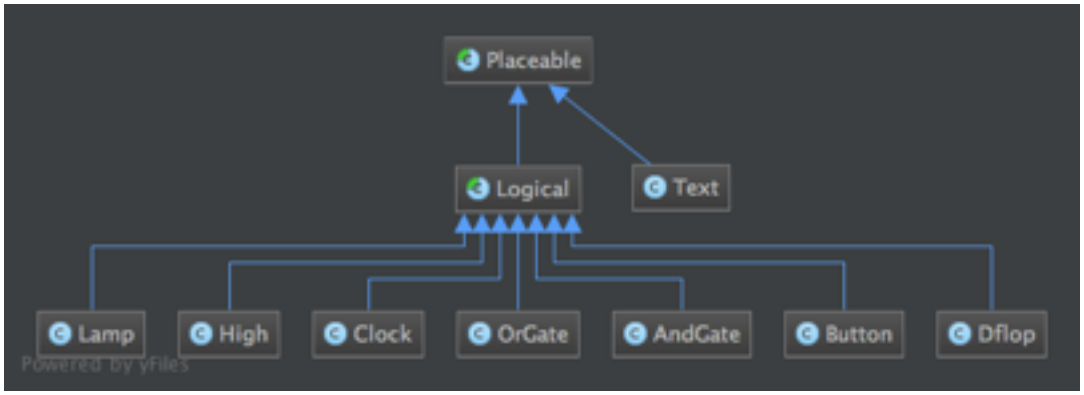
6. Implementationsbeskrivning

6.1. Övergripande

Den övergripande designen av programmet ser ut som följer. Klassen SimFrame ansvarar för layout, meny och vad som ska hända när man trycker på knappar i menyn. Klassen CompList ärver av JScrollPane och är den skroll-lista med komponenter som syns till vänster. SimComponent ärver av JComponent syns till höger i layouten. Denna klass ansvarar äger en instans av klassen Simulation och ansvarar för att ta användar-input från mus och tangentbord och skicka vidare till simulationen. Den ansvarar också för att kalla på situationens draw-metod när det behövs. Klassen Simulation håller allt state för en simulation. Den innehåller en lista på komponenter och logik för att sammankoppla dessa. Klassen Placeable är förälderklassen till alla komponenter som kan placeras i simulationen. Den innehåller delad bland annat funktionalitet för att flytta runt komponenter i simulationen. Text och Logical ärver i sin tur från Placeable. Klassen klassen Logical är abstrakt och kan inte instansieras, medan klassen Text inte är abstrakt och kan användas för att lägga till kommenterar till sina simulationer. Det finns en mängd logiska komponenter som alla ärver från förälderklassen Logical som kan läggas till i situationen och kopplas samman med varandra.

6.2.Struktur av simulering

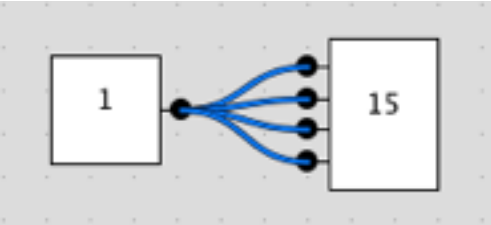
Alla logiska komponenter ärver från den gemensamma förälderklassen Logical. Denna förälderklass innehåller funktionalitet som är gemensam för att logiska komponenter. Den innehåller bland annat två listor över alla input- och output-sockets som tillhör komponenten. En AND-gate har till exempel två inputs och en output, men en 4 bit display har fyra inputs och inga outputs.



