Föreläsning 6

Eleganta metoder
Separation of concern
Command-Query Separation Principle
Kontraktbaserad design
Assertions
Självdokumenterande kod

Använd beskrivande namn

Använd metodnamn som indikerar syftet med metoden, dvs vad man vill åstadkomma med metoden, inte hur metoden är implementerad.

Typiska namn på metoder är verb eller verbfraser

sort print move add

Metoder som returnerar ett värde skall ha namn som återspeglar värdet som returneras

getSize isVisible getPosition isEmpty

Namnen på metoder skall, i likhet med namnen på alla entiteter i ett program (variabler , klasser, interface), väljas med stor omsorg.

Använd beskrivande namn

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
        if (l[z] == 10) {
            q += 10 + (l[z + 1] + l[z + 2]);
            z += 1;
        } else if (l[z] + l[z+ 1] == 10) {
            q += 10 + l[z + 2];
            z += 2;
        } else {
            q += l[z] + + l[z + 1];
            z += 2;
        }
    }
    return q;
}
```

Förstår du vad metoden gör?

Använd beskrivande namn

```
public int evaluateScore() {
    int score = 0;
    int frame = 0;
    for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
        if (rolls[frame] == 10) {
            score += 10 + rolls[frame +1] + rolls[frame + 2];
            frame += 1;
        } else if ((rolls[frame] + rolls[frame + 1]) == 10) {
            score += 10 + rolls[frame + 2];
            frame += 2;
        } else {
            score += rolls[frame] + rolls[frame + 1];
            frame += 2;
        }
    }
    return score;
}
```

Är det nu lättare att förstå vad metoden gör?

Refactoring:

Använd beskrivande namn

```
public int evaluateScore() {
   int score = 0;
   int frame = 0;
   for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
      if (rolls[frame] == 10) {
         score += 10 + rolls[frame +1] + rolls[frame + 2];
         frame += 1;
      } else if ((rolls[frame] + rolls[frame + 1]) == 10) {
         score += 10 + rolls[frame + 2];
         frame += 2;
      } else {
         score += rolls[frame] + rolls[frame + 1];
         frame += 2;
      }
   }
   return score;
}
```

Refactoring: Döp om identifierare

Är det nu lättare att förstå vad metoden gör?

Funktionell dekomposition

Funktionell dekomposition innebär att man bryter ner koden i ändamålsenliga metoder.

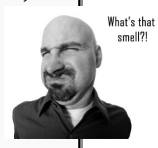
Fördelar:

- ***underlättar läsbarheten***. Ett beskrivande namn på en metod är lättare att läsa än 10 rader kod.
- ***ökar abstraktionsnivån***. Programmerarna kan betrakta metoder som om de vore inbyggda högnivå konstruktioner i programspråket, och kan på så vis sättas sig in i mycket större mängder kod.
- ***reducerar duplicering av kod***. Om ett kodavsnitt upprepas på flera ställen, skall denna kod brytas ut till en metod och den duplicerade koden ersättas med ett metodanrop till metoden. Tillåter att implementationen av en metod ändras utan att koden där metoden anropas påverkas.

Funktionell dekomposition

Exempel: Duplicerad kod

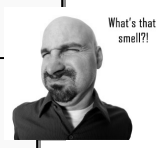
```
public void parse(String expression) {
    . . .
    if (!nextToken.equals("+")) {
        //error
        System.out.println("Expeceted +, but found " + nextToken);
        System.exit(0);
    }
    . . .
    if (!nextToken.equals("*")) {
        //error
        System.out.println("Expeceted *, but found " + nextToken);
        System.exit(0);
    }
    . . .
}//parse
```



Funktionell dekomposition

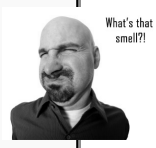
```
public void parse(String expression) {
    . . .
    if (!nextToken.equals("+")) {
        handleError("Expeceted +, but found " + nextToken);
    }
    . . .
    if (!nextToken.equals("*")) {
        handleError("Expeceted *, but found " + nextToken);
    }
    . . .
}//parse
```

```
private void handleError(String message) {
    System.out.println(message);
    System.exit(0);
}//handleError
```



Funktionell dekomposition

