I-120 (h-99)

Foreleser: Michal Walicki
e-mail: michal@ii.uib.no
kontor: HiB, 4126

Gruppeleder: Cato Ervik
e-mail: cato@ii.uib.no

I-120 webside: http:// www.ii.uib.no / undervisning / kurs / i120/
Undervisning
Kurs dette semesteret
I120 - Kursside

Pensum:

- "Data Structures and Algorithms using JAVA"
  M.T.Goodrich & R.Tamassia
  (deler)

- ev. notater delt ut på forelesning

Gruppeøvelser:

fredag, kl.10:15 : HER – starter DENNE uken

– gjennomgang av oppgaver/pakker, noen løsningsforslag

i tillegg – aktivitets- og diskusjonsgrupper for *spesielt interesserte*

i120 web-siden:

http: // www.ii.uib.no / undervisning / kurs / i120

– transparenter: **.pdf (2x1)** format

– ukeoppgaver : **.html** format

i120 pakker:

/ local / pub / kurs / i120 / pakker - legg till CLASSPATH

tidligere godkjente Oblig'ene:

– kan brukes (man må bare melde seg til eksamen)

Pensum: fra boken (H-99, foreløpig)

	unntatt	kursorisk	poenger
KAP. 1	1.4.2		**OO, ABSTRAKSJON ...**
KAP. 2			**O-NOTASJON**
KAP. 3	3.2.4, 3.5	**3.1.3, 3.2.3, 3.4**	**STABEL, KØ, LISTE, ADAPTER**
KAP. 4			**SEKVENS; RANK, POSITION; ENUMERATION-ITERATOR**
KAP. 5	5.5	**5.4.4**	**TRÆR, B-TRÆR + BSF/DFS (PRE/POST/IN-ORDER)**
KAP. 6		**6.3.4, 6.4**	**PRIORITETSKØ, HEAP; TOTALORDNING/COMPARATOR**
KAP. 7	7.4, 7.5	**7.6.2, 7.6.3, 7.7**	**ORDBOK, BST, HASHTAB**
KAP. 8	8.1.3, 8.2, 8.6	**8.4, 8.5**	**QUICKSORT (MERGESORT); RANDOMISERING**
KAP. 9		**9.4.5**	**GRAF, DIGRAF, DFS/BFS, FW**
KAP. 10	10.2.2-10.2.4, 10.3		**SS-SP** (DIJKSTRA,BELLMAN-FORD)**, MST** (KRUSKAL)

Mål og Mening

Ikke programmering (i-110)

men effektiv og korrekt programmering
dvs algoritmer og strukturering

type tekst/program	boksider	tekstlinjer	listelengde
Wolfe 'Forfengelighetens fyrverkeri'	661	26 440	132m
Tolkien 'Hobbiten+Ringtrilogien+Simarillion'	1 847	73 880	222m
enkel kompilator	300	12 000	36m
lønnssystem, industri	600	24 000	80m
utlånssystem, forsikring	7 500	300 000	900m
abonnentsystem, avis	10 650	425 000	1 275m
logistikksystem, oljeplatform	18 750	750 000	2 250m

program	**mengde data :**	**n**	**n*10**
effektiv		**1 min**	**10 min**
lite effektiv		**1 min**	**1 000 min**

Programmereres aktivitet

år	nye prosjekter	vedlikehold, forbedring	vedlikehold / total
1950	90	10	10%
1960	8 500	1 500	15%
1970	65 000	35 000	35%
1980	1 200 000	800 000	40%
1990	**3 000 000**	**4 000 000**	**57%**
2000	4 000 000	6 000 000	60%
2010	5 000 000	9 000 000	64%
2020	7 000 000	14 000 000	66%

1. Typing og Abstraksjon

10101010000101010000101000101010100.......

