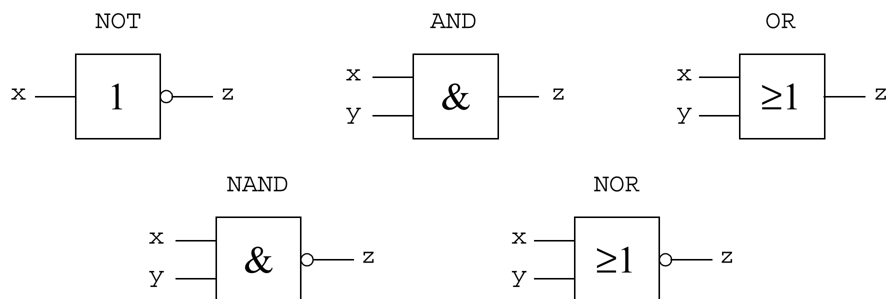
Laboration nr 4. Grindar

Avsikt

I denna laboration får du träna på att använda abstrakta klasser, arv, dynamisk bindning och exceptions. Dessutom får du läsa från filer och använda generiska containerklasser.

Bakgrund

När man konstruerar digitala system utgår man från enkla elektroniska kretsar som brukar kallas *grindar* (*gates*). Grindarnas in- och utsignaler är elektriska spänningar på "låg nivå" eller "hög nivå". Dessa brukar betecknas som 0 och 1 (false resp. true). De enklaste grindarna avbildar på elektronisk väg de logiska funktionerna "icke", "och" och "eller" och kallas därför INVERTERARE, OCH-grindar resp. ELLER-grindar (NOT gates, AND gates resp. OR gates.). Man använder också NAND-grindar och NOR-grindar. Dessa fungerar som OCH-grindar resp. ELLER-grindar fast med en inventerare på utgångssignalen. När digitala nät ritas används följande symboler för att beskriva de grundläggande grindarna. (`x` och `y` betecknar insignaler och `z` utsignaler.)



Uppgift 1 (obligatorisk)

Din första uppgift är att konstruera en grupp klasser som beskriver de grundläggande grindarna. Du skall skapa en abstrakt superklass `Gate` och utgå från denna. Klassen `Gate` skall ha en abstrakt subklass `BasicGate` vilken i sin tur skall vara superklass till de fem subklasserna `NotGate`, `AndGate`, `OrGate`, `NandGate` och `NorGate`. Dessutom skall du konstruera en klass `SignalGate` som beskriver konstanta insignaler. Denna klass skall vara direkt subklass till klassen `Gate`.

Klassen GateException

Deklarera en egen exception-klass `GateException`. Denna klass skall användas i fortsättningen för att rapportera sådana fel som uppstår i klassen `Gate` och dess subklasser. Låt klassen `GateException` vara en subklass till standardklassen `RuntimeException`. Klassen `GateException` skall ha en konstruktor med en `String` som parameter. Denna parameter skall användas för att skicka felmeddelanden.

Klassen Gate

Klassen `Gate` skall ha fyra instansvariabeler, `name`, som innehåller grindens namn, `outputValue`, som innehåller grindens aktuella utsignal (**`false`** eller **`true`**) samt två listor, `outputGates` och `inputGates`,

vilka skall innehålla referenser till de grindar (dvs. andra `Gate`-objekt) som är kopplade som input resp. output till den aktuella grinden. Observera att alla instansvariabler skall vara *privata*.

Klassen `Gate` skall ha följande instansmetoder:

- `init`, skall beräkna utsignalens startvärde och lägga detta värde i variabeln `outputValue`. (Initieringen kan inte göras i en konstruktor eftersom alla insignaler till grinden måste kopplas innan beräkningen kan ske.)
- `getName`, ger grindens namn som resultat.
- `setName`, ändrar grindens namn.
- `getOutputValue`, returnerar värdet av instansvariabeln `outputValue`.
- `setInputGate`, får som parameter en referens till en grind `g`, skall koppla grinden `g` som input till den aktuella grinden och den aktuella grinden som output till `g`.
- `getInputGates`, skall returnera en lista (av typen `List<Gate>`) med referenser till alla grindar vilkas utsignaler utgör insignaler till den aktuella grinden.
- `calculateValue`, en *abstrakt* metod som skall beräkna utvärdet för den aktuella grinden, utgående från insignalernas värden. Hur beräkningen går till beror förstås på vilken typ av grind det är fråga om. Det beräknade värdet skall ges som resultat.
- `inputChanged`, en *abstrakt* metod som man anropar för att tala om för grinden att någon av dess insignaler har ändrats. Metoden saknar parametrar och ger inget resultatvärde.
- `outputChanged`, en metod som man anropar för att tala om för grinden att dess utsignal skall ändras. Det nya värdet ges som parameter. Metoden `outputChanged` skall bara kunna anropas från klassen själv och dess subklasser. Metoden skall uppdatera instansvariabeln `outputValue` så att den kommer att innehålla det nya värdet. När detta har skett skall den tala om för alla grindar som har utsignalen från den aktuella grinden som input att signalen har ändrats.

