Invarianter

IN1010 – Objektorientert programmering

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
in1010@ifi.uio.no

Introduksjon

Plan for timen

- Vi skal lære litt om *tilstander* og *tilstandsoverganger*
- Det lar oss snakke om *egenskaper* ved tilstander og invarianter
- Vi skal lære om *før- og etterbetingelser*
- Vi skal lære om to former for *invarianter*:
  - Klasseinvarianter
  - Løkkeinvarianter
- Vi ser et litt større eksempel på bruk av klasseinvarianter
- Til slutt se litt på dokumentasjon

Tilstander

Tilstander

- En tilstand gir et *øyeblikksbilde* av et kjørende program
- En tilstand består av:
  - verdiene til alle variabler i hele programmet, samt
  - hvor i programutførelsen man er
- Hvor i programutførelsen man er gis av en *programteller*
  - Vi vil tenke på programtelleren som neste linjenummer som skal eksekveres
  - Dette er en forenkling, siden den vil egentlig være mer finkornet
  - Vi kaller den `PC`, kort for Program Counter

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Ved første linje har vi kun en programteller

PC
1

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Ved linje fire har vi initialisert tre variabler

PC	n	i	f
4	5	1	1

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

I løkken endrer vi variablene `f` og `i`

PC	n	i	f
6	5	1	1
7	5	1	1
5	5	2	1

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

I løkken endrer vi variablene `f` og `i`

PC	n	i	f
6	5	2	1
7	5	2	2
5	5	3	2

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

I løkken endrer vi variablene `f` og `i`

PC	n	i	f
6	5	3	2
7	5	3	6
5	5	4	6

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

I løkken endrer vi variablene `f` og `i`

PC	n	i	f
6	5	4	6
7	5	4	24
5	5	5	24

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

I løkken endrer vi variablene `f` og `i`

PC	n	i	f
6	5	5	24
7	5	5	120
5	5	6	120

Tilstander (eksempel)

Innholdet i `main`:

```
1  int n = 5;
2  int i = 1;
3  int f = 1;
4
5  while (i <= n) {
6      f *= i;
7      i++;
8  }
9
10 System.out.println(f);
```

Se eksempelet i Python tutor.

Ved siste linje skrives verdien av `f` ut

PC	n	i	f
10	5	6	120

Litt mer om tilstander

- En kjøring av et program kan ses på som en sekvens av tilstander
- Hver programsetning tar programmet fra en tilstand til en annen
  - Disse kalles tilstandsoverganger
- Programtelleren endres implisitt
  - Semikolon kan ses på som en inkrementering av programtelleren
  - Metodekall, `if`, `while`, og lignende har egne regler for programtelleren
- Variabler endres eksplisitt med tilordninger (=)
- En *debugger* er et verktøy som lar deg inspisere tilstanden til et program
- Tilstanden til *hele* programmet blir veldig fort vanskelig å tenke på
  - Vi ønsker å kunne resonnere over deler av tilstanden

Tilstand i objekter

- Tilstanden i et *objekt* er gitt av verdiene til alle instansvariablene
- Objektdiagrammer er en måte å illustrere tilstanden i et objekt

```
navn: "Alice"
alder: 25
tlf: "+4711235813"
e-post: "alice@wonderland.com"
```

- Objekter gir en meningsfull måte å dele opp tilstanden på

Invarianter

Invarianter

- En invariant er en påstand om programmets tilstand som alltid er sann
  - En påstand eller betingelse er noe som kan sies å være sant eller usant
- Eksempler:
  - `antall` er et ikke-negativt heltall
  - I et person-objekt med peker til en forelder er `this.alder < this.forelder.alder`
  - Etter `i` iterasjoner inneholder `f` det `i`-te fakultetstallet
  - Etter `i` iterasjoner er de `i` siste tallene i arrayet sortert
  - Hastigheten til en partikkel er mindre enn lysets hastighet
- Ofte vil de uttrykke ting som virker helt åpenbare
  - For eksempel uttrykker typen til en variabel en invariant
- De er nyttige fordi de gir noen holdepunkter når vi resonnerer om programmer
- Brudd på invarianter er et tydelig symptom på at noe er galt i programmet

assert

- `assert` tar et boolsk uttrykk, og kaster et unntak hvis det er `false`
  - Se Java sin dokumentasjon
- En `assert` kommer i to former:
  - `assert booleanExpression;`