1. **LD**(r1,1000) 2. **LOAD**(r2,r1) 3.**MIN**(r2,k)
4. **IF**(r2,7) 5. **INC**(r1) 6.**GO**(2) **7...**

```
x= 1; ARRAY N[100];
while N[x]-k do { x=x+1; }
```

```
String ARRAY N[100], D[100];
String PROCEDURE GET(String k) {
    int x; boolean Funnet;
    x= 1; Funnet= false;
    while !Funnet do
        { x=x+1;
          if N[x]==k funnet=true;}
    return D[k];
}
```

```
RECORD ELEM {
    String key;
    String data;
}
```

```
Elem array N[100];
get(String k) {...}
```

```
CLASS ELEM {
  private String k,d;
  public String key()
  { return k; }
}
```

```
CLASS ORDBOK {
  private Elem array N[100];
  public String get(String e)
  public String put(Elem k)
}
```

1.a En type

int, boolean, char, String, ...

gir bruker (programmerer) mulighet til å

- *konstruere nye instanser og*
- *behandle disse gjennom et gitt grensesnitt*

INT x=0, y=5; x= x+2; x= x % y; y= y-x; ...

1,2,3... :	→ INT
_+_ :	INT × INT → INT
_-_ :	INT × INT → INT
_%_ :	INT × INT → INT

BOOLEAN a, b; a= x<y; b= b **OR** a; b= !b; ...

true,false :	→ BOOLEAN
_==_ :	INT × INT → BOOLEAN
_<_ :	INT × INT → BOOLEAN
!_ :	BOOLEAN → BOOLEAN

- ***uten å vite noenting om implementasjon !!!***

1.b Brukerdefinerte typer

```
PUBLIC CLASS ORDBOK {
   private Elem[] N;
   PUBLIC OrdBok(int k) { N= new Elem[k]; }
   PUBLIC ELEM GET(STRING e) {...}
   PUBLIC VOID PUT(ELEM k) {...}
}
```

NEW_ :	INT → ORDBOK
.PUT :	ORDBOK × ELEM → ORDBOK
.GET :	ORDBOK × STRING → ELEM

```
OrdBok ob= NEW OrdBok(100);
ob.PUT(new Elem(“a”,”aaaaaaa”));
ob.PUT(new Elem(“b”,”bbbb”));
ob.PUT(new Elem(“c”,”cccccc”));
Elem e= ob.GET(“b”);
```

utvider språket

1.c Datatyper og typing

En datatype er:

- **en sort** T: en mengde av instanser
- **et grensesnitt**:
  et sett med operasjoner som kan utføres på instanser
- **en implementasjon** av T og operasjonene

Typing:

- *krever økt programmeringsdisiplin*
- *tillater å oppdage mange feil under kompilering*
- *øker betraktelig pålitelighet av programvare*
- *fører til muligheter for*
  – gjenbruk og
  – modularisering av programvare

2. Objekt-orientering og typing

```
CLASS ELEM
{ PRIVATE String k,d;
  PUBLIC ELEM(String a, String b)          { k= a; d= b; }
  PUBLIC STRING KEY()      {return k;}
  PUBLIC STRING DATA()     {return d;}
}
```

```
CLASS ORDBOK
{ PRIVATE Elem array N[]; PRIVATE int max;
  PUBLIC ORDBOK(int m)
     { max= m; N= new Elem[max]; }
  PUBLIC VOID PUT(Elem e)
     { max++; N[max]= e; }
  PUBLIC ELEM GET(String k)
     { for (int i=1; i<=max; i++)
          if (N[i].key()==k) return N[i];
       return null;
}}
```

```
CLASS ORDBOKLS EXTENDS ORDBOK
{ PUBLIC VOID LISTSORTERT()
  { for (int i=1; i<=max; i++)
       for (int j=1; j<=max-i; j++)
          if (N[j].key()>N[j+1].key()) swap(j,j+1)
    for (int i=1; i<=max; i++)
       skriv(N[i]);
} }
```

