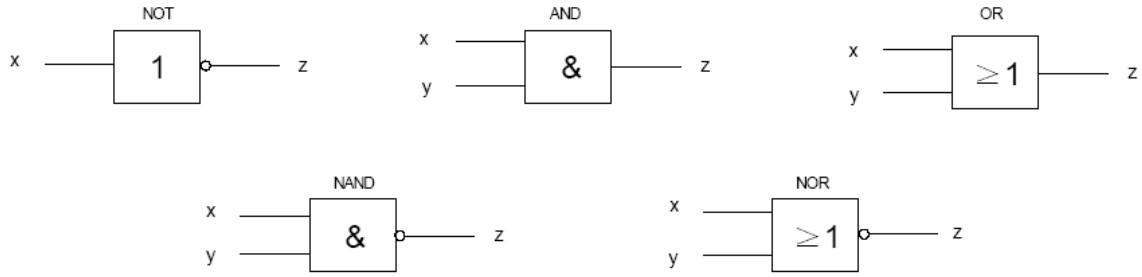
Laboration nr 4. Grindar

Avsikt

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

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" samt "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

När du löser denna uppgift är det lämpligt att rita ett klassdiagram som visar arvsrelationerna mellan klasserna.

Vi skall utgå från två gränssnitt som beskriver grundläggande operationer hos grindar. Gränssnittet `Gate` beskriver grindar som kan ha utsignaler:

```
public interface Gate {
    String getName();
    void setName(String name);
    void addOutput(InputGate g);
    boolean getOutputValue();
    void init();
}
```

Grindar som även har insignaler beskrivs av gränssnittet:

```
public interface InputGate {
    void addInput(Gate g);
    List<Gate> getInputs();
    void inputChanged();
}
```

Din första uppgift är att konstruera två klasser som modellerar grundläggande beteende hos grindar: `AbstractOutputGate` och `AbstractInputOutputGate`. Men först en undantagsklass:

`public class GateException`

Denna undantagsklass skall användas i fortsättningen för att rapportera fel som uppstår i grindklasserna. Låt `GateException` vara subklass till standardklassen `RuntimeException`. Klassen skall ha en konstruktor med en sträng som parameter. Denna parameter skall användas för felmeddelanden.

public class AbstractOutputGate **implements** Gate

Klassen skall ha instansvariablerna: `name`, som innehåller grindens namn, `outputValue`, som innehåller grindens aktuella utsignal (**false** eller **true**) samt en lista, `outputs`, vilken skall vara en lista innehållande referenser till de grindobjekt (av typen `InputGate`) som är kopplade till utgången på den aktuella grinden. Observera att *alla* instansvariabler *skall* vara privata.

Klassen skall ha följande instansmetoder:

- **public** `String getName()`
  ger grindens namn som resultat.
- **public void** `setName(String s)`
  ändrar grindens namn till `s`.
- **public void** `addOutput(InputGate g)`
  kopplar utsignalen hos denna grind till `g`.
- **public boolean** `getOutputValue()`
  returnerar värdet i instansvariabeln `outputValue`.
- **public void** `init()`
  beräknar utsignalens startvärde och lägger detta i variabeln `outputValue`. (Initieringen kan inte göras i en konstruktor eftersom alla insignaler till grinden måste kopplas innan beräkningen kan ske.)
- **protected void** `outputChanged(`**boolean** `newValue)`
  en metod som kan anropas från subklassernas metoder för att tala om för grinden att dess utsignal skall ändras. Det nya värdet ges som parameter. Metoden skall uppdatera instansvariabeln `outputValue` med det nya värdet. När detta har skett skall den tala om för alla grindar som har utsignalen från den aktuella grinden som insignal att värdet har ändrats.
- **public abstract boolean** `calculateValue()`
  en *abstrakt* metod som skall beräkna och returnera utvärdet för den aktuella grinden, utgående från insignalernas värden. Hur beräkningen går till beror förstås på vilken typ av grind det är och implementeras genom att överskugga metoden.

public class AbstractInputOutputGate
extends AbstractOutputGate
implements InputGate

Klassen skall ha en instansvariabel: `inputs`, vilken skall vara en lista innehållande referenser till de grindobjekt (av typen `Gate`) som är kopplade till ingångarna på den aktuella grinden. Senare i laborationen skall tidsfördröjningar läggas in. Därför skall klassen dessutom ha en privat klassvariabel (statisk variabel), `delay`, av typen **int**. Observera att variablerna *skall* vara privata. Variabeln `delay` skall initieras till t.ex. 100 ms;