Notera att ett par av metoderna skall vara abstrakta och därför inte implementeras i klassen `Gate`.

Senare i laborationen skall tidsfördröjningar läggas in. Därför skall klassen `Gate` dessutom ha en privat klassvariabel (statisk variabel), `delay`, av typen **`int`**. Det skall också finnas klassmetoderna `setDelay` och `getDelay` som man kan anropa för att ändra och avläsa värdet av variabeln `delay`.

Klassen BasicGate

Klassen `BasicGate` skall vara subklass till klassen `Gate`. I klassen `BasicGate` behöver man bara konstruera en enda metod:

- `inputChanged`, en implementering av den ärvda abstrakta metoden `inputChanged`. Den skall beräkna ett nytt utvärde för den aktuella grinden. Därefter skall den omedelbart tala om för grinden att dess utsignal skall ändras. (Metoden `inputChanged` blir alltså här mycket enkel, men skall göras mer komplicerad senare när tidsfördröjningar läggs in.)

Klasserna AndGate, OrGate, NotGate, NandGate och NorGate

Dessa klasser skall vara subklasser till `BasicGate`. De måste alla innehålla följande metod:

- `calculateValue`, en egen version av den ärvda abstrakta metoden `calculateValue`. Denna metod skall först kontrollera att den aktuella grinden har tillräckligt många insignaler. (För alla subklasser till `BasicGate`, utom för klassen `NotGate` gäller att det måste finnas minst två insignaler. För klassen `NotGate` gäller att det måste finnas exakt en insignal.) Om det visar sig att antalet insignaler inte

stämmer skall `calculateValue` signalera detta genom att generera en exception av typen `GateException`. Felmeddelandet skall innehålla grindens namn och en lämplig beskrivning av felet. Om antalet insignaler är korrekt skall `calculateValue` beräkna utvärdet för den aktuella grinden, utgående från insignalernas värden. Observera att metoden `calculateValue` *inte* skall ändra grindens utsignal. Den skall bara ge det beräknade värdet som resultat.

Klassen `NotGate` måste, förutom metoden `calculateValue`, även innehålla följande metod:

- `setInputGate`, en egen version av metoden `setInputGate` som överskuggar den version som ärvs från klassen `Gate`. Metoden skall, innan den kopplar in den nya grinden, kontrollera att man inte försöker koppla mer än en insignal. I så fall skall en exception av typen `GateException` genereras.

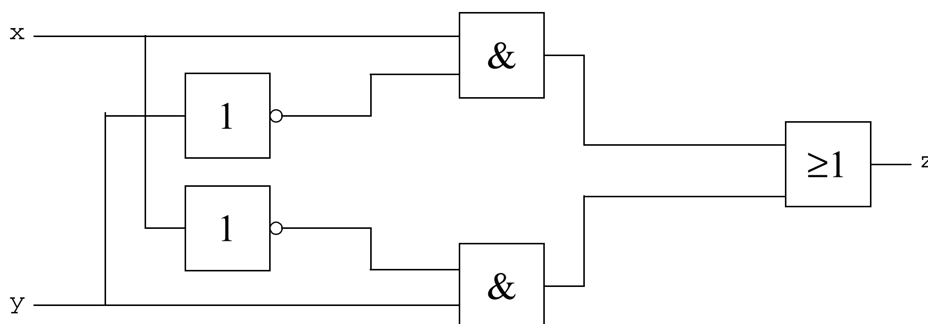
Klassen SignalGate

Klassen `SignalGate` skall vara subklass till klassen `Gate`. Den används för att beskriva konstanta insignaler till det nät man vill koppla upp. Man kan tänka sig en `SignalGate` som en "låtsasgrind" som saknar insignaler och som alltid ger samma utsignal. Klassen skall innehålla följande metoder:

- `setValue`, får en parameter (av typen **`boolean`**) som anger den konstanta signalens värde. Metoden skall tala om för grinden att dess utsignal skall ändras.
- `calculateValue`, egen version av den ärvda abstrakta metoden. Skall helt enkelt ge grindens utsignal som resultat.
- `setInputGate`, egen version av den ärvda metoden. Skall generera en felsignal av typen `GateException`, eftersom man inte får koppla någon insignal till en `SignalGate`.
- `inputChanged`, egen version av den ärvda metoden. Skall generera en felsignal av typen `GateException`, eftersom det inte finns några insignaler till en `SignalGate`.

Testkörning

Det finns tre färdiga huvudprogram med namnen `Gatetest1.java`, `Gatetest2.java`, och `Gatetest3.java`, som du kan använda för att testa dina klasser. Programmet `Gatetest1` simulerar nätet i följande figur, vilket bildar operationen *exclusive or*.



Programmet `Gatetest2` simulerar ett nät som bildar en en-bitars jämförare. Nätet har två insignaler `a` och `b` samt tre utsignaler `fa`, `fb` och `fe`. Om `a>b` skall `fa` bli 1 och de övriga utsignalerna 0, om `b>a` skall `fb` bli 1 och de övriga utsignalerna 0 och om `a=b` skall `fe` bli `1` och de övriga utsignalerna 0.

Nätet som programmet `Gatetest3` simulerar har tre ingångssignaler `a`, `b` och `c`. Om minst två av signalerna är 1 skall utsignalen `z` bli 1, annars skall den bli 0. Studera programmet och rita upp det nät det simulerar.

Uppgift 2 (obligatorisk)

`Gate`-klasserna skall naturligtvis kunna användas för att simulera olika nät. Det blir då lite klumpigt om man måste skriva ett nytt huvudprogram för varje nät. Därför skall vi i fortsättningen beskriva näten med hjälp av *konfigurationsfiler*. En konfigurationsfil är en textfil i vilken det på varje rad finns information om en grind. Först står grindens namn (ett namn man hittar på själv). Därefter står grindens typ (t.ex. `AND`, `not` eller `signal`). Sist på varje rad finns en lista med grindnamn. Dessa anger vilka grindar som skall kopplas som insignaler till den aktuella grinden. Det är också tillåtet att lägga in rader med kommentarer. Dessa inleds med tecknet `'*'` eller `'/'`. Studera som exempel konfigurationsfilen `xor.txt` vilken beskriver nätet i figuren ovan. Filen `har` utseendet:

```
** xor **
x SIGNAL
y SIGNAL
y' NOT y
x' NOT x
a1 AND x y'
a2 AND y x'
z  OR a1 a2
```

Din uppgift är nu att utöka klassen `Gate` med en klassmetod (statisk metod) `createGates` som läser en konfigurationsfil och som bygger upp det nät som denna fil beskriver. Metoden `createGates` får en parameter av standardtypen `File`. Denna parameter anger vilken fil som skall läsas. Som resultat skall metoden `createGates` ge ett värde av typen `Map<String,Gate>`. Det är alltså en avbildningstabell där nycklarna är grindarnas namn och värdena referenser till de `Gate`-objekt som beskriver grindarna. (*Tips*. Använd dig av klassen `LinkedHashMap` inne i metoden. När man löper igenom tabellen får man då grindarna i samma ordning som de ligger i filen.)

Om filen inte går att öppna skall en felsignal av typen `GateException` genereras. Du måste läsa igenom filen två gånger. (Öppna den på nytt vid andra genomläsningen.) Vid första genomläsningen skall alla `Gate`-objekten skapas och avbildningstabellen byggas upp. Då läser man grindarnas namn och typ från filen. Typen skall kunna skrivas både med stora och små bokstäver. När du skall skapa `Gate`-objekten är det *inte* tillåtet att ha en **`if`**-sats eller **`switch`**-sats där du testar vilken typ som stod i filen. (Det är inte tillräckligt flexibelt och tillåter inte att man lägger till nya `Gate`-klasser.) Du skall istället utnyttja att standardklassen `Class` innehåller ett par metoder som man kan använda för att skapa nya objekt av en viss klass om man vet klassens namn. Studera följande uttryck i vilket `className` är en variabel av typen `String`.

```
Class.forName(className).newInstance()
```