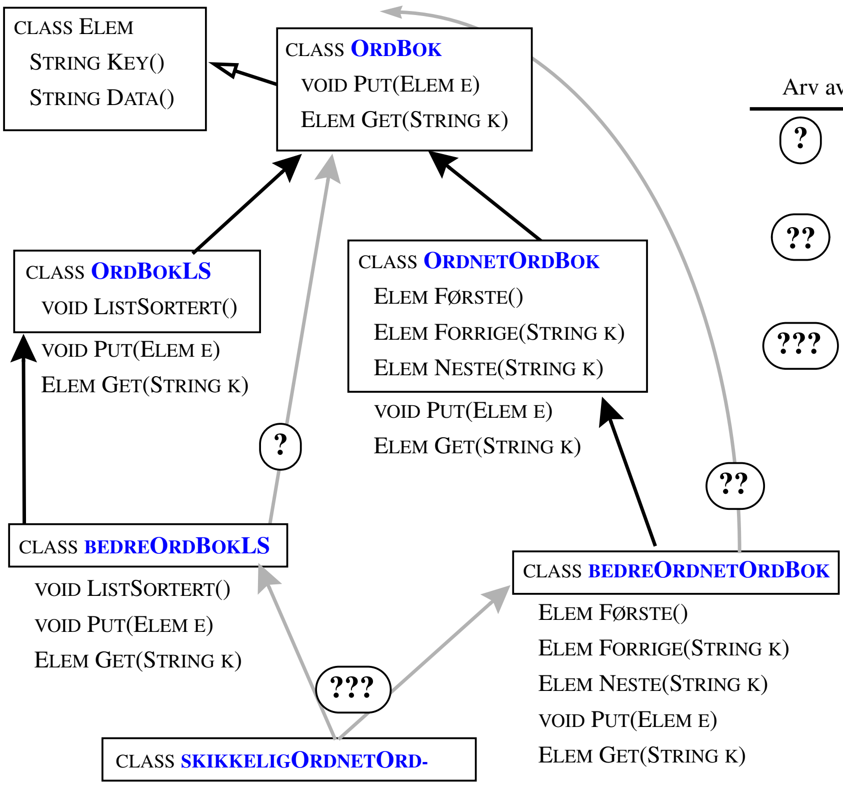
2. Objekt-orientering og typing

```
CLASS ELEM
{ private String k,d;
  PUBLIC ELEM(String a, String b)   { k= a; d= b; }
  PUBLIC STRING KEY()   {return k;}
  PUBLIC STRING DATA()  {return d;}
}
```

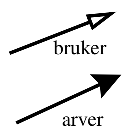
```
CLASS ORDBOK
{ private Elem array N[]; private int max;
  PUBLIC ORDBOK(int m)
    { max= m; N= new Elem[max]; }
  PUBLIC VOID PUT(Elem e)
    { max++; N[max]= e; }
  PUBLIC ELEM GET(String k)
    { for (int i=1; i<=max; i++)
        if (N[i].key()==k) return N[i];
      return null;
}}
```

```
CLASS ORDBOKLS EXTENDS ORDBOK
{ PUBLIC VOID LISTSORTERT()
  { for (int i=1; i<=max; i++)
      for (int j=1; j<=max-i; j++)
        if (N[j].key()>N[j+1].key()) swap(j,j+1)
    for (int i=1; i<=max; i++)
      skriv(N[i]);
} }
```


2.a Statisk klassehierarki



Arv av :	type	implem.
?	ORDBOKLS;	kun fra ORDBOK
??	ORDNETORDBOK;	ikke ORDBOK (datastr.)
???	ORDBOKLS + ORDNETORDBOK	???



OO og arv: oppsummering

1. organiserer ***begreper*** i et statisk hierarki av abstraksjoner og forfininger
2. arver ***både type*** (grensesnitt) og ***implementasjon*** (intern datastruktur, algoritmer...)
3. kan ***gjenbruke implementasjon*** fra superklasser

Dette er bra når det virker men ofte kan føre til en rekke problemer

- forbedring av implementasjon av eksisterende klasser (mer effektiv datastruktur) :
  - kan kreve tilsvarende endringer i alle subklasser
  - to implementasjoner av en klasse er lite nyttige med mindre de begge har en felles superklasse
- forbedring av implementasjon av en subklasse kan kreve:
  - fullstendig overskriving av metoder fra superklasse
  - innføring av en helt ny datastruktur (superklassens datastruktur blir overflødig men dog arvet)
- ofte en ny klasse kan settes inn på forskjellige steder i klassehierarki
  - multipel arv er problematisk pga. arv av implementasjon

Hvor er den skyldige?