```
public void parse(String expression) throws ParseException {
    . . .
    if (!nextToken.equals("+")) {
        handleError("Expeceted +, but found " + nextToken);
    }
    . . .
    if (!nextToken.equals("*")) {
        handleError("Expeceted *, but found " + nextToken);
    }
    . . .
}//parse
```




```
private void handleError(String message) throws ParseException {
    throw new ParseException(message);
}//handleError
```

Funktionell dekomposition

```
public void parse(String expression) throws ParseException {
    . . .
    checkLegalToken(nextToken, "+");
    . . .
    checkLegalToken(nextToken, "*")
    . . .
}//parse
```

Refactoring: Extrahera metoder

```
private void checkLegalToken(String token, String legal) throws ParseException {
    if (!token.equals(legal))
        handleError("Expeceted " + legal + ", but found " + token);
}//checkLegalToken
```

```
private void handleError(String message) throws ParseException {
    throw new ParseException(message);
}//handleError
```

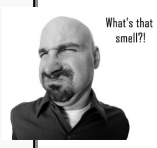
The Separation of Concerns Principle

> ***The Separation of Concerns Principle:***
> *En metod skall endast göra en sak och göra denna sak bra.*

Den uppgift som en metod har skall kunna beskrivas i en mening utan att använda orden *och* eller *samt*. Om detta inte går skall metoden troligen delas upp i två eller flera metoder.

Exempel: En metod som gör flera saker

```
public void doThisOrThat(boolean flag) {
   . . .
   if (flag) {
      //twenty lines of cod to do this
   }
   else {
      //twenty lines of cod to do that
   }
}//doThisOrThat
```



The Separation of Concerns Principle

Genom att införa metoderna `doThis` och `doThat` gör metoden `doThisOrThat` endast en sak, nämligen att fördela arbetet till en av dessa metoder.

```
public void doThisOrThat(boolean flag) {
   . . .
   if (flag) {
      doThis();
   }
   else {
      doThat();
   }
}//doThisOrThat
```

```
private void doThis() {
   //code to do this
}//doThis

private void doThat() {
   //code to do that
}//doThis
```

Refactoring: Extrahera metoder

The Separation of Concerns Principle

```
public void pay() {
   for (Employee e : employees) {
      if (e.isPayday()) {
         Money pay = e.calcuatePay();
          e.deliverPay(pay);
      }
   }
}
```

Metoden `pay` löper igenom alla anställda och för varje anställd

1. kontrollerar att utbetalning skall ske
2. beräknar hur mycket lönen är
3. gör utbetalningen

Metoden gör alltså 3 saker!

The Separation of Concerns Principle

En omskrivning av metoden `pay`, där vi har 3 metoder som var och en gör endast en sak, får följande utseende:

```
public void pay() {
   for (Employee e : employees)
      payIfNecessary(e);
}
private void payIfNecessary(Employee e) {
   if (e.isPayday())
      calculateAndDeliverPay(e);
}
private void calculateAndDeliverPay(Employee e) {
   Money pay = e.calcuatePay();
   e.deliverPay(pay);
}
```

Refactoring: Extrahera metoder

Följer man *Separation of Concern* fås små och överblickbara metoder som är lätta att läsa och förstå!

Bowling-exemplet: Slutlig design

```
public int evaluateScore() {
    int score = 0;
    int frame = 0;
    for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
        if (isStrike(frame)) {
            score += 10 + nextTwoBallsForStrike(frame);
            frame += 1;
        } else if (isSpare(frame)) {
            score += 10 + nextBallForSpare(frame);
            frame += 2;
        } else {
            score += twoBallsInFrame(frame);
            frame += 2;
        }
    }
    return score;
}
```

Refactoring: Extrahera metoder