    ```
    Exception in thread "main" java.lang.AssertionError
        at Main.main(Main.java:4)
    ```
  - `assert booleanExpression : "uffda";`

    ```
    Exception in thread "main" java.lang.AssertionError: uffda
        at Main.main(Main.java:4)
    ```
- Assert må aktiveres med et kommandolinjeflagg:

```
$ java -ea Main
```

Når `assert` bør brukes

- Du kan legge inn `assert` for å sørge for at invariantene ikke brytes
- Det skal *ikke* brukes for å behandle feil
  - Det er en måte å oppdage feil *under utvikling og testing*
- Ofte tar det tid å sjekke om invarianten holder

```
1 assert myList.isSorted();
```

  - Derfor er de avskrudd ved vanlig kjøring (i produksjon)
- Tommelfingerregel:
  - De bør *ikke* brukes til å sjekke input til `public` metoder
  - De bør brukes til å sjekke input til `private` metoder

Før- og etterbetingelser

- Metoder kan ha før- og etterbetingelser (eng: pre- and post conditions)
- Førbetingelser må være oppfylt før et metodekall
  - `collection != null && !collection.isEmpty()`
  - `file.exists()`
  - `0 <= alder && alder <= 120`
- Etterbetingelser må være oppfylt når metoden returnerer
  - `result != null`
  - `collection.isEmpty()`
  - `0 < alder`
- Å programmere med før- og etterbetingelser kalles «Design by contract»

Klasseinvarianter

- En klasseinvariant er en invariant som holder for *alle* instanser av klassen
  - Konstruktøren må gjøre invarianten sann
  - Hver metode i klassen har klasseinvarianten som både før- og etterbetingelse
- Ofte er det tilstrekkelig å legge dem inn som en kommentar i klassen
  - De kan eksplisitt sjekkes ved å legge til en `assert` i slutten av hver metode
- Hold instansvariabler `private`
  - Da er det tilstrekkelig å sørge for at alle metoder opprettholder invarianten
  - Hvis en instansvariabel er `public` har du ingen garantier utover typen
  - (Det samme gjelder ubegrensede `public` settere)
- Sjekk input til `public` metoder
  - Vi har ingen garantier for hvordan `public` metoder blir kalt
  - Sjekk inputverdier og kast fornuftige unntak
- Bruk `assert` for `private` metoder
  - Private metoder kan kun kalles fra andre metoder i samme klasse
  - Innad i klassen kan du anta at metoden kalles med fornuftige argumenter

Løkkeinvarianter

- En løkkeinvariant er en invariant som holder for alle iterasjoner av løkken
- Oftest vil løkkeinvarinten si noe om *målet* med løkken
  - `sum` inneholder summen av de `i` første tallene i `arr`
  - De `i` første elementene i `arr` er sortert
- Du kan bruke løkkeinvarianten til å vise at en algoritme er korrekt

Løkkeinvarianter (eksempel)

```java
public static int min(int arr[]) {
    if (arr.length == 0) {
        throw new IllegalArgumentException("Arrayet må ikke være
        ↪  tomt");
    }
    int min = arr[0];
    // min er det minste tallet mellom arr[0] og arr[0]
    for (int i = 1; i < arr.length; i++) {
        // min er det minste mellom arr[0] og arr[i-1]
        if (arr[i] < min) { min = arr[i]; }
        // min er det minste mellom arr[0] og arr[i]
    }
    // min er det minste mellom arr[0] og arr[arr.length - 1]
    return min;
}
```