Om variabeln `className` t.ex. innehåller texten `"AndGate"` så skapar uttrycket ett nytt objekt av typen `AndGate`. (Resultatet från uttrycket är av typen `Object`, men man kan förstås göra en typomvandling innan man tilldelar resultatet till en variabel.) Om variabeln `className` innehåller namnet på en klass som inte finns eller om klassen inte har någon defaultkonstruktor genererar uttrycket ovan en exception. Detta kan man utnyttja för att kontrollera att det inte står något felaktigt typnamn i konfigurationsfilen. När man skapar `Gate`-objekten skall man också kontrollera att inte två eller flera `Gate`-objekt ges samma namn.

Vid andra genomläsningen av filen är alla `Gate`-objekten skapade och inlagda i avbildningstabellen. Uppgiften är då att koppla ihop grindarna på det sätt som beskrivs i filen.

Alla exceptions som genereras i metoden `createGates` skall, förutom själva felmeddelandet, också innehålla filens namn (det korta namnet) och radnumret för felet.

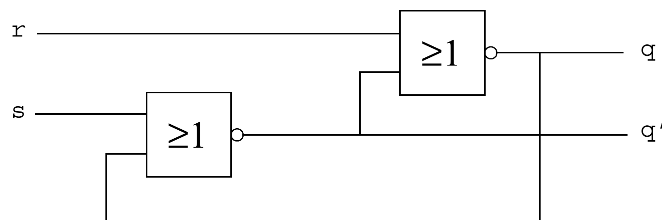
Testkörning

Det finns ett färdigt testprogram med namnet `GateProg.java` som du kan använda i fortsättningen. Detta program låter användaren välja en konfigurationsfil med hjälp av en fildialogruta och anropar sedan metoden `createGates`. Om konfigurationsfilen är korrekt och metoden `createGates` har byggt upp nätet på rätt sätt kan man i ett grafiskt fönster experimentera med olika insignaler och studera utsignalerna från grindarna i nätet. (Vill du bilda dig en uppfattning om hur det kan se ut när allt är färdigt kan du ladda ner filen GateProg-setup.exe och installera en demoversion av programmet.)

Provkör programmet genom att välja filerna `xor.txt`, `compare1.txt` och `nandtest.txt`. Dessa simulerar samma nät som programmen `Gatetest1`, `Gatetest2` och `Gatetest3` gjorde i uppgift 1. Provkör också med konfigurationsfilen `compare4.txt` som beskriver en 4-bitars jämförare. Provkör slutligen med filerna `fel1.txt`, `fel2.txt`, `fel3.txt`, `fel4.txt`, `fel5.txt` och `fel6.txt`. Dessa innehåller olika typer av fel vilka alla skall resultera i att felsignaler av typen `GateException` genereras.

Uppgift 3 (frivillig)

I de nät vi kopplat upp hittills har utsignalerna varit entydigt bestämda av insignalerna vid en viss tidpunkt. Sådana nät kallas *kombinatoriska nät*. Om man återkopplar utsignalerna i ett nät till ingångarna får man en annan typ av nät, s.k. *sekvensnät*. I dessa beror utsignalerna inte bara på de aktuella insignalerna, utan också på vilket tillstånd nätet befinner sig i sedan tidigare. Man kan använda sig av sådana nät för att åstadkomma en minnesfunktion. Som exempel kan vi studera följande nät som beskriver en s.k. SR-latch.



Om signalen `r` (reset) är 1 och `s` (set) är 0 kommer utsignalen `q` att få värdet 0. Om signalen `s` är 1 och `r` är 0 kommer istället utsignalen `q` att få värdet 1. Utsignalen `q'` kommer i båda dessa fall att få det inverterade värdet av `q`. Om både `r` och `s` är 0 hamnar nätet i ett stabilt tillstånd där `q` och `q'` behåller (minns) sina värden. Om både `r` och `s` är lika med 1 hamnar nätet i ett labilt tillstånd där utsignalerna är odefinierade. Denna kombination av insignaler är därför otillåten. En SR-latch används för att komma ihåg en binär siffra. Nätet i figuren finns beskrivet i konfigurationsfilen `srlatch.txt`. Provkör programmet `GateProg` med denna fil och studera vad som händer!