```
private boolean isStrike(int frame) {
    return rolls[frame] == 10;
}
private boolean isSpare(int frame) {
    return (rolls[frame] + rolls[frame + 1]) == 10;
}
private int nextTwoBallsForStrike(int frame) {
    return rolls[frame +1] + rolls[frame + 2];
}
private int nextBallForSpare(int frame) {
    return rolls[frame + 2];
}
private int twoBallsInFrame(int frame) {
    return rolls[frame] + rolls[frame + 1];
}
```

Sidoeffekter

En sidoeffekt av en metod är varje modifikation av data som är *observerbar* när metoden anropas.

Om en metod inte har någon sidoeffekt, kan man anropa metoden så ofta man vill och alltid få samma svar (såvida ingen annan metod med sidoeffekt har anropats under tiden). Detta är uppenbart en tilltalande egenskap.

Sidoeffekter

Exempel:

```
A a = new A();
int r = a.m(1) – a.m(1);
```

Är `r = 0` efter detta? Om inte, är det svårt att utifrån koden resonera om programmet!

Vissa språk, s.k. funktionella språk, saknar sidoeffekter helt och hållet.

I objektorienterade språk är det acceptabelt att en mutator-metod har en sidoeffekt, nämligen att förändra tillståndet hos den implicita parametern, d.v.s. det objekt som metoden utförs på.

Sidoeffekter

En metod skulle även kunna förändra andra objekt än den implicita parametern

- explicit parametrar (de objekt som ges i metodens parameterlista)
- tillgängliga statiska variabler

Den som använder en metod förväntar sig inte att metoden förändrar de explicita parametrarna.

Har vi en metod `addAll` för att lägga till alla element i en lista till en annan lista och gör anropet

```
a.addAll(b);
```

förväntar vi oss att alla element i listan `b` lagts till listan `a`. Om metoden även skulle ändra innehållet i `b`, t.ex. ta bort alla element ur listan `b`, så skulle vi få en *oönskad sidoeffekt*.

Sidoeffekter

En annan typ av sidoeffekt är att förändra tillståndet i tillgängliga statiska variabler, t.ex `System.out`.

```
public void addMessage() {
  if (newMessage.isFull())
    System.out.println("Sorry – no space");
  . . .
}
```

Kasta istället en undantag för att rapportera att ett fel uppstått.

Undantag ger mycket större flexibilitet, eftersom användaren ges möjlighet att ta hand om felet.

Slutsats: Undvik oönskade sidoeffekter.

Accessor eller Mutator

En implikation av *The Separation of Concerns Principle* är att en metod inte både skall ändra tillståndet i ett objekt och returnera ett värde:

- en metod som returnerar information om ett objekt, dvs en access-metod, skall inte ändra tillståndet hos objektet, dvs samtidigt vara en mutator-metod.
- en mutator-metod skall inte returnera information om objektet, dvs metoden skall vara en **void**-metod.

The Command-Query Separation Principle:
En metod skall antingen vara en accessor eller en mutator, inte både och.

Accessor eller Mutator

Det finns många exempel på klasser i Javas standardbibliotek som inte följer denna princip.

```
Scanner sc = new Scanner("token1 token2 token3");
String t = sc.next();
```

Metoden `next()` returnerar nästa token. Metoden `next()` är således en access-metod, men metoden förändrar även tillståndet i hos `Scanner`-objektet. Nästa gång man anropar metoden `next()` får man en annan token. Således är metoden också en mutator-metod. Varför konstruktörerna gjort som de gjort, kan de endast själva svara på.

Accessor eller Mutator

I klassen `Stack` finns en metod `pop` som både returnerar och tar bort det översta elementet på stacken. I detta exempel kan man dock säga att orsaken är att metoden `pop()` med detta beteende är en vedertagen praxis (av historiska skäl).

Klassen `Stack` har dock även en metod `peek()` som returnerar det översta värdet på stacken.

Det kan ur bekvämlighetssynpunkt vara acceptabelt för användaren att låta en mutator-metod returnera ett värde, men då skall det även finnas en access-metod som returnerar samma värde utan att objektets tillstånd förändras.

Slutsats: Överallt där det är möjligt skall en metod antingen vara en access-metod eller mutator-metod.

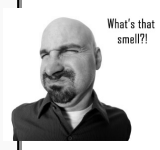
Kod på samma abstraktionsnivå

All kod i en metod skall exekvera på samma abstraktionsnivå.

Exempel: Kod på olika abstraktionsnivåer

