Dagens tema:

Kodegenerering

- Array-er
- Funksjoner og kall
- Hovedprogrammet
- Noen siste gode råd

Redigeringsverktøy

- Emacs
- Eclipse

Versjonskontroll

- CVS og Subversion

Array-er

Deklarasjon av array-er

Når programmet deklarerer en array, må kompilatoren vår sette av plass til det oppgitte antallet elementer.

```
int x;
int a[10];
int v;
```

gir følgende kode:

```
Code     5:                     RES                      1  # int x
Code     6:                     RES                     10  # int a[10]
Code    16:                     RES                      1  # int v
```

Generelt om variable

I vår kompilator får *alle* variable fast plass i minnet.

+ Det forenkler kodegenereringen.
+ Det gjør det enklere å finne feil.
– Vi kaster bort en del plass.
– Lengden av array-er må være kjent under kompileringen.
– Vi kan ikke ha rekursive funksjoner.

Oppslag i array-er

Oppslag i array-er er enkelt takket være fleksibiliteten i Rask-koden.

⟨v⟩[⟨n⟩]	⇒	SET R2,R0,⟨n⟩ LOAD R1,R2,Adr(⟨v⟩)
⟨v⟩[⟨v_2⟩]	⇒	LOAD R2,R0,⟨v_2⟩ LOAD R1,R2,Adr(⟨v⟩)

Men hva med **a[-1]**? Er det lov?
Hva med **a[v+1]** eller **a[a[1]]**?

Det er like enkelt for tilordning til array-elementer:

⟨v⟩[⟨n⟩] = ⟨e⟩;	⇒	⟨Beregn ⟨e⟩, svar i R_1⟩ SET R2,R0,⟨n⟩ STORE R1,R2,Adr(⟨v⟩)
⟨v⟩[⟨v_2⟩] = ⟨e⟩;	⇒	⟨Beregn ⟨e⟩, svar i R_1⟩ LOAD R2,R0,Adr(⟨v_2⟩) STORE R1,R2,Adr(⟨v⟩)

```
a[x] = -2;
v = a[10] + 1;
```

genererer følgende kode

```
Code   21:  201000000000002  SET    1  0    2  #       2
Code   22:  501000000000001  SUB    1  0    1  # R1 = -
Code   23:  102000000000005  LOAD   2  0    5  # R2 = x
Code   24:  301020000000006  STORE  1  2    6  # a[...] =
Code   25:  202000000000010  SET    2  0   10  # R2 = 10
Code   26:  101020000000006  LOAD   1  2    6  # a[...]
Code   27:  203010000000000  SET    3  1    0  # Save operand i
Code   28:  201000000000001  SET    1  0    1  # R1 = 1
Code   29:  401030000000001  ADD    1  3    1  #   +
Code   30:  301000000000016  STORE  1  0   16  # v =
```

Legg merke til at vi *ikke* sjekker om indeksen er lovlig.

Funksjoner

Den enklest mulige funksjonen (og et kall på den) ser slik ut:

```
int f() { }
int main () { f(); }
```

Den genererte koden ser slik ut:

```
Code    3:                  RES                      1  # Return address in f
Code    4:                  RES                      1  # Refuge for R3 in f
Code    5:  331000000000003 STORE    31  0           3  # Save return address
Code    6:  303000000000004 STORE     3  0           4  # Save R3
Code    7:  103000000000004 LOAD      3  0           4  # Restore R3
Code    8:  131000000000003 LOAD     31  0           3  # Restore return address
Code    9: 1700000000000000 RET       0  0           0  # Return from f

Code   14: 1600000000000005 CALL      0  0           5  # Call f(...)
```

Vi finner følgende elementer:

Kallet

Kallet skjer enkelt ved å generere en CALL. Adressen er alltid kjent. (Hvorfor?)

Start av funksjonen

Når funksjonen kalles, ligger returadressen i R_{31}-registeret. Vi må ta vare på den (hvorfor?) og trenger en egen variabel til det.