Att det inte fungerar beror på att dina `Gate`-klasser inte tar hänsyn till att grindarna kan vara återkopplade. Därför hamnar man i en situation där de olika `Gate`-objekten anropar varandra cirkulärt. Det uppstår en oändlig rekursion. Din uppgift är nu att göra de justeringar som behövs för att man skall kunna ha återkopplade nät. Samtidigt skall du lägga in tidsfördröjningar i grindarna. Detta måste man ha om man skall kunna simulera s.k. flanktriggade vippor, nät där man har klockpulser och där insignalerna endast kan påverka utsignalerna under klockpulsens positiva eller negativa flank.

Det behövs inga större förändringar. Det är framför allt metoden `inputChanged` i klassen `BasicGate` som behöver ändras. Hittills har den ju påverkat grindens utsignal omedelbart, men nu skall du lägga in en fördröjning. Börja med att lägga till en `Timer` i klassen `BasicGate` och låt objektet självt vara lyssnare. Tanken är att denna timer skall vara igång när det beräknade värdet håller på att överföras till grindens utsignal. När metoden `inputChanged` anropas skall den börja med att undersöka om timern redan är igång (metoden `isRunning` kan användas). Om så är fallet skall metoden `inputChanged` inte göra någonting.

Annars skall det nya utvärdet beräknas utgående från ingångarnas värden. Om det visar sig att det nya värdet avviker från den nuvarande utsignalens värde skall det nya värdet sparas, men ännu inte överföras till utsignalen. Istället skall då timern aktiveras så att den ger en avbrottssignal efter en viss fördröjning. Använd klassvariabeln `delay`. När avbrottet senare inträffar skall man stoppa timern och tala om för grinden att dess utsignal skall ändras.

Testkörning

Provkör på nytt programmet med konfigurationsfilen `srlatch.txt` och förvissa dig om att det fungerar nu.

Studera sedan filen `dlatch.txt`. Det nät den beskriver är en s.k. klockad D-latch. Det är ett nät som har två insignaler, en datasignal `d` och en klocksignal `c`. Det finns två utsignaler `q` och `q'`. `q'` skall alltid vara inversen av `q`. Datasignalen `d` överförs till `q` när klocksignalen `c` är 1. Om `c` är 0 kan `d` inte påverka utsignalen. D-latchen används därför för att lagra en binär siffra. Siffran som skall lagras ansluts till ingången `d` medan klocksignalen `c` är 0. Inskrivning i minnet sker sedan när `c` sätts till 1. Testkör programmet med konfigurationsfilen `dlatch.txt` och förvissa dig om att det fungerar så som beskrivits här.

Provkör också filen `dvipp.txt`. Den beskriver en flanktriggad D-vippa. Denna skall fungera på samma sätt om den klockade D-vippan, men ändringar i utsignalen kan bara ske då klocksignalen går från 0 till 1.

Provkör slutligen programmet med konfigurationsfilen `jkvipp.txt`. Denna beskriver förstås en s.k. JK-vippa. Även i denna kan utsignalen endast påverkas när klocksignalen växlar från 0 till 1. Om insignalerna J och K har värdena 0,0 ändras inte utsignalen, om de har värdena 1,0 sätts utsignalen till 1, om de har värdena 0,1 sätts utsignalen till 0 och om de har värdena 1,1 växlar utsignalen tecken.

Uppgift 4 (obligatorisk)

Studera programmet i filen `GateProg.java` och svara skriftligt på följande frågor.

1. Vad händer när användaren stänger ett fönster?
2. Vad är `inputMap` och `outputMap`? Vad används de till?
3. Vad händer när användaren klickar på en insignal för att ändra dess värde?
4. Hur gör programmet för att hela tiden hålla sig uppdaterat om vilka signaler som finns i nätet?
5. Vad händer när en exception av typen `GateException` inträffar?

Godkännande

När dina klasser fungerar ihop med testprogrammet för de olika konfigurationsfilerna skall dina programtexter lämnas in tillsammans med svaren i uppgift 4. För att laborationen skall bli godkänd räcker det inte med att programmet fungerar. Dina programfiler måste också vara skrivna på ett snyggt och begripligt sätt. Programraderna skall t.ex. indenteras (dras in) på det sätt som lärs ut i kursen.