```
public void doThisOrThat(boolean flag) {
   . . .
   if (flag) {
      doThis();
   }
   else {
      //twenty lines of cod to do that
   }
}//doThisOrThat

private void doThis() {
   //code to do this
}//doThis
```



Kontraktbaserad design

Betrakta klassen `MessageQueue`

```
public class MessageQueue {
   private Message[] elements;
   . . .
   public MessageQueue(int capacity) {. . .}
   public void add(Message m) { . . . }
   public void remove() { . . . }
   public Message peek() { . . . }
   public int size() { . . . }
   . . .
}//MessageQueue
```

Vad händer om en användare av denna klass försöker ta bort ett element från en tom kö?

Är inte detta dokumenterat så VET VI INTE (utan att läsa koden).

En implementation av `MessageQueue`

```
public class MessageQueue {
   private Message[] elements;
   private int head;
   private int tail;
   private int count;

   public MessageQueue(int capacity) {
      elements = new Message[capacity];
      count = 0;
      head = 0;
      tail = 0;
   }//constructor

   public void remove() {
      head = (head + 1) % elements.length;
      count--;
   }//remove

   public void add(Message m) {
      elements[tail] = m;
      tail = (tail + 1) % elements.length;
      count++;
   }//add

   public int size() {
      return count;
   }//size

   public Message peek() {
      return elements[head];
   }//peek
}//MessageQueue
```

Kön är realiserad med hjälp av ett cirkulärt fält.

Kontraktbaserad design: *Design by Contract*

När *design by contract* används, betraktas metoder som agenter som uppfyller ett kontrakt mot sina klienter. Detta kan jämföras med hur kontrakt upprättas vid andra typer av affärsförbindelser.

Det viktiga är att klassen som tillhandahåller en service och den som använder klassen skall ha en *formell överenskommelse*, i form av ett kontrakt, om hur klassen används.

Kontraktbaserad design

Kontraktet mellan en metod och den som anropar metoden specificerar vilka förpliktelser parterna har.

Kontraktet innehåller, förutom vad metoden gör, vilken indata metoden behöver och vilken utdata metoden returnerar, även

- förvillkor (*preconditons*)
- eftervillkor (*postconditions*)
- klassinvarianter (*class invariants*)

Design by Contract: *Don't accept anybody else's garbage*

Kontraktbaserad design

Ett förvillkor är ett villkor som måste gälla för att metoden skall hålla sin del av kontraktet

- är en förpliktelse som klienten åtar sig.

Ett eftervillkor är ett villkor som skall gälla när den efterfrågade servicen levereras.

- är en förpliktelse som metoden åtar sig.

En klassinvariant är ett villkor som anger vilka tillåtna tillstånd som ett objekt får befinna sig i.

- är en förpliktelse som metoden åtar sig.

Kontraktbaserad design

Klienten har skyldigheten att uppfylla förvillkoren innan metoden anropas.

- Om klienten uppfyller förvillkoren, har metoden skyldigheten att garantera eftervillkor och klassinvarianter.
- Om klienten inte uppfyller förvillkoren när metoden anropas, kan metoden i princip vidta de åtgärden den själv finner lämpliga oavsett hur katastrofala dessa kan bli för klienten.

Förvillkor

```
 /**
   Remove message at head of the queue.
     @pre size() > 0
**/
public void remove() {
   head = (head + 1) % elements.length;
   count--;
}//remove
```

En viktig egenskap för ett förvillkor är att det kan kontrolleras av klienten. Förvillkoret `size() > 0` i metoden `remove` är kontrollerbart eftersom klienten har tillgång till metoden `size()`.

Javadoc

Javadoc är ett verktyg som utifrån vissa kommentarer och koden i en Java klass skapar en dokumentation av klassen genom att generera en html-fil.

Kommentarer som är avsedda att ingå i dokumentationen skrivs enligt:

```
/**
  Detta hör till dokumentationen
**/
```

I en dokumentationskommentar kan man lägga till en uppsättning fördefinierade *annotationer*, t.ex:

```
/**
  @author       vem som skrivit klassen
  @version      text som avger vilken version
  @param        förklarning av inparametrar
  @return       förklarning av vad som returneras
  @throws       lista över vilka undantag som kastas
  @exception    anger när ett visst undantag kastas
  @see          hänvisning till andra klasser
*/
```

Javadoc

Man kan lägga till egna annotationer i Javadoc.

Vi kommer att använda annotationerna @pre och @post för att ange förvillkor respektive eftervillkor.

För att få med dessa i dokumentationen måste vi tala om för Javadoc dels att vi använder dessa notationer, dels vad vi använder dem till. Detta görs när Javadok kör genom att bifoga en argumentlista:

```
javadoc -tag pre:a:"Precondition:" -tag post:a:"Postcondition:" TheClass.java
```

Egendefinierade annotationer som inte anges i argumentlista utelämnas av Javadoc.

Förvillkor

```
/**
   Appends a message at the tail of the queue.
   @param aMassage the message to be appended
   @pre size() < elements.length
**/
public void add(Message aMessage) {
      elements[tail] = m;
      tail = (tail + 1) % elements.length;
      count++;
}//add
```

Kan *inte* kontrolleras av klienten

Här kan förvillkoret `size() < elements.length` inte kontrolleras av användaren, eftersom `elements` är en detalj i den privat implementation hos klassen `MessageQueue`.

Förvillkor

För att kunna ge ett kontrollerbart förvillkor till metoden `add` måste vi införa en ny metod i klassen `MessageQueue`:

```
/**
  @return  true if no more elements can
           be added, otherwise false
**/
public boolean isFull() {
   return count == elements.length;
}//isFull
```

Finns en sådan metod kan vi ange förvillkor enligt:

```
/**
   Appends a message at the tail of the queue.
   @param aMassage the message to be appended
   @pre !isFull()
**/
public void add(Message m) {
     elements[tail] = m;
     tail = (tail + 1) % elements.length;
     count++;
}//add
```

Kan kontrolleras av klienten

Assertions

Om klienten inte uppfyller förvillkoren kan en metod vidta de åtgärder som den själv finner lämpligast, utan hänsyn till vilka konsekvenser detta får för användaren.

Det enklaste är att inte göra något alls. Men detta kan bl.a. resultera i en besvärlig debugging.

> ***"Garbage in, garbage out"* är ingen bra filosofi.**

För att kunna uppmärksamma klienten på att förvillkoret inte är giltigt tillhandahåller Java en speciell konstruktion; **assertion**-mekanismen.

Assertions

Satsen

assert condition;

kontrollerar huruvida villkoret `condition` är sant eller inte.

Om villkoret är sant fortsätter exekveringen normalt. Men om villkoret är falskt kastas ett `AssertionError`. Normalt kommer sedan programmet att terminera.

Det finns ytterligare en variant av **assert**-satsen, i vilken man kan ge en förklaring till det inträffade som bifogas `AssertionError`-objektet:

assert condition: explanation;

där *explanation* är en `String`.

Assertions

```
/**
   Remove message at head of the queue.
   @pre size() > 0
**/
public void remove() {
   assert count > 0 : "Violated precondition size() > 0. ";
   head = (head + 1) % elements.length;
   count--;
}//remove
```

För att assertionkontrollen skall göras måste programmet köras med en speciell flagga

java -enableassertion TheProgramToRun

Används inte denna flagga ignoreras alla **assert**-satser.

Observera att **assertion**-mekanismen är ett debug-verktyg som används under utvecklingsfasen, inte under drift.

Exceptions i kontraktet

Ett vanligt sätt att handha problem är att kasta ett undantag (*exception*).

Undantag är ofta en del i ett kontrakt. Nedan ges ett exempel från Javas klassbibliotek.