Klassen skall ha följande metoder:

- **public void** `addInput(Gate g)`
  kopplar `g` som input till denna grind och denna grind som output hos `g`.
- **public** `List<Gate> getInputs()`
  returnerar en lista med alla grindar vilkas utsignaler utgör insignaler till den aktuella grinden.
- **public void** `inputChanged()`
  anropas för att tala om för grinden att någon av dess insignaler har ändrats.. Den skall beräkna ett nytt utvärde för den aktuella grinden. Därefter skall den omedelbart tala om för grinden att dess utsignal skall ändras. (Metoden blir alltså här mycket enkel, men skall göras mer komplicerad senare när tidsfördröjningar läggs in.)
- **protectd void** `checkInputs(`**int** `minInputs)`
  kontrollerar att grinden har minst `minInputs` insignaler. Om så inte är fallet kastas undantaget `GateException` med information om hur många som fattas. Metoden har skyddad access och skall endast

användas i subklasserna.

- **public static void** `setDelay(`**int** `ms)`
  sätter grindnätets tidsfördröjning i instansvariabeln `delay` till `ms`.
- **public static int** `getDelay()`
  returnerar grindnätets tidsfördröjning.

public class `AndGate` **extends** *lämplig basklass*,
public class `OrGate` **extends** *lämplig basklass*,
public class `NotGate` **extends** *lämplig basklass*,
public class `NandGate` **extends** *lämplig basklass*,
public class `NorGate` **extends** *lämplig basklass*

Dessa klasser måste alla överskugga metoden:

- **public boolean** `calculateValue()`
  metoden skall först kontrollera att den aktuella grinden har tillräckligt många insignaler. För alla utom `NotGate` gäller att det måste finnas minst två insignaler. För `NotGate` gäller att det måste finnas exakt en insignal. Om det visar sig att antalet insignaler inte stämmer skall `calculateValue` signalera detta genom att kasta ett undantag 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 `calculateValue` inte skall ändra grindens utsignal. Den skall bara ge det beräknade värdet som resultat.

Klassen `NotGate` måste, förutom metoden `calculateValue`, även överskugga metoden `addInput`. Metoden skall, innan den kopplar in den nya grinden, kontrollera att man inte försöker koppla mer än en insignal. I så fall skall `GateException` kastas.

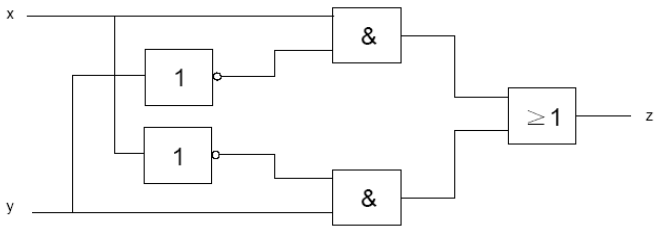
public class `SignalGate` **extends** *lämplig basklass*

Klassen används för att beskriva konstanta insignaler till det nät man vill koppla upp. Man kan tänka sig en signalgrind som en "låtsasgrind" som saknar insignaler och som alltid ger samma utsignal. Klassen skall ha metoderna:

- **public void** `setValue(`**boolean** `newValue)`
  får en parameter som anger den konstanta signalens värde. Metoden skall tala om för grinden att dess utsignal skall ändras till `newValue`.
- **public boolean** `calculateValue()`
  Skall helt enkelt ge grindens utsignal som resultat.

Testkörning

Det finns tre färdiga huvudprogram i `Gatetest1.java`, `Gatetest2.java`, och `Gatetest3.java`, som du kan använda för att testa dina klasser. `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