Type ∩ implementasjon ≠ Ø : *disse bør holdes adskilt !!!*

3. Utvikling av Typer = utvikling av Språket

Packages
- *moderne språk distribueres sammen med et sett **pakker***
- *disse er **ikke en del** av språket men*
- *er verktøy utviklet i språket for **spesifikke formål***

java.util.Dictionary

public Object get(Object key)
@Param: ***key*** - a key in this dictionary.
@Returns: the value to which the ***key*** is mapped in this dictionary; ***null*** if none

public Object put(Object key, Object value)
Maps the specified ***key*** to the specified ***value*** in this dictionary. Neither the ***key*** nor the ***value*** can be ***null***. The ***value*** can be retrieved by calling the ***get*** method with a ***key*** that is equal to the original ***key***.
@Param: ***key*** - the hashtable key;
value - the value;
@Returns: the previous ***value*** to which the ***key*** was mapped in this dictionary, or ***null*** if the ***key*** did not have a previous mapping.
@exception: ***NullPointerException*** – if the ***key*** or ***value*** is ***null***.

java.util.Date

public Date( )
Allocates a Date object and initializes it so that it represents the time at which it was allocated measured to the nearest millisecond.

public boolean after(Date when) Tests if this ***Date*** is after the specified ***Date***
@Param: ***when*** - a ***Date***.
@Returns: ***true*** iff this ***Date*** is after the argument ***Date***; ***false*** otherwise

public int setMinutes(int minutes)
@Param: ***minutes*** - the value of the minutes to be set for this ***Date***

public void getMinutes( )
@Returns: the number of minutes past the hour represented by this ***Date*** ; The value returned is between 0 and 59

3.a En pakke

- *er en type (eller samling av typer) som utvider språket til et* ***domenespesifikk språk***

- ***dokumentasjon*** *av metoder er*
  – alt som er tilgjengelig utenfra og, faktisk
  – alt som bruker trenger for å benytte seg av pakken

- *bruker vet ikke (og vil ikke vite) hvordan tingene er implementert inne i pakken*

dvs.

Alt som trenges

for å benytte seg av en Type (pakke, innebygget datatype...)

er kjennskap til

spesifikasjon av grensesnittmetoder

3.b Package

- *En ADT (interface) samler noen relaterte funksjoner*
- *Dens implementasjon bruker en valgt Data Struktur*
- *Flere ADT'er og DT'er vil ofte utgjøre en helhet – en pakke – for håndtering av spesielt sett av problemer.*

1. Opprett i din hjemmekatalog en underkatalog og kall den, f.eks. 'Pakker' – alle dine pakker skal ligge i denne katalogen

2. I Unix, kjør kommando
> setenv CLASSPATH . : /local/pub/kurs/i120 : $HOME/Pakker : /usr/java/lib
(dette kan legges til din .cshrc filen)

3. Når du lager en ny pakke, si, niceIO
- **opprett katalog 'niceIO' i katalogen 'Pakker'**
- **alt som hører til 'niceIO'-pakken skal legges i katalogen ' $HOME/Pakker/niceIO/ '**
- **enhver fil 'klass.java' i denne katalogen skal starte med**
  package niceIO ;

4. For å bruke en pakke, niceIO, i en klass, skriv
import niceIO.* ;
på toppen – før public class klass { ... i filen 'klass.java'

For å bruke bare en spesifikk klasse, terminalIO, fra denne pakken:
import niceIO.terminalIO ;

5. F.eks. java.awt er en pakke for grafisk brukergrensesnitt;
- **alle dens klasser ligger i katalogen ' /usr/java/src/java/awt '**
- **du bruker den/importerer ved å starte din 'klass.java' med**
  import java.awt.* ;