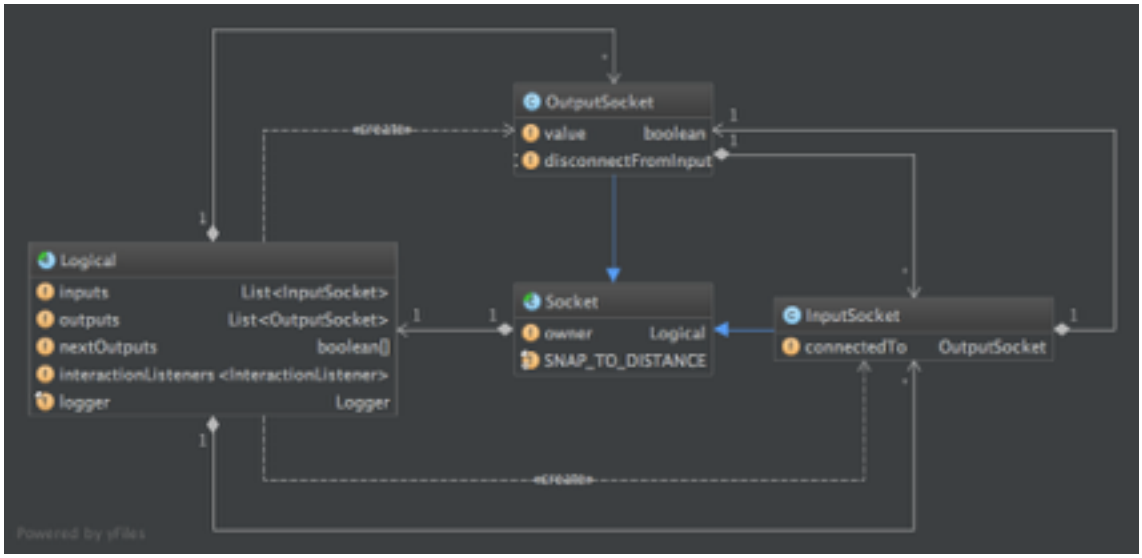
I förälderklassen Logical finns en abstrakt metod som heter func som bestämmer komponentens funktion, dvs hur komponenten reagerar på en viss input och vilken output den genererar. Kollar man i input-komponenten High ser vi att metoden func förväntar sig att det inte finns någon input och att den alltid kommer returnera outputen true. Kollar man istället på AndGate-klassens implementation ser vi att den förväntar sig två input-signaler och att outputen kommer vara de två signalerna AND:ade med varandra. I Dflop-klassens implementation av func finns det logik för att kolla om klocksignalen är en stigande flank och i så fall skicka vidare input-signalen till outputen.

För att koppla samman logiska komponenter med varandra används referenser från komponentens input-sockets till andra komponenters output sockets. Eftersom en input-socket endast kan vara kopplad till en output, men en output kan vara kopplad till flera inputs uppstår en fler-till-en-relation. Denna relation är smidig att modelera genom att ha en referens från input-sockets till outputsockets men inte tvärtom (se fältet connectedTo i klassen InputSocket). Om en OutputSocket vill koppla bort sig från en InputSocket som den är ansluten till måste den därför be input-socketen att ta bort relationen (connectedTo = null). Detta kan en OutputSocket göra genom att hålla en lista med all inputs som den är kopplad till (se fältet disconnectFromInputListeners i klassen OutputSocket).

Här är en bild som illustrerar fler-till-en-relationen mellan inputs och outputs.



På grund av denna en-till-flera-relation är det också smidigt att spara det faktiska signal-värdet som är vid en socker enbart i OutputSocket och sedan endast referera till värdet som är sparat där om man vill veta signal-värdet hos en InputSocket (se metoden getValue i klassen InputSocket).



Särskilt intressant i bilden ovan är alltså fältet connectedTo i InputSocket som refererar till en OutputSocket. Även fältet value av typ boolean i Outputsocket.

Varje gång en ändring sker som kräver att situationen uppdateras så kallas funktionen step i klassen Simulation. Denna metod kallar i sin tur på step funktionen i varje logisk komponent som använder func-metoden för att räkna ut vilket output som ska sättas härnäst på komponentens outputs. Denna output sparas först temporärt i listan nextOutputs i varje komponent. När alla outputs är beräknade itererar jag ytterligare alla komponenter och kallar på commit-metoden för sätta den faktiska outputen till de färden som är sparade i nextOutputs-listan. Innan jag hade implementerat flipflops gjordes allt direkt i step funktionen under en enda iteration. För att flipflops skulle fungerar var jag tvungen att dela upp uträkningen och tilldelningen av värden till outputs i två iterationer för att output från en flipflop inte skulle hinna propagera genom komponenten in i nästa under en och samma klockpuls. För att ändringar ska propagera genom kretsar med stort grind-djup så körs step-commit-proceduren för logiska komponenter flera gånger per step i simulationen (se fältet ITERATIONS_PER_STEP i klassen Simulation).