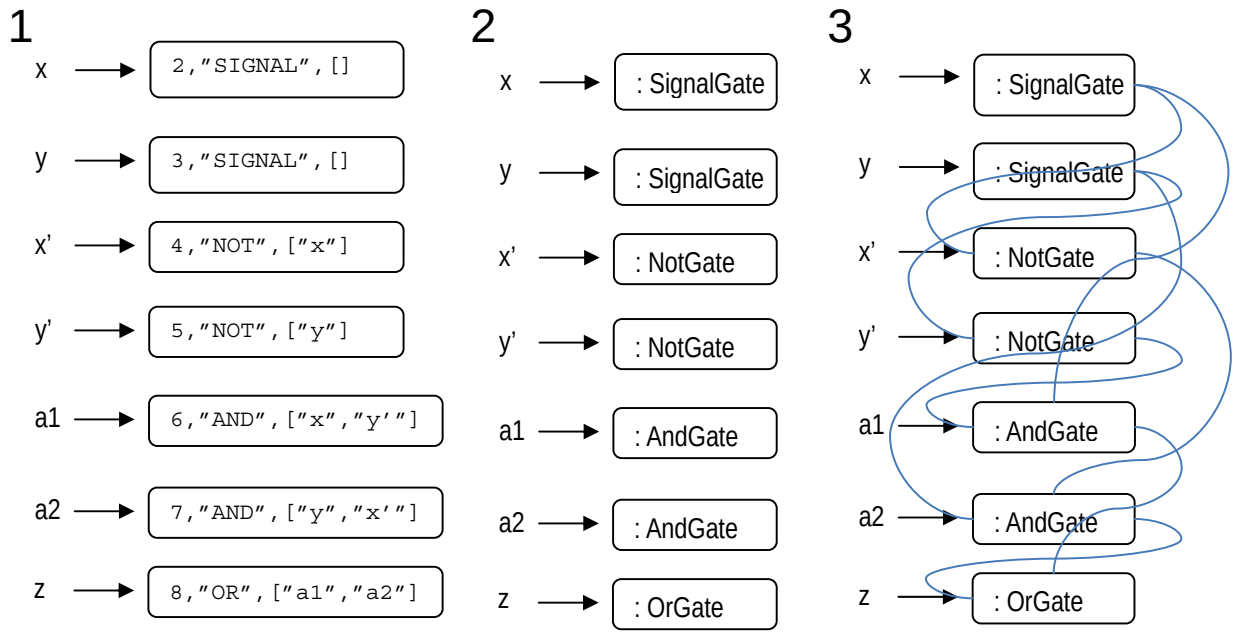
Grindklasserna skall naturligtvis kunna användas för att simulera olika nät. Det blir då lite klumpigt om man måste skriva ett nytt huvudprogram för varje nät. Därför skall vi i fortsättningen beskriva näten med hjälp av *konfigurationsfiler*. Programmet `CircuitViever`, vilket vi återkommer till, kan visualisera grindnät i ett fönster. För att möjliggöra detta måste vi först översätta konfigurationsfiler till grindnät i ett format som `CircuitViever` kan hantera.

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
x' NOT x
y' NOT y
a1 AND x y'
a2 AND y x'
z OR a1 a2
```

Att bygga ett grindnät från en sådan konfigurationsfil kan göras i tre steg:

1. Läs texten från filen och spara informationen i en *parsningstabell* som avbildar grindnamn på radnummer, grindtyp och insignaler. (Radnumren skall ingå i ev. felutskrifter.)
2. Upprätta en *grindtabell* som avbildar grindnamn på grindobjekt genom att genomlöpa parsningstabellen och ladda klasserna som motsvarar de aktuella grindtyperna. Ex. Om grindtypen ”`AND`” finns i tabellen skall klassen `AndGate` laddas och instansieras.
3. Genomlöp parsningstabellen igen och koppla ihop grindobjekten i grindtabellen enligt informationen om insignalerna i parsningstabellen.



Din uppgift är nu att färdigställa följande klass för inläsning av grindnät från konfigurationsfiler:

```
public class CircuitParser {
    private static class ParseData {
        int lineNo = 0;
        String gateType;
        List<String> inputNames;
    }

    public static Map<String,Gate> loadCircuit(File file) { ... }

    private static Map<String,ParseData> parseConfigFile(File file) { ... }

    private static Map<String,Gate> loadGateClasses(
                                        String fileName,
                                        Map<String,ParseData> parseTable) { ... }

     private static void connectGates(String fileName,
                                      Map<String,ParseData> parseTable,
                                      Map<String,Gate> gateTable) { ... }