Register R_3

Vi må også ta vare på R_3. (Hvorfor? Hvorfor ingen andre registre?)

Avslutning av funksjonen

Et tilbakehopp må skje til den lagrete returadressen.

```
LOAD    R3,R0,⟨lagret R3⟩
LOAD    R31,R0,⟨lagret returadresse⟩
RET
```

Parametre

Vi kan gi funksjonen 0–4 parametre:

```
int fa (int aa)
{ }

int fd (int da, int db, int dc, int dd)
{ }

int main ()
{
   fa(1);  fd(1, 2, 3, 4);
}
```

Konvensjonen sier at parameterens verdi skal overføres i registrene R_{11}–R_{14}.[1]

Kall Ved kallet må vi sørge for at parametrene er i de riktige registrene.

Start Når funksjonen starter, må den legge parametrene i egne variable. (Hvorfor?)

Underveis Mens koden i funksjonen utføres, er parametrene helt vanlige variable.

Avslutning —

[1] Denne formen for parameteroverføring kalles *verdioverføring*.

Resultatverdi

Funksjoner har også en resultatverdi:

```
int pi ()
{
   return 3;
}

int main ()
{
   int a;   a = pi();
}
```

Resultatverdien skal overføres i R_1-registeret.

Kall —

Start —

Underveis return-setningen gjør to ting:

1. Beregner uttrykket og legger svaret i R_1-registeret
2. Hopper til slutten av funksjonen (Denne hoppadressen er foreløbig ukjent og må oppdateres senere.)

Etterpå Resultatverdien ligger i R_1-registeret hvor den skal være.

Hopp til hovedprogrammet

Hovedprogrammet er funksjonen main. Den *skal* ha 0 parametre.

Rask starter alltid eksekveringen i adresse 0. Instruksjon 0 vil derfor alltid være et kall på hovedprogrammet etterfulgt av et kall på exit(0) (hvorfor?).

Siden adressen er ukjent før alle funksjonene er kompilert, må adressen settes inn senere. (Nærmere forklaring ble gitt forrige uke.)

Et siste eksempel

```
int pot2 (int x) {
   int p2;   p2 = 1;
   while (2*p2 <= x) { p2 = 2*p2; }
   return p2;
}

int x;

int main () {
   int v;   v = getint();
   x = pot2(v);   putint(x);   putchar(10);
}
```