Et større eksempel med klasseinvarianter

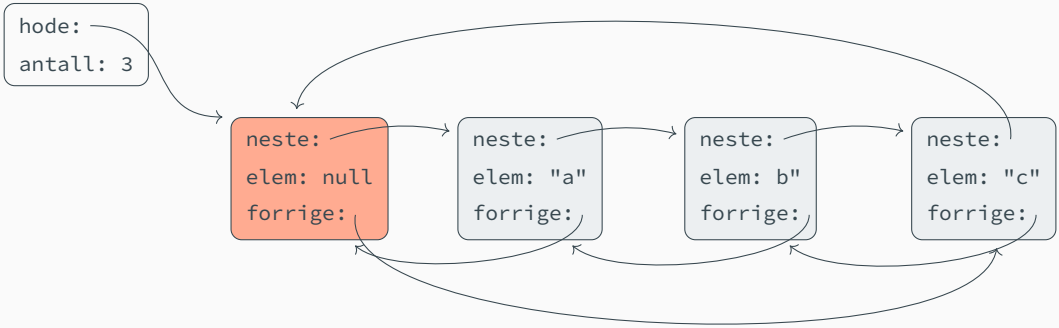
SortertMengde

- Vi skal lage en enkel datastruktur som implementerer en mengde som en ordnet syklisk dobbeltlenket liste
- Den inneholder klasseinvarianter som:
  - Elementene er ordnet (fra minst til størst)
  - Det er ingen duplikater
  - En node sin neste sin forrige er noden selv (og andre veien)
  - Den lagrede størrelsen svarer til antall noder i strukturen
- Koden ligger her:
  `https://gist.github.uio.no/larstvei/1c022166f50d10b37ebc192c0acd25f3`

SortertMengde (struktur)

```
1  public class SortertMengde<T extends Comparable<T>> {
2      private final Node hode;
3      private int antall = 0;
4
5      private class Node {
6          Node forrige, neste;
7          T elem;
8          void settInn(T elem) { /* ··· */ } // Sett inn etter
9          void fjern() { /* ··· */ }         // Fjern denne
10     }
11
12     public SortertMengde() { /* ··· */ }
13     public boolean erTom() { /* ··· */ }
14     public int antall() { /* ··· */ }
15     public SortertMengde<T> settInn(T elem) { /* ··· */ }
16     public SortertMengde<T> fjern(T elem) { /* ··· */ }
17 }
```

SortertMengde (tilstand)



SortertMengde(sjekkInvarianter)

```
1 private void sjekkInvarianter() {
2     assert erSortert() : "Den sorterte mengden er ikke
      ↪  sortert!";
3     assert erRiktigLenket() : "Nodene er ikke riktig lenket!";
4     assert ingenDuplikater() : "Mengden inneholder duplikater!";
5     assert sjekkStørrelse() : "Antall noder samsvarer ikke med
      ↪  antall()!";
6 }
```

SortertMengde (settInn)

```
1  public SortertMengde<T> settInn(T elem) {
2      Node it = hode.neste;
3      while (it != hode && it.elem.compareTo(elem) < 0) {
4          it = it.neste;
5      }
6      if (!elem.equals(it.elem)) {
7          antall++;
8          it.forrige.settInn(elem);
9      }
10     sjekkInvarianter();
11     return this;
12 }
```

SortertMengde (fjern)

```java
public SortertMengde<T> fjern(T elem) {
    Node it = hode.neste;
    while (it != hode && !elem.equals(it.elem)) {
        it = it.neste;
    }
    if (elem.equals(it.elem)) {
        antall--;
        it.fjern();
    }
    sjekkInvarianter();
    return this;
}
```

SortertMengde (erSortert)

```
1  private boolean erSortert() {
2      Node it = hode.neste;
3      while (it.neste != hode) {
4          if (0 <= it.elem.compareTo(it.neste.elem)) {
5              return false;
6          }
7          it = it.neste;
8      }
9      return true;
10 }
```

SortertMengde (ingenDuplikater)

```
1  private boolean ingenDuplikater() {
2      Node it = hode.neste;
3      while (it.neste != hode) {
4          if (it.elem.equals(it.neste.elem)) {
5              return false;
6          }
7          it = it.neste;
8      }
9      return true;
10 }
```

SortertMengde (erRiktigLenket)

```
1  private boolean erRiktigLenket() {
2      Node it = hode.neste;
3      while (it.neste != hode) {
4          if (it.neste.forrige != it || it.forrige.neste != it) {
5              return false;
6          }
7          it = it.neste;
8      }
9      return true;
10 }
```