6.3.Drag and drop

Jag har implementerat mitt egna system för drag en drop av komponenter från listan till vänster till simulationen till höger i layouten. Klassen CompList har en lista på instanser av CompListItem. Varje instans av CompListItem får en klass som ärver av placable inskickad i konstruktorn. När en CompListItem plockas upp från listan till vänster skapas en CompListDraggable som läggs i ett lager ovanpå alla andra grafiska element (se JPanel front i SimComponent). Detta element kan sedan dras över till simulationen, varpå en ny instans av den barnklassen till Placable som just denna CompListItem blivit tilldelad skapas och läggs till i simulationen.

6.4. Användning av fritt material

Bilden av ett vitt plus-tecken i en grön cirkel är tagen från Google bilder. Inget annat material har används som jag inte själv har gjort eller som inte är del av Java 8. Den lilla bilden på ett plus lade jag till för att uppfylla betygskriteriet att använda resurser inlästa från disk och hantering av eventuella fel.

6.5. Användning av objektorientering

1)

Jag har tre **interfaces** i koden, StepListener, InteractionListener och CompDropTarget. Klassen SimComponent implementerar gränssnittet StepListener för att bli notiserad om uppdateringar i simulationen så att den kan rita om skärmen. Ett alternativ till detta hade varit att skicka med SimComponent till Simulation:s konstruktor och sedan låtit Simulation kalla på SimComponent:s repaint-funktion direkt efter en uppdatering. Jag gillade inte denna lösning lika mycket eftersom att det gör att Simulation blir mer hårt bunden till SimComponent. Med den lösning som jag har nu kan Simulation fortfarande fungera helt utan att det ens finns en SimCompoent. Det ända Simulation vet är att någon vill veta om att uppdateringar sker. Resten får den klass som blir notiserad sköta.

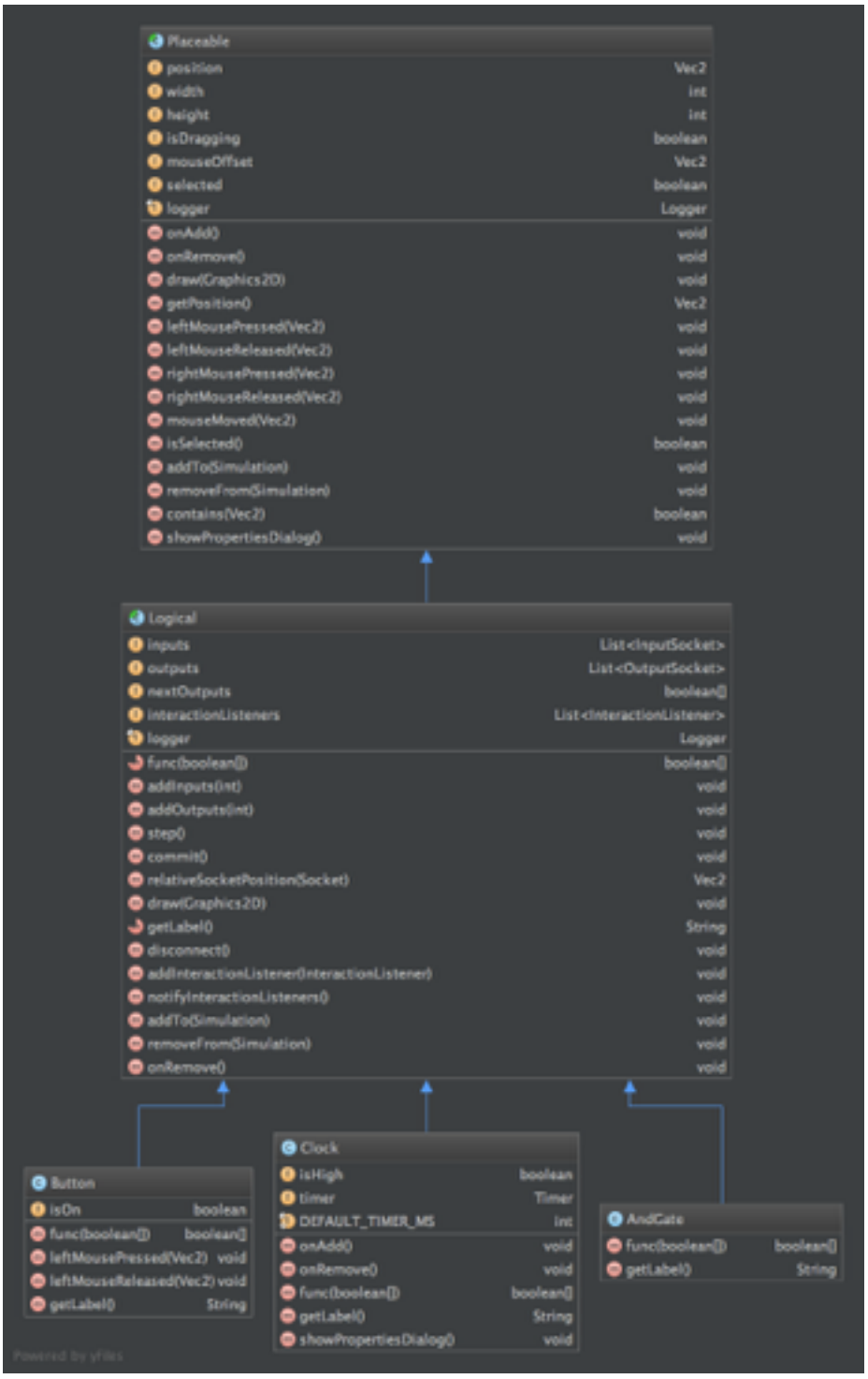
Klassen Simulation implementerar InteractionListener för att få reda på när en logisk komponent ändrar på sitt värde som ett resultat av en interaktion från användaren eller timer. Som exempel kan det vara så att en knapp ändrar sin output när man klickar på den. Då signaleras denna ändring till alla dess InteractionListener:s. Jag gillade denna lösning för att de logiska elementen inte själva måste känna till Simulation och kalla på dess step-metod utan bara behöver meddela ändringar och sedan låta den klass som lyssnar efter ändringar göra resten.