Programmet kjøres slik:

```
$ rusc -logC demo.rusc
$ rask demo.rask
200
128
```

```
Code    0: 1600000000000000  CALL      0  0          0  # main();
Code    1:  211000000000000  SET      11  0          0  #     (0)
Code    2: 1600000000009990  CALL      0  0       9990  # exit
Code    3:                   RES                    1  # Return address in pot2
Code    4:                   RES                    1  # Refuge for R3 in pot2
Code    5:                   RES                    1  # int x
Code    6:                   RES                    1  # int p2
Code    7:  331000000000003  STORE    31  0          3  # Save return address
Code    8:  303000000000004  STORE     3  0          4  # Save R3
Code    9:  311000000000005  STORE    11  0          5  # Save parameter x
Code   10:  201000000000001  SET       1  0          1  # R1 = 1
Code   11:  301000000000006  STORE     1  0          6  # p2 =
Code   12:  201000000000002  SET       1  0          2  # R1 = 2
Code   13:  203010000000000  SET       3  1          0  # Save operand in R3
Code   14:  101000000000006  LOAD      1  0          6  # R1 = p2
Code   15:  601030000000001  MUL       1  3          1  #   *
Code   16:  203010000000000  SET       3  1          0  # Save operand in R3
Code   17:  101000000000005  LOAD      1  0          5  # R1 = x
Code   18: 1101030000000001  LESSEQ    1  3          1  #   <=
Code   19: 1401000000000000  JUMPEQ    1  0          0  # Exit while-loop when test is fals
Code   20:  201000000000002  SET       1  0          2  # R1 = 2
Code   21:  203010000000000  SET       3  1          0  # Save operand in R3
Code   22:  101000000000006  LOAD      1  0          6  # R1 = p2
Code   23:  601030000000001  MUL       1  3          1  #   *
Code   24:  301000000000006  STORE     1  0          6  # p2 =
```

```
Code   25: 1400000000000012  JUMPEQ    0  0          12  # Repeat while-loop
--->   19: 1401000000000026                          26  # Update break address
Code   26:  101000000000006  LOAD      1  0           6  # R1 = p2
Code   27: 1400000000000000  JUMPEQ    0  0           0  # return
--->   27: 1400000000000028                          28  # Update address of return jump
Code   28:  103000000000004  LOAD      3  0           4  # Restore R3
Code   29:  131000000000003  LOAD     31  0           3  # Restore return address
Code   30: 1700000000000000  RET       0  0           0  # Return from pot2
Code   31:                   RES                     1  # int x
Code   32:                   RES                     1  # Return address in main
Code   33:                   RES                     1  # Refuge for R3 in main
Code   34:                   RES                     1  # int v
Code   35:  331000000000032  STORE    31  0          32  # Save return address
Code   36:  303000000000033  STORE     3  0          33  # Save R3
Code   37: 1600000000009992  CALL      0  0        9992  # Call getint(...)
Code   38:  301000000000034  STORE     1  0          34  # v =
Code   39:  111000000000034  LOAD     11  0          34  # R11 = v
Code   40: 1600000000000007  CALL      0  0           7  # Call pot2(...)
Code   41:  301000000000031  STORE     1  0          31  # x =
Code   42:  111000000000031  LOAD     11  0          31  # R11 = x
Code   43: 1600000000009994  CALL      0  0        9994  # Call putint(...)
Code   44:  211000000000010  SET      11  0          10  # R11 = 10
Code   45: 1600000000009993  CALL      0  0        9993  # Call putchar(...)
Code   46:  103000000000033  LOAD      3  0          33  # Restore R3
Code   47:  131000000000032  LOAD     31  0          32  # Restore return address
Code   48: 1700000000000000  RET       0  0           0  # Return from main
--->    0: 1600000000000035                          35  # Fix call to main()
```

Noen siste gode råd

Basert på egne og andres (til dels bitre) erfaringer mener jeg dere bør:

- diskutere ideer og løsningsforslag med hverandre.
- lese kompendiet, ukeoppgavene og lysarkene fra forelesningene.
- utvide programmet i små steg som testes godt før dere går videre.
- skrive mange små testprogrammer i RusC.
- bruk standard rusc som fasit.
- spørre gruppelærerne.
- ta det helt rolig om dere blir innkalt til samtale om prosjektet.

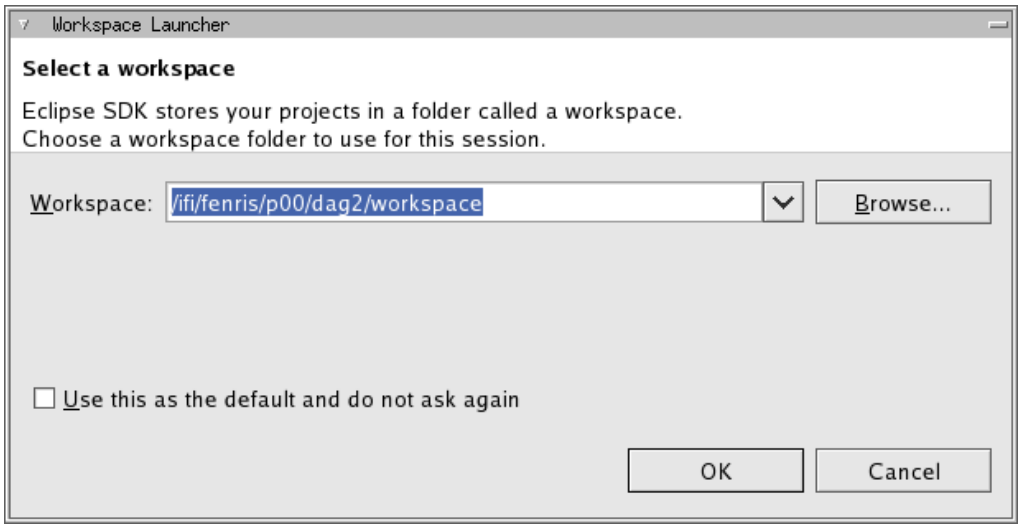
Hvilket programmeringsverktøy?