```
/**
   Creates a new FileReader, given the name of file to read from.
   @param fileName- the name of file to read from
   @throw FileNotFoundException - if the named file does not exist,
   is a directory rather than a regular file, or for some other reason cannot
   be opened for reading.
*/
public FileReader readFile(String fileName) throws FileNotFoundException {
   . . .
}//readFile
```

Som vi ser lovar metoden att kasta ett undantag av typen `FileNotFoundException` om det inte finns någon fil med angivet namn.

Exceptions i kontraktet

Det är en viktig skillnad mellan ett förvillkor och ett undantag som specificeras i kontraktet.

Metoden `readFile` ovan har inget förvillkor. Men här ges ett tydligt löfte om vad som kommer att göras om det inte finns en fil med angivet filnamn, nämligen att kasta ett undantag av typen `FileNotFoundException`.

Varför ges inte förvillkoret:

@pre fileName must be the name of a valid file

Eftervillkor

```
/**
    Appends a message at the tail of the queue.
    @param aMessage the message to be appended
    @pre !isFull()
    @post size() > 0
**/
public void add(Message aMessage) {
   . . .
}//add
```

Metoden `add` har eftervillkoret `size() > 0`. Detta villkor är användbart eftersom det implicerar förvillkoret till metoden `remove`. Efter att man har lagt till ett element med metoden `add`, är det alltid säkert att anropa metoden `remove`:

```
q.add(m);
//Postcondition of add: q.size() > 0
//Precondition of remove: q.size() > 0
m = q.remove();
```

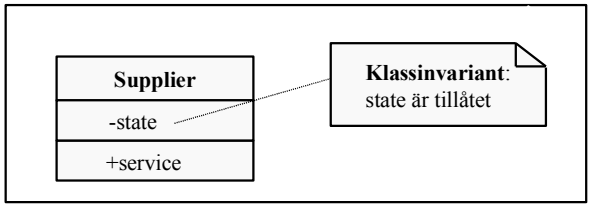
Vilka eftervillkor har metoden remove?

Klassinvarianter

En klassinvariant är ett logiskt villkor som håller för alla objekt av klassen.

En klassvariant:

- beskriver vilka tillåtna tillstånd objekten kan ha
- måste vara sant före och efter det att en metod anropas
- kan temporärt åsidosättas inuti metoden.



Klassinvarianter

Här är en klassinvariant för implementationen av `MessageQueue`:

head >= 0 && head < elements.length

För att bevisa en invariant måste man undersöka att

1. villkoret är sant efter att varje konstruktor har utförts
2. villkoret bevaras av varje mutator.

Vi bryr oss inte om accessorer eftersom dessa inte förändrar objektets tillstånd.

Punkt 1 ovan garanterar att inga objekt med ogiltiga tillstånd kan skapas. Således vet vi att tillståndet hos objektet är giltigt första gången en mutator appliceras.

Punkt 2 ovan garanterar att objektet har ett giltigt tillstånd efter att den första mutatorn har avslutats. Med samma logik måste den andra mutatorn bevara invariantvillkoret, osv.

Klassinvarianter

Gäller invarianten efter att konstruktorn i `MessageQueue` exekverats?

```
/**
Constructs an empty queue.
@param capacity the maximum size of the queue
**/
public MessageQueue(int capacity) {
    elements = new Message[capacity];
    count = 0;
    head = 0;
    tail = 0;
}//constructor
```

Villkoret `head >= 0` är sant, eftersom `head` har satts till 0. Men villkoret `head < elements.length` kan vi inte garantera!

Varför inte?

Klassinvarianter

För att kunna garantera att `head < elements.length` måste vi ge konstruktorn ett förvillkor:

```
/**
Constructs an empty queue.
@param capacity the maximum size of the queue
@pre capacity > 0
**/
public MessageQueue(int capacity) {
    elements = new Message[capacity];
    count = 0;
    head = 0;
    tail = 0;
}//constructor
```

Nu vet vi att `elements.length` måste vara > 0. Därför är invarianten sann vid slutet av konstruktorn.

Klassinvarianter

Det finns endast en metod i klassen `MessageQueue` som ändrar värdet av `head`, nämligen `remove`. Vi måste visa att `remove` bevarar invarianten.