Klassen SimComponent implementerar ComDropTarget. Detta för att göra CompListItem mer generell i sin funktionalitet. Så länge en klass implementerar gränssnittet kan en CompListItem släppas på en instans av klassen.

Jag har använt gränssnitt för att minska beroendet mellan klasser. I ett icke OO-språk finns inte riktigt samma problem med tight kopplade klasser eftersom det inte finns klasser i huvud taget. Men man skulle kunna jämföra det med funktioner som är väldigt beroende av vilken indata det får. I ett icke OO-språk kan man försöka tänka på att skriva funktioner som är generella och kan användas på flera olika ställen i programmet.

2)

Placeable är en **abstrakt klass** som alla komponenter som kan läggas till i simulationen ärver av. Den innehåller delad funktionalitet för att flytta runt komponenter i simulationen. Det finns även kod för att rita en streckad linje runt komponenter som är markerade. I klassen Simulation finns en lista på alla placables vilket gör att man kan kalla på till exempel draw-metoden på alla komponenter utan att veta exakt vilken typ av komponent det är. I bilden nedan kan man se att den mesta funktionaliteten ligger i de båda abstrakta klasserna Placeable och Logical och att endast det det som gör de logiska komponenterna unika ligger i sina egna klasser.



3)

Jag använder **implementationsärvning med modifikation** i metoden draw i klassen Logical. Först kallar jag på super.draw() för att Placeable ska rita ut den streckade rektangeln om komponenten är markerad. Sen fortsätter jag med att rita själva logiska komponenten. Jag använder även implementationsärvning med modifikation i metoden rightMousePressed i Switch. Först kallar jag super.rightMousePressed för att koden i Placeable ska kolla om komponenten ska markeras. Sen fortsätter koden Switch att ändra värdet på komponentens output om Switch:en klickats på.