Det finnes to typer verktøy:

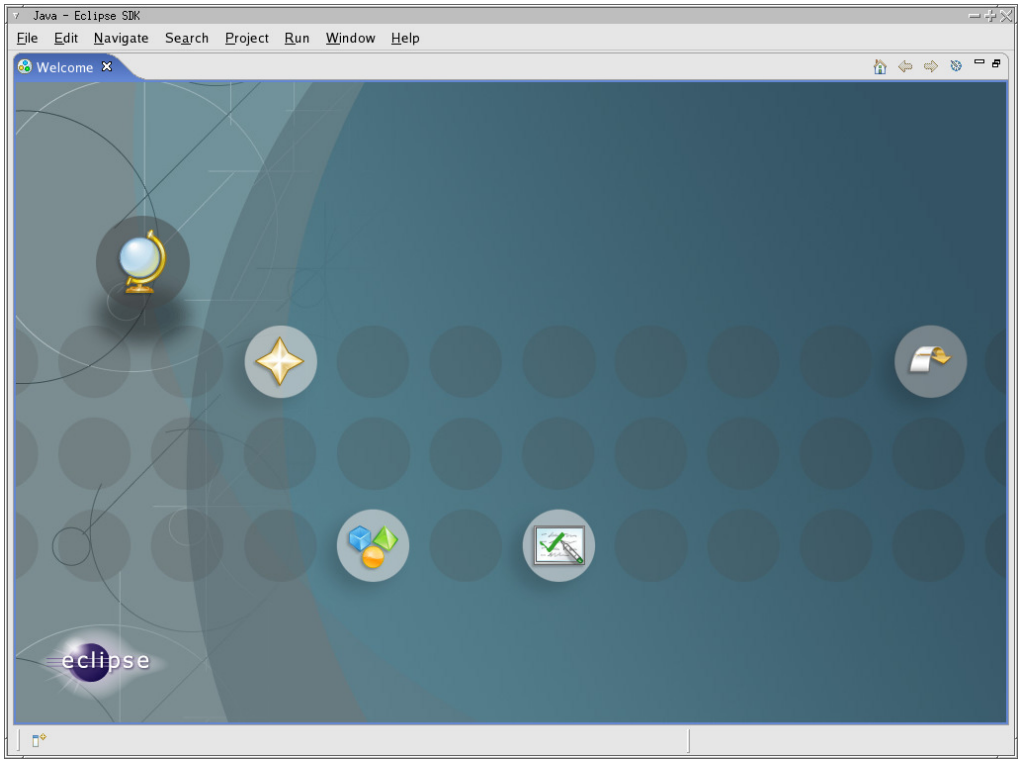
- **Generelle verktøy** (som Emacs) kan håndtere alle programmeringsspråk.
- **Spesialverktøy** (som Eclipse) er laget for ett programmeringsspråk (og ofte bare for én omgivelse).