```
/**
   Remove message at head of the queue.
   @pre size() > 0
   @post size() >= 0
**/
public void remove() {
   head = (head + 1) % elements.length;
   count--;
}//remove
```

Klassinvarianter

Metoden beräknar tilldelningen:

$$head_{new} = (head_{old} + 1) \% elements.length$$

Här betecknar $head_{old}$ värdet av `head` innan metoden anropas och $head_{new}$ betecknar värdet av `head` efter att metoden avslutats. Eftersom vi antar att $head_{old}$ uppfyller invarianten när metoden startar vet vi att

$$head_{old} + 1 > 0$$

Således

$$head_{new} = (head_{old} + 1) \% elements.length >= 0$$

Från definitionen av operatorn %, följer att

$$head_{new} < elements.length$$

Vi kan nu dra slutsatsen att varje access av formen `elements[head]` är giltig.

Klassinvarianter

Vi kan ge ytterligare två klassinvarianter för `MessageQueue`:

tail >= 0 && tail < elements.length
count >= 0 && count <= elements.length

vilka också är enkla att visa. Gör vi det har vi bl.a. *bevisat* att det i klassen `MessageQueue` aldrig inträffar att adressering sker utanför indexgränserna i fältet `elements`.

De invarianter vi visat här är mycket enkla, men de är också mycket typiska. Så länge som instansvariablerna i en klass är privata, har klassen full kontroll över alla metoder som modifierar instansvariablerna. Man kan vanligtvis garantera att vissa värden ligger inom giltiga intervall, eller att vissa referenser aldrig är **null**.

Klassinvarianter

Vi kan särskilja två typer av invarianter;

- gränsnittsinvarianter
- implementationsinvarianter

Implementationsinvarianter involverar detaljer i implementationen och används av den som implementerar eller underhåller klassen för att tillförsäkra korrektheten hos de implementerade algoritmerna.

Gränsnittsinvarianter är villkor som endast berör det publika gränssnittet till klassen och är av intresse för den som använder klassen, eftersom de ger garantier för hur alla objekt av denna klass beter sig.

Gränsnittsinvarianter måste uttryckas i termer av det publika gränssnittet för klassen. Till exempel

getMinute() >= 0 && getMinute() < 60

Klassinvarianter

Varje klass har sina egna invarianter:

```
protected boolean invariant() {
  return hour >= 0 && hour < 24 &&
         minute >=0 && minute < 60 &&
         second >= 0 && second < 60;
}//invariant
```

Varför **protected**?

```
/**
   . . .
   @pre . . .
   @post . . .
**/
public void addTime(Time increaseTime) {
  //code
  assert invariant();
}//addTime
```

Kontrollera klassens invarianter!

Kontraktbaserad design och arv

Förvillkoren för en metod i en subklass får *inte vara starkare* än förvillkoren hos superklassen eller hos interfacet som implementeras av klassen. Subklassen får med andra ord inte ställa hårdare krav på en klient än vad superklassen eller interfacet gör.

Eftervillkoren för en metod i en subklass får *inte vara svagare* än eftervillkoren hos superklassen eller hos interfacet som implementeras av klassen. Subklassen får med andra ord inte göra mindre för en klient än vad superklassen eller interfacet gör.

Kontraktbaserad design och LSP

Med hjälp av för- och eftervillkor kan vi uttrycka *Liskov Substitution Principle* på följande sätt:

> Det är acceptabelt att göra klassen S till en subklass av klassen T om och endast om det för varje publik metod med identiska signaturer i S och T gäller att S's metoder inte har starkare förvillkor än T's metoder och att S's metoder inte har svagare eftervillkor än T's metoder.

Gäller alltså både instansmetoder (*overriding*) och statiska metod (*hiding)*.

Kontraktbaserad design: Fördelar

Formaliserar vilka förpliktelser som klient respektive server har.

Förtydligar ansvarsfördelningen.

Innebär implicit att varje publik metod dokumenteras, vilket ger klienterna den information de behöver för att använda metoderna.

Ger en bättre förståelse för programutveckling.

Förenklar designen.

Underlättar debugging, testning och kvalitetssäkring.

Kontraktbaserad design: Sammanfattning

- Specificera alla för- och eftervillkor för alla publika metod i den externa dokumentationen. Dessa villkor definierar kontraktet för metoderna.
- Användaren av klassen måste kunna kontrollera alla förvillkor som en metod har. Förvillkor för publika metoder kan endast involvera de publika metoderna i klassen.
- Klienten lovar att uppfylla förvillkoren.
- Uppfylls förvillkoren lovar metoden att uppfylla eftervillkoren när metoden