SortertMengde (sjekkStørrelse)

```
1  private boolean sjekkStørrelse() {
2      Node it = hode.neste;
3      int beregnetStørrelse = 0;
4      while (it != hode) {
5          it = it.neste;
6          beregnetStørrelse++;
7      }
8      return beregnetStørrelse == antall;
9  }
```

SortertMengde (Testprogram)

```java
public static void main(String[] args) throws IOException {
    String filsti = "wordle-ord-tilfeldig-med-duplikater.txt";
    List<String> ord = Files.readAllLines(Paths.get(filsti));
    System.out.println("Antall ord i filen: " + ord.size());

    SortertMengde<String> set = new SortertMengde<>();

    for (String s : ord) { set.settInn(s); }
    System.out.println("Etter innsetting: " + set.antall());

    // Ingen duplikater snikes inn?
    for (String s : ord) { set.settInn(s); }
    System.out.println("Etter ny innsetting: " + set.antall());

    for (String s : ord) { set.fjern(s); }
    System.out.println("Etter sletting: " + set.antall());
}
```

SortertMengde (Kjør test)

```
$ time java SortertMengde
Antall ord i filen: 29710
Etter innsetting: 14855
Etter ny innsetting: 14855
Etter sletting: 0


_____________________________________________________
Executed in    3.18 secs    fish           external
   usr time    3.20 secs    0.17 millis    3.20 secs
   sys time    0.05 secs    1.33 millis    0.05 secs
```

SortertMengde (Kjør test med -ea)

```
$ time java -ea SortertMengde
Antall ord i filen: 29710
Etter innsetting: 14855
Etter ny innsetting: 14855
Etter sletting: 0


------------------------------------------------------
Executed in   29.06 secs    fish           external
   usr time   29.07 secs    0.23 millis   29.07 secs
   sys time    0.08 secs    1.82 millis    0.07 secs
```

Dokumentasjon

Kode, kommentarer og dokumentasjon

> *Programs must be written for people to read, and only incidentally for machines to execute.*
>
> — *Harold Abelson*

- Kode inneholder sannheten om nøyaktig hva programmet gjør
- Kommentarer er et supplement til koden
  - De bør *ikke* si det samme som koden allerede sier
  - Som regel bør de si noe om *hvorfor* koden er skrevet som den er
  - Kommentarer er beskjeder til kolleger eller deg selv i fremtiden
- Dokumentasjon rettes mot de som *bruker*, men ikke utvikler, koden
  - Bruk i denne konteksten betyr å bruke koden som et bibliotek
  - Vi skal se litt konkret på hvordan dokumentasjon skrives i Java

Selvforklarende kode

- Ideelt sett skal koden du skriver være *selvforklarende*
  - Hva som er selvforklarende er avhengig av leseren
  - Helt selvforklarende kode er sjeldent mulig å oppnå
- Når koden ikke er selvforklarende, bør den suppleres med kommentarer
  - Før du skriver en kommentar bør du først forsøke å forenkle koden
- Noen tips for å gjøre koden selvforklarende:
  - Finn gode navn (dette er veldig mye lettere sagt enn gjort)
  - Følg språkets *idiomer* (følge «god stil» i det aktuelle språket)

javadoc

- Java kommer med en kommando `javadoc` som genererer en nettside fra kildekode
- Verktøyet rapporterer feil hvis:
  - `public` metoder står udokumentert
  - Parametere til `public` metoder står udokumentert
  - Returverdien til `public` metoder står udokumentert
  - Unntak som kastes fra `public` metoder står udokumentert
- Vi tar en titt på kildekoden til Java sin `LinkedList` implementasjon
  - `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/LinkedList.java`
  - og den tilhørende dokumentasjonen
  - `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/LinkedList.html`
- Eksempelet fra denne forelesningen er også dokumentert:
  - `https://gist.github.uio.no/larstvei/1c022166f50d10b37ebc192c0acd25f3`
  - Du kan generere dokumentasjon for det selv med
    ```
    $ javadoc SortertMengde.java
    ```