Oppstart av Eclipse:

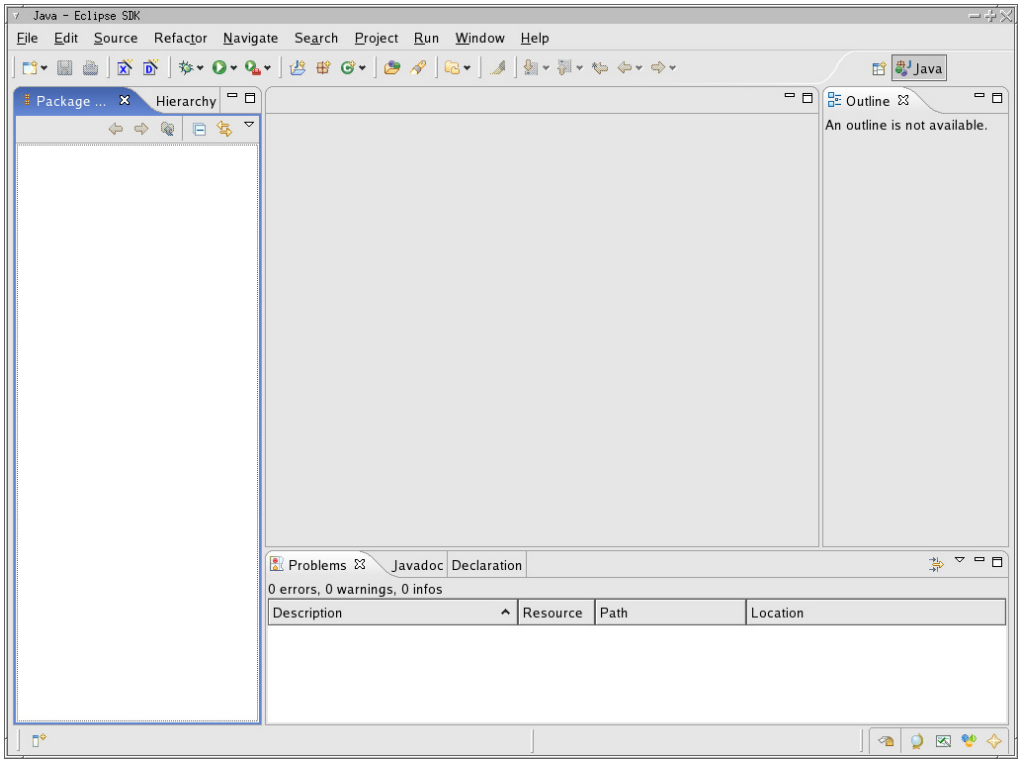
> eclipse&



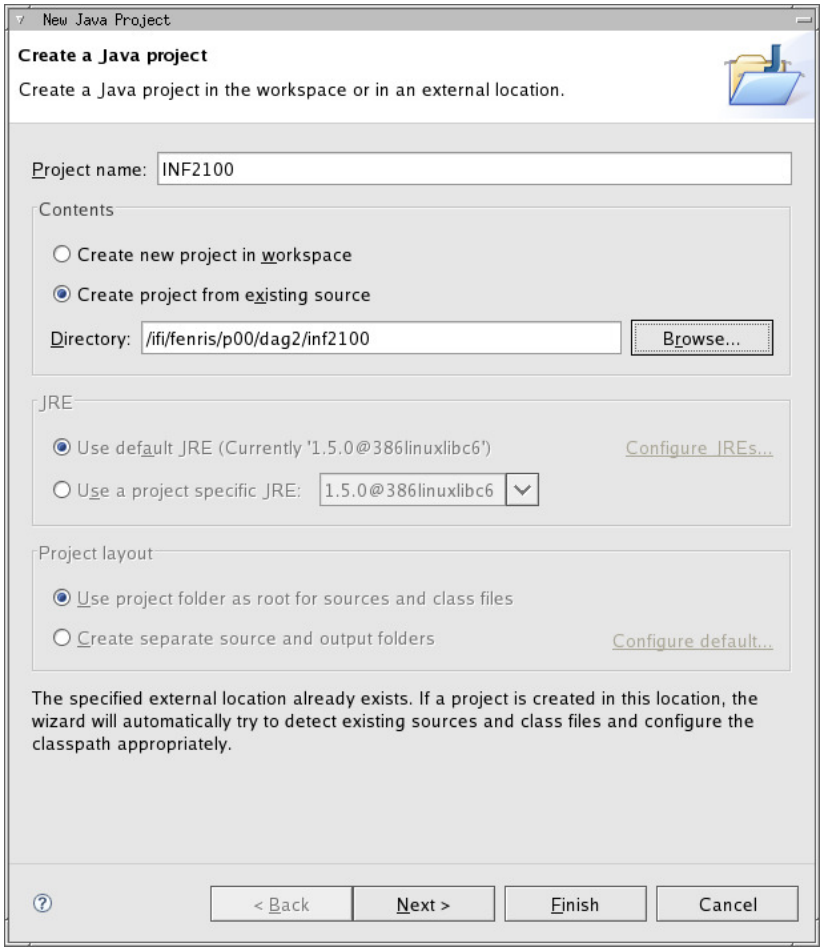
Velkomstvinduet i Eclipse



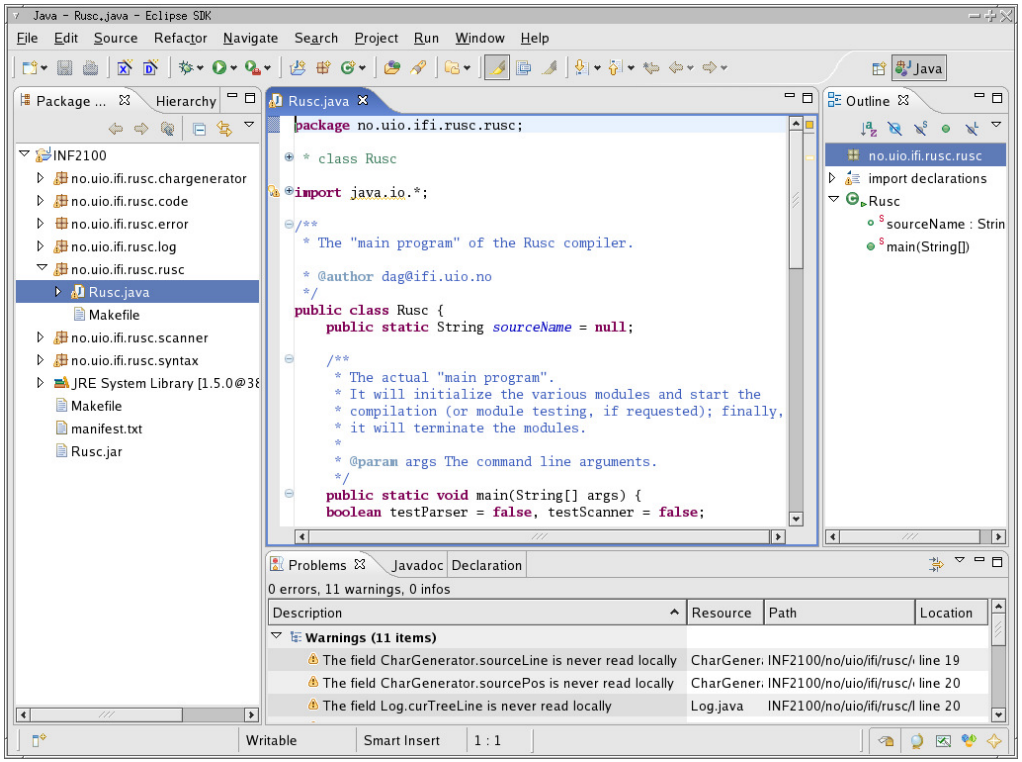
Eclipse-vinduet



Start et nytt prosjekt



Arbeid med Eclipse



Det er mange fordeler med et spesialverktøy:

- Mange liker å jobbe med «pek-og-klikk».
- Verktøyet vil oppdage feil veldig tidlig.
- Mye kode settes inn automatisk.
- Ved feil under testing får man øyeblikkelig vite hvor feil oppsto.