4. Abstrakte DataTyper

1.	Ø	=	***grensesnitt*** *Hva*	$\cap$	***implementasjon*** *Hvordan*
2.	***datatype***	=	***grensesnitt***	+	***implementasjon***
	grensesnitt	=	***datatype***	–	***implementasjon***
3.	***Abstrakt DataType***	=	***datatype***	–	***implementasjon***
	ADT	=	***grensesnitt:*** *Hva*		
	ADT modul	=	*implementasjon av en ADT som avhenger kun av andre ADT* *(meningsfull kun i en omgivelse med flere ADT)*		

4.a Pseudokode

```
public int[] SS( int[] tab)
{   int m, i;
    for ( int k = 0; k < tab.length;k++)
    {  m = tab[k]; i = k;
       for ( int j = k+1; j < tab.length; j++)
            if (tab[j] < m) { m = tab[j]; i = j; }
       tab[i] = tab[k];
       tab[k] = m;
    }
    return tab;
}
```

4.a Pseudokode

```
public int[] SS( int[] tab)
{     int m, i;
      for ( int k = 0; k < tab.length;k++)
      {   m = tab[k]; i = k;
          for ( int j = k+1; j < tab.length; j++)
               if (tab[j] < m) { m = tab[j]; i = j; }
          tab[i] = tab[k];
          tab[k] = m;
      }
      return tab;
}
```

```
/** SS - sorterer input array:
 *      @param  - int tab[0...n]
 *      @return - sortert tab
 *   for ( k = 0,1,2...n-1) {
 *      i = indeksen til minste elementet i
 *          tabellsegmentet tab[k...n]
 *      bytt elementene ved indeks k og i
 *      }
 */
```

Dette er nok for å se at det virker:

k= 0	2	4	1	3	5	i= 2
k= 1	1	4	2	3	5	i= 2
k= 2	1	2	4	3	5	i= 3
k= 3	1	2	3	4	5	i= 3
k= 4	1	2	3	4	5	i= k

Løkkeinvariant:
Etter k-te iterasjon er elementene 0...k riktig plassert

4.b ADT = programmering med egenskaper (grensesnitt)

```
/** SS - sorterer input array:
 *      @param  - int tab[0...n]
 *      @return - sortert tab
 *   for ( k = 0,1,2...n-1) {
 *      i = indeksen til minste elementet i
 *          tabellsegmentet tab[k...n]
 *      bytt elementene ved indeks k og i
 *      }
 */
```

```
public void SS(int[] N) {        // int >
    int m, i;
    for (int k=0; k < N.length; k++) {
        m = N[k]; i = k;
        for (int j=k+1; j < N.length; j++)
            if ( N[ j ] > m ) { m = N[j]; i= j; }
        N[i]= N[k]; N[k]= m;
} }
```

```
public void SS(int[] N) {        // int <
    int m, i;
    for (int k=0; k < N.length; k++) {
        m = N[k]; i = k;
        for (int j=k+1; j < N.length; j++)
            if ( N[ j ] < m ) { m = N[j]; i= j; }
        N[i]= N[k]; N[k]= m;
} }
```

```
public void SS(Date[] N) {        // Date <
  int m, i;
  for (int k=0; k < N.length; k++) {
     m = N[k]; i = k;
     for (int j=k+1; j < N.length; j++)
       if (N[ j ] . before(m)) { m = N[j]; i= j; }
     N[i]= N[k]; N[k]= m;
} }
```

4.c Comparator ADT

```
public interface Comparator
{       /** @return true iff a < b; false otherwise */
boolean isLessThan (Object a, Object b)
        /** @return true iff a > b; false otherwise */
boolean isGreaterThan (Object a, Object b)
        /** @return true iff a = b; false otherwise */
boolean isEqualTo (Object a, Object b)
        /** @return true iff a <= b; false otherwise */
boolean isLessThanOrEqualTo (Object a, Object b)
        /** @return true iff a >= b; false otherwise */
boolean isGreaterThanOrEqualTo (Object a, Object b)
}
```

```
public void SS(Date[] N) {        // Date <
      j; }
```

```
public void SS(int[] N) {         // int >
      j; }
```

```
public void SS(int[] N) {         // int <
    int m, i;
    for (int k=0; k < N.length; k++) {
      m = N[k]; i = k;
      for (int j=k+1; j < N.length; j++)
         if ( N[ j ] < m ) { m = N[j]; i= j; }
      N[i]= N[k]; N[k]= m;
}  }
```

```
public void GSS(Object[] N, Comparator CP) {
    int m, i;
    for (int k=0; k < N.length; k++) {
       m = N[k]; ind = k;
       for (int j=k+1; j < N.length; j++)
          if ( CP. isLessThan (N[ j ], m) ) { m = N[j]; i= j; }
       N[i]= N[k]; N[k]= m;
}  }
```