4)

Jag använder **sen dynamisk bindning** av metoden getValue i förälderklassen Socket. Om du kollar längst ner i metoden draw i klassen Simulation så kallar jag getValue() på den socket som användaren för tillfället håller på att dra en sladd från. Kollar man på implementationer av getValue() i OutputSocket ser man att den returnerar värdet direkt. Men kollar man istället på implementationen i InputSocket ser man att den hämtar värdet från den OutputSocket som den är kopplad till. Denna kring-gång är alltså abstraherad i förälderklassen och syns inte utifrån när man kallar getValue() på en Socket.

5)

I Logical finns en **abstrakt metod** kallad func som bestämmer funktionen hos en specifik logisk komponent. Metoden func kallas på i step-metoden i Logical trots att Logical inte vet hur funktionskroppen ser ut. Den vet bara att in kommer inputs och ut kommer nästa outputs. Själva implementationen av func ligger i varje barn till Logical. Om man jämför func hos till exempel FourBitDisplay, AndGate och Dflop ser man att de ser ganska annorlunda ut men att de i huvudsak har samma uppgift. Vilket är att ta in input-signaler och reagera på dessa för att ändra sitt eget state eller generera en output.

6)

Klassen ExampleFactory är en så kallad **faktory-klass** som har statiska metoder för att returnera olika exempel-simulationer. Ett alternativ till detta hade varit att göra en ny klass för varje exempel som ärver från Simulation och som i sin konstruktor byggde upp ett exempel. Jag gillar dock faktory-mönstret bättre eftersom funktionaliteten hos alla exempel är likadana, det är bara instansernas “state” som är annorlunda.

6.6. Motiverade designbeslut med alternativ

1)

I klassen Simulation finns en metod kallad add(Placeable placeable) som kan användas för att lägga till både vanliga placeable-komponenter så som Text men även logiska komponenter som till exempel en AndGate eftersom Logical ärver Placeable. För att lägga till en komponent som ärver Logical till simulationen behöver man dock göra några saker annorlunda jämfört med att lägga till en komponent som bara ärver Placeable. Skillnaden syns i metoderna addLogical och addPlaceable. För att ta reda på vilken av dessa metoder add-metoden i sin tur ska kalla på skulle jag kunna använt en if-sats som använder instanceof för att ta reda på om inparametern placeable i själva verket är en Logical. Istället för att göra det valde jag istället att lägga en metod i Placeable kallad addTo(Simulation sim) som i sin tur kallar på addPlaceable på simulationen man skickar in. I klassen Logical har jag skrivit över addTo(Simulation sim) så att den istället kallar på addLogical på

simulationen man skickar in. Jag tyckte detta var en cool lösning som gör att jag kan undvika en instanceof-check.

2)

I klassen CompListItem finns en metod som heter createInstance(int x, int y) som är intressant. Den används för att skapa en ny instans av den klass som skickas in i konstruktorn som Class<? extends Placeable> placeableClass. I klassen CompList skapar jag flera instanser av CompListItem och skickar med klasserna för alla komponenter som jag vill att man ska kunna lägga till i simulationen. Med hjälp av createInstance() skapar CompListItem en ny instans av komponenten när den dras och släps över den gråa ytan till höger.

En alternativ lösning till detta hade varit att skapa en copy-metod i Placeable. Då skulle varje CompListItem kunna ta en instans av varje komponent istället för själva klassen. När CompListItem släps över den gråa ytan hade man kunnat kalla på copy() för att få en ny instans som man lägger till i simulationen. Jag gillar dock inte denna lösning lika mycket eftersom man skulle behöva göra nya copy-metoder för alla barn som skiljer sig från Placeable. Generellt sätt tycker jag att copy-metoder brukar vara krångliga eftersom det lätt blir kvar referenser mellan data i instanserna efter att man försökt kopiera fält. Den lösning som jag implementerat tycker jag både känns felsäker och kräver väldigt lite kod.

3)

I klassen Placeable finns två metoder onAdd() och onRemove() som är intressanta. De har inga metods-kroppar så man kan tycka att de borde vara abstrakta. Men tanken är att detta är två metoder som ett barn bara ska behöva implementera om de är intresserade av att bli notifierade när de läggs till eller tas bort från simulationen. Jag var tvungen att lägga till dessa metoder när jag skapade klassen Clock. Denna klass äger en Timer som man måste kalla start() på när Clock läggs till i simulationen. Innan dess hade jag problemet att den instansen av Clock som låg i CompListItem i listan till vänster växlade mellan 0 och 1 vilket var väldigt distraherande. Jag är också tvungen att kalla stop() på timern i onRemove() för att simulationen ska sluta steppa efter att en Clock tagits bort från simulationen. Om ett barn inte skriver över metod-kroppen så kommer den tomma kroppen i Placeable användas.