Hvorfor bruker da ikke alle spesialverktøy?

- Det er ikke laget spesialverktøy for alle språk og omgivelser.
- Spesialverktøy er grafiske – det er ikke alltid man kan kjøre slike programmer.
- Det er mye jobb å lære et nytt grafisk verktøy for hvert språk man skal jobbe i.
- Man jobber ofte raskere med tastaturet enn med musen.
- Noen nyttige operasjoner finnes ikke i spesialverktøyene (bytt to tegn, bytt om variabelnavn, . . .).

Konklusjon

Alle bør prøve begge typen verktøy.

Så velger man det man foretrekker til hvert prosjekt.

Når flere samarbeider

Når flere jobber sammen, kan man tråkke i beina på hverandre:

1. Per tar en kopi av en kildefil og begynner å rette på den.
2. Kari gjør det samme.
3. Kari blir første ferdig og kopierer filen tilbake.
4. Per blir ferdig og kopierer filen tilbake. Karis endringer går tapt.

Løsningen

Et *versjonskontrollsystem* er løsningen.

De fleste slike systemer er *utsjekkingssystemer* basert på *låsing*:

1. Per ber om og *sjekker ut* (dvs får en kopi av) filen og begynner å rette på den.
2. Kari ber om en kopi, men får den ikke fordi den er *låst*.

Først når Per er ferdig og *sjekker inn* filen, kan Kari få sin kopi.

Fordeler med et slikt utsjekkingssystem:

- Lettforståelig.
- Ganske sikkert.

  (Men hva om Per og Kari begge må rette i to filer? Da kan de starte med hver sin fil, men når de er ferdige med den første filen kan systemet inneholde feil.)
- Kari bør kunne få en lese-kopi selv om Per jobber med filen. (Mange systemer tillater det, men ikke alle.)

Ulemper:

- Hva om Per glemmer å legge tilbake filen?
- Det burde vært lov for Per og Kari å jobbe på ulike deler av filen samtidig.

Insjekkingssystemer

En bedre løsning er *innsjekkingssystemer*:

- Alle kan nårsomhelst sjekke ut en kopi.
- Ved innsjekking kontrolleres filen mot andre innsjekkinger:
  - Hvis endringene som er gjort ikke er i konflikt med andres endringer, *blandes* endringene med de tidligere.
  - Ved konflikt får brukeren beskjed om dette og må manuelt klare opp i sakene.

Et scenario

1. Per sjekker ut en kopi av en fil. Han begynner å gjøre endringer i slutten av filen.
2. Kari sjekker ut en kopi av den samme filen. Hun endrer bare i begynnelsen av filen.
3. Per sjekker inn sin kopi av filen.
4. Kari sjekker inn sin kopi, og systemet finner ut at de har jobbet på hver sin del. Innsjekkingen godtas.

Når man er alene

Selv om du jobber alene med et prosjekt, kan det være svært nyttig å bruke et versjonskontrollsystem:

- Man kan enkelt finne frem tidligere versjoner.
- Det kan hende man jobber på flere datamaskiner.
- Noen ganger lager man flere versjoner av et program, for eksempel tilpasset ulike operativsystemer. Men ofte er 90% eller mer av koden felles.

CVS og Subversion

Det mest kjente innsjekkingssystemet er **CVS** («Concurrent Versions System») laget i 1986 av *Dick Grune*. Det er veldig mye brukt i Unix-miljøer.

For å bøte på noen svakheter i CVS laget firmaet *CollabNet* **Subversion** i 2000.

Gratis implementasjoner finnes for alle vanlige operativsystemer; se http://subversion.tigris.org/.