    private static boolean isComment(String s) { ... }
}
```

Den interna klassen `parseData` används för att lagra inläst information (utom grindnamnet) från en rad i konfigurationsfilen. Eftersom klassen bara är synlig internt behöver inte instansvariablerna vara privata.

- **`public static`** `Map<String,Gate> loadCircuit(File file)`
  läser konfigurationsfilen `file` och bygger upp det nät som beskrivs i filen. Som resultat skall `loadCircuit` ge ett värde av typen `Map<String,Gate>`. Det är alltså en avbildningstabell där nycklarna är grindarnas namn och värdena referenser till `Gate`-objekt. Metoden skall lösa problemet enligt trestegsmetoden ovan genom att på lämpligt sätt utnyttja klassens privata metoder nedan.

- **`private static`** `Map<String,ParseData> parseConfigFile(File file)`

  läser konfigurationsfilen och skapar en parsningstabell, vilken sedan returneras. Tabellen skall ha en avbildning för varje rad i filen enligt punkt 1 ovan. Om filen inte går att öppna skall ett undantag av typen `GateException` kastas. Grindtypen skall kunna skrivas både med stora och små bokstäver. Vid läsningen av filen skall också kontrolleras att inte två eller flera grindar getts samma namn i filen. (Tips. Använd `LinkedHashMap`, vid genomlöpning av tabellen fås då grindarna i samma ordning som de ligger i filen.)

- **`private static`** `Map<String,Gate> loadGateClasses(`
  `String fileName,`
  `Map<String,ParseData> parseTable)`

  genomlöper `parseTable` och skapar en avbildningstabell med `Gate`-objekt , enligt punkt 2 ovan.
  När du skapar `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.) Du skall istället utnyttja att standardklassen `java.lang.Class` innehåller ett par metoder 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 `className` t.ex. innehåller texten `"AndGate"` så laddar `forName` klassen `AndGate` (från klassfilen `AndGate.class`) och därefter skapar `newInstance` ett nytt objekt av klassen. (Uttrycket har typen `Object`, så man får 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 default-konstruktor kastar uttrycket ovan ett undantag. Detta kan man utnyttja för att kontrollera att det inte står något felaktigt typnamn i

konfigurationsfilen. Parametern `fileName` innehåller namnet på konfigurationsfilen. Namnet skall ingå i eventuella felmeddelanden, se nedan.

- ```
  private static void connectGates(String fileName,
                                   Map<String,ParseData> parseTable,
                                   Map<String,Gate> gateTable)
  ```

  genomlöper `parseTable` och kopplar ihop grindarna i `gateTable` enligt steg 3 ovan.

- **`private static boolean`** `isComment(String s)`
  returnerar **`true`** om `s` innehåller en kommentar, annars **`false`**.

Alla undantag som kastas i `CircuitParser` skall vara av typen `GateException` och skall, förutom själva felmeddelandet, också innehålla filens namn och radnumret för felet. Exempel:

Felutskrift i `parseConfigFile`:

```
In file: jkvipp.txt, row 6: the name x is already used
```

Felutskrift i `loadGateClasses`:

```
In file: xortest.txt, row 3: unknown circuit type: ALLAN
```

Felutskrift i `connectGates`:

```
In file: logic.txt, row 6: unknown circuit name: tjoho
```

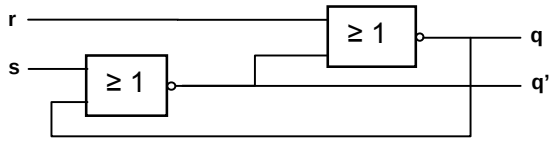
Testkörning

Det finns ett färdigt testprogram med namnet `CircuitViever.java` som du kan använda i fortsättningen. Detta program låter användaren välja en konfigurationsfil med hjälp av en fildialog och anropar sedan `loadCircuit`. Om konfigurationsfilen är korrekt och `loadCircuit` har byggt upp nätet på rätt sätt kan man i ett grafiskt fönster experimentera med olika insignaler och studera utsignalerna från grindarna i nätet.

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`, `fel6.txt` och `fel7.txt`. Dessa innehåller olika typer av fel vilka alla skall resultera i att undantaget `GateException` kastas.

Uppgift 3

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 `CircuitViewer` 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 `AbstractInputOutputGate` 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 `AbstractInputOutputGate` 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 metoden `setInitialDelay` i timer-klassen för att sätta fördröjningen. 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. Jkvippa. 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.

Godkännande

Källkoden för de klasser som utvecklats av labgruppen skall skickas in via Fire. Koden skall vara välstrukturerad, använda lämpliga namn, vara snyggt redigerad och innehålla lämpliga kommentarer.