4)

Koden för ritning är gemensam för alla logiska komponenter. Om du kollar i draw-metoden i klassen Logical så är det allt som behövs för att rita alla typer av logiska komponenter. Jag har alltså inte skrivit över denna metod i någon av barnen till Logical. Metoden draw i Logical använder den abstrakta metoden getLabel som varje barn till Logical måste implementera för att skilja på vilken text som ska stå på komponenten. Jag har valt att ha en metod getLabel istället för att bara ha ett vanligt fält för label för att jag vill ha lite mer kontroll över dynamiskt räkna ut vilken label som ska användas beroende på en komponents “state”. Till exempel vill jag visa 0 eller 1 beroende på om en Switch-komponent är aktiv eller inte.

5)

Den relation som finns mellan InputSocket och OutputSocket modeleras enligt mig på ett väldigt snyggt sätt. Den kan liknas med en fler-till-en-relation som är vanlig i många databaser. I en klassisk MySQL-databas kan man modelera en fler-till-en-relation genom att ha ett kolumn i tabellen med rader som kan referera till enbart en annan rad i en annan tabell. På ett liknande sätt har jag valt att ha ett fält kallad connectedTo i klassen InputSocket som refererar till en OutputSocket. För att en InputSocket ska kunna koppla

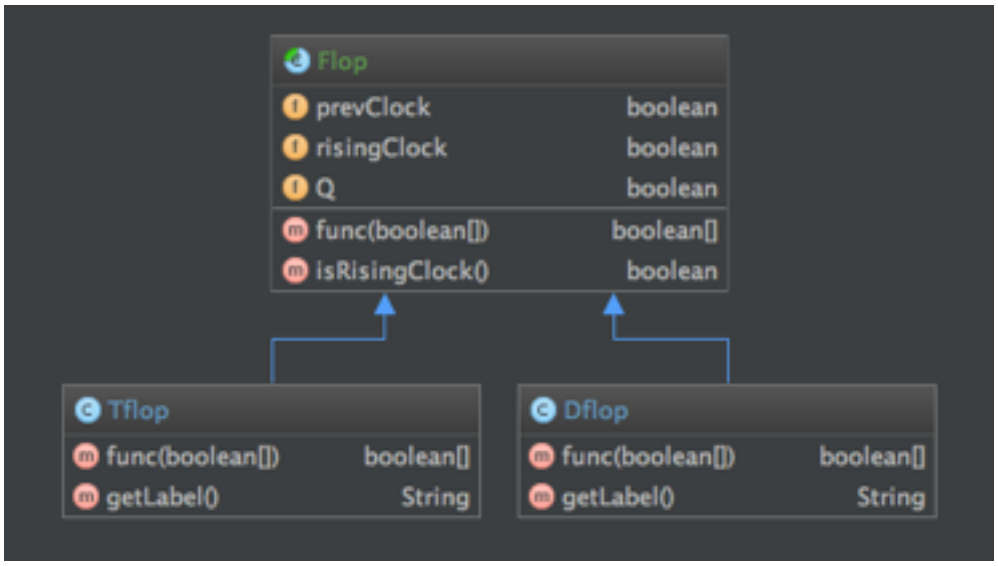
bort sig från den OutputSocket den är kopplad till är det bara att sätta connectedTo = null (se disconnect() i InputSocket). För att en OutputSocket ska kunna koppla bort sig från en InputSocket måste den be alla InputSocket:s den är kopplad till att ta bort relationen (se disconnect i OutputSocket).

6)

För att följa “best practice” gällande “single source of truth (SSOT)” har jag valt att spara de logiska signalerna i OutputSocket istället för att både ha dem i OutputSocket och InputSocket. Om du kollar i step-funktionen i klassen Logical kan du se att jag kallar på metoden getValue() på alla inputs på komponenten. Det ser ut som att värdet hämtas från inputen med det som egentligen händer är att inputen kontaktar outputen den är kopplad till och frågar den vilket signal-värde som ska användas. Detta ser du om du går in i implementationen av getValue() i InputSocket. Denna ökade komplexitet är som tur är något som är bra abstraherat och därför slipper jag tänka på det i step-funktionen när jag hämtar signal-värdena från komponentens inputs.