Nære og fjerne systemer

Subversion kan operere på to måter:

- Alt skjer i det lokale filsystemet.
- Man kan starte en Subversion-tjener på en maskin og så sjekke inn og ut filer over nettet.

Vi skal gjøre det siste og bruke Ifis Subversion-tjener.

Opprette et *repository*

1. Gå inn på nettsiden

   https://wwws.ifi.uio.no/system/svn/

2. Logg inn.
3. Velg «My repositories» og «Create new repository». (I dette eksemplet heter det Hallo.)

   (Alle kan lage inntil tre «repositories».)
4. Hvis det er flere på prosjektet, velg «Edit user access».

Legge inn filer

Så kan vi legge inn mapper. La oss lage en *gren* med mappen Hei-1 som inneholder filen Hello.java:

```
$ cd Hei-1
$ svn import https://sub.ifi.uio.no/repos/users/dag-Hallo -m "2100demo"
Adding         Hei/Hello.java

Committed revision 1.
```

Opsjonen -m gir en kort beskrivelse av denne grenen.

Sjekke ut filer

Nå kan vi (for eksempel fra en annen mappe) hente ut mappen vår:

```
$ svn co https://sub.ifi.uio.no/repos/users/dag-Hallo
A  dag-Hallo/Hello.java
Checked out revision 1.
$ ls -l
drwxr-xr-x    3 dag      ifi-a        4096 2007-11-13 06:46 dag-Hallo
$ ls -la dag-Hallo
total 16
drwxr-xr-x    3 dag      ifi-a        4096 2007-11-13 06:46 .
drwxr-xr-x    3 dag      ifi-a        4096 2007-11-13 06:46 ..
drwxr-xr-x    6 dag      ifi-a        4096 2007-11-13 06:46 .svn
-rw-r--r--    1 dag      ifi-a         500 2007-11-13 06:46 Hello.java
```

Sjekke inn filer

Etter at filen er endret, kan vi sjekke den inn igjen:

```
$ svn commit -m"Enklere kode"
Sending        Hello.java
Transmitting file data .
Committed revision 2.
```

Vi behøver ikke nevne hvilke filer som er endret — det finner Subversion ut selv. (Etter første utsjekking inneholder mappen skjulte opplysninger om repository-et, så det trenger vi ikke nevne mer.)

Andre nyttige kommandoer

svn update . henter inn eventuelle oppdateringer fra repository.

svn info viser informasjon om mappen vår:

```
$ svn info
Path: .
URL: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository Root: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository UUID: 8c927215-bc3e-0410-a56f-b2451114731f
Revision: 2
Node Kind: directory
Schedule: normal
Last Changed Author: dag
Last Changed Rev: 2
Last Changed Date: 2007-11-13 07:02:16 +0100 (Tue, 13 Nov 2007)
```

svn diff viser hvilke endringer som er gjort:

```
$ svn diff -r 1:2
Index: Hello.java
===================================================================
--- Hello.java  (revision 1)
+++ Hello.java  (revision 2)
@@ -7,10 +7,9 @@
     Properties prop = System.getProperties();
     String versjon = prop.getProperty("java.version"); // Versjonen
     String koding = prop.getProperty("file.encoding"); // Koding
-    String hei;
+    String hei = "Hallo";

-    hei = "Hallo";
-    hei = hei + ", alle sammen!";
+    hei += ", alle sammen!";
     System.out.println(hei);
     System.out.println("Dette er versjon " + versjon);
     System.out.println("Kodingen er " + koding);
```

Om dere likte kurset

har dere flere kurs innen samme felt:

INF2270 (tidligere INF1070) lærer dere om hvordan en datamaskiner er bygget opp og å programmere Intel maskinspråk

INF3110 introduserer dere for mange flere programmeringsspråk (og litt mer om kompilering)

INF5110 gir en grundig innføring i hvordan man skriver en ekte kompilator