*bruker **ADT Comparator** – dvs. vilkårlig implementasjon !!!*


```
Integer[] N;
Date[] D;
```

```
class IntReversComp implements Comparator
  public boolean isLessThan (Object a, Object b)
     { return ((Integer)a).intValue() > ((Integer)b).intValue(); }
  public boolean isGreaterThan (Object a, Object b)
     ...() < ((Integer)b).intValue(); }
  ...ect a, Object b)
  ...qualTo (Object a, Object b)
  ...rEqualTo (Object a, Object b)
```

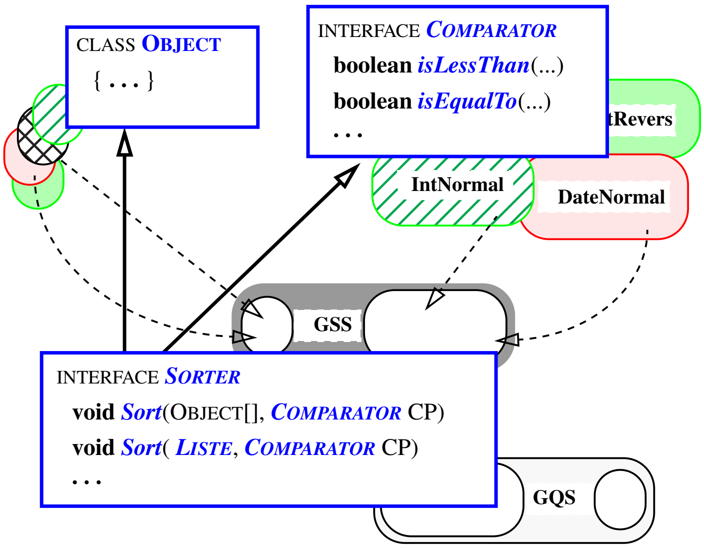
```
class IntNormalComp implements Comparator
  public boolean isLessThan (Object a, Object b)
    { return ((Integer)a).intValue() < ((Integer)b).intValue(); }
  public boolean isGreaterThan (Object a, Object b)
    { return ((Integer)a).intValue() > ((Integer)b).intValue(); }
  public boolean isEqualTo (Object a, Object b)
    { return ((Integer)a).intValue() == ((Integer)b).intValue(); }
  public boolean isLessThanOrEqualTo (Object a, Object b)
    { return ((Integer)a).intValue() <= ((Integer)b...
  public boolean isGreaterThanOrEqualTo (Ob...
    { return ((Integer)a).intValue() >= ((Integer)b...
```

```
class DateNormalComp implements Comparator
  public boolean isLessThan (Object a, Object b)
    { Date ad= (Date)a; Date bd= (Date)b;
      if (ad.getYear() < bd.getYear()) return true;
      else if (ad.getYear() > bd.getYear()) return false;
      else if (ad.getDay() < bd.getDay()) return true;
      else if (ad.getDay() > bd.getDay()) return false;
      else if (ad.getHour() < bd.getHour()) return true;
      else if ....    }
  public boolean isGreaterThan (Object a, Object b)   { ... }
```

```
...
GSS(N, new IntNormalComp());
GSS(N, new IntReversComp());
GSS(D, new DateNormalComp());
...
```