7)

Eftersom alla typer av flip flops behöver spara en bit i “state" och är intresserade av stigande klockpulser har jag valt att samla denna funktionalitet i en gemensam förälderklass.



I bilden kan man se att en bit av “state” sparas i variabeln Q. Det finns även en funktion som barnen kan kalla på kallad isRisingClock().

8)

Jag har valt att lägga till ett interface kallad StepListener som klassen SimComponent implementerar. I metoden setSimulation i klassen SimComponent kallar jag på simulation.addStepListener(this). Detta för att SimComponent ska få reda på när simulationen förändras och då rita om allt. Jag har valt att lösa detta med en observer istället för att skicka med SimComponent till Simulation och låta Simulation kalla repaint på SimComponent eftersom det betyder att klasserna skulle bli väldigt starkt kopplade till varandra. Detta gör också att jag kan skapa instanser av simulation i min ExampleFactory utan att ha tillgång till någon SimComponent.

9)

När jag lägger till en logisk komponent med metoden addLogical i klassen Simulation kallar jag på logical.setInteractionListener(this). Detta har jag gjort för att Clock, Button och Switch ska kunna meddela Simulation när en förändring sker. Jag har valt att inte låta barn till Logical kalla direkt på simulation.step() för att jag inte tycker att det är deras ansvar. Det är bättre att låta dem meddela Simulation att en förändring skett och låte Simulation själv avgöra vad som ska göras.

10)

I klassen Simulation finns det två listor, placeables och logicals. Det intressanta med dem är att listan logicals alltid är den delmängd till listan placeables. Detta eftersom Logical ärver från Placeable. Jag har valt att ha två listor på detta sätt för att i till exempel draw-metoden i Simulation kunna iterera alla komponenter för att rita dem, men i step itererar jag endast den delmängd av komponenter som ärver klassen Logical.

7. Användarmanual

För att bygga en krets kan du dra komponenter från listan till höger och placera dem på den gråa ytan till höger. Koppla ihop komponenter med varandra genom att dra i de svarta prickarna intill komponenterna för att skapa en signal-sladd. Det går bara att koppla inputs till outputs och vise versa. Det går inte att koppla inputs till inputs eller outputs till outputs. En input kan endast vara kopplad till en enda output. Men en output kan vara kopplad till flera inputs. Markera komponenter genom att klicka på dem och sedan trycka delete eller backspace. Du kan ändra på text-element genom att högerklicka på dem. Det går även att ändra klockfrekvensen genom att högerklicka på en klock-komponent.

Det första användaren möts av när den öppnar programmet är en lite guide så jag tror inte jag behöver förklara programmet mer här. Kolla gärna på alla exempel som går att välja i menyn för att få en känsla av vilka komponenter som finns och vilka typer av kretsar man kan bygga.

8. Slutliga betygsambitioner

Jag satsar på betyg 5. Jag tycker att min kod visar att jag kan använda och förstår många av de avancerade aspekterna av objektorienterad programmering. Jag har väldigt lite kod-duplicering, korta och tydliga metoder och smarta lösningar som gör att ganska avancerad funktionalitet ser simpel ut och är enkla att följa. Jag har korrekt felhantering, och har varken över- eller under-kommenterat koden, få magiska värden och en struktur som är modulär och enkel att bygga vidare på.

9. Utvärdering och erfarenheter

Jag tycker att allt har flutit på bra när det kommer till programmeringen. Jag hade gärna haft ytterligare ett redovisningstillfälle när halva tiden gått på kursen där man kan få visa hur långt man kommit. Jag tror att det hade varit en bra motivation att komma igång med projektet tidigare. Eftersom jag programmerat ganska mycket innan hade jag inte så mycket frågor till handledaren utan satt mest hemma och kodade.

Till nästa års elever skulle jag vilja ge tipset att inte skjuta upp rapportskrivandet till natten innan deadline.