4.d Modularisering av implementasjon

1. GSS *benytter kun abstrakte egenskaper – grensesnitt metoder – av* ***Comparator*** *(og* ***Object[]****)*
2. *den er uavhengig av hvordan disse er implementert*
3. *og kan akseptere enhver ny (endret/forbedret) implementasjon så lenge det er en implementeasjon av respektivt grensesnitt*




```
public void GSS(Object[] N, Comparator CP) {
   int m, i;
   ...
        if ( CP. isLessThan (N[ j ], m) )
   ...
```

1. *overalt i mine programmer der jeg har behov for sammenlikning skal jeg bruke* ***Comparator ADT***
2. *alle mine* **(A)DTer** *vil benytte kun* ***Comparator*** *(i grensesnitt) og ikke noen implementasjon av denne*
3. *og alle mine* **implementasjoner** *skal foreta sammenlikninger kun gjennom* ***Comparator*** *grensesnittet*

The "Millennium Bug"

```
MinSS store= new MinSS();
store.ins(new prod(...)) ;
....
  public void remOverdue() {
      boolean done= false;
      aa = getYear();
      mm = getMonth();
      dd = getDay();
      prod p= (prod)store.getMin();
      while (! done) {
          if ( p.a < aa || (p.a==aa && p.m < m)
              || (p.a==aa && p.m==m && p.d < dd) )
              { store.remMin();
              p= (prod)store.getMin();
          } else done = true ;
```

```
class prod {
  public int d, m, a;
  public boolean lt(prod p) {
  if ( a < p.a || (a==p.a && m < p.m)
      || (a==p.a && m==p.m && d < p.d) )
      return true;
  else return false; }
```

n && d < p.d))

60% av kostnader går ikke til programutvikling men til vedlikehold !!!

ADT programmering

```
import IDate,IMin,Comparator;
MinSS store;
Comparator dcp = new DatoCp();
store = new MinSS(dcp);
store.ins(new prod(...)) ;
....
 public void remOverdue() {
   boolean done = false;
   IDate today = getDate();
   prod p= (prod)store.getMin();
   while (! done) {
     if (dcp.isLessThan(p.dt,today)) {
       store.remMin();
       p= (prod)store.getMin();
     }
     else done = true;
   }
 ...
```

```
class prod {
   public IDate dt;
... }
```

```
public interface IDate {
   int d();
   int m();
   int a();
   setA(int aa);
... }
```

```
public class DatoCp implements Comparator {
//sammlikner IDate-objekter
public boolean isLessThan(Object d1,d2) {
    Dato e= (Dato)d1; Dato f= (Dato)d2;
    if (e.a()<100) e.setA(1900+e.a());
    if (f.a()<100) f.setA(1900+f.a());
    if (e.a<f.a || (e.a==f.a && e.m<f.m)
        ||(e.a==f.a && e.m==f.m && e.d<f.d) )
        return true;
    else return false;
    }
public boolean isEqualTo(Object d1,d2) {
    ...
```


Oppsummering: Programutvikling

Bruk av ADTer gir modulære programmer med økt gjenbrukbarhet og forenklet vedlikehold

Programmerer (bruker) vil ikke vite hvordan moduler/pakker han benytter er implementert
– man er interessert kun i hvilken funksjonalitet de tilbyr, dvs. kun i grensesnittet

ADT = grensesnittmetoder med tilstrekkelig dokumentasjon (interface + Javadoc)

DT = implementasjon av en ADT med et valg av datastruktur og passende algoritmer

---

1. Finn ut hvilke moduler du trenger :
   - hvordan de skal samarbeide og
   - gjennom hvilke grensesnitt
     – noen vil være tilgjengelig fra eksisterende bibliotek/pakker
     – andre vil du måtte lage selv

For hver enkel modul

2. velg en data struktur og
3. design nødvendige algoritmer

(Her skal du tenke effektivitet)

4. Implementer alle moduler og sett de sammen iflg. 1.

Metodologi og teknikker

1 *ADT programmering vs. i-110 interface, OO*
2 mer ADT og OO
3 algoritmeanalyse, implementasjon
4 rekursjon

Deretter mer konkret

– algoritmer og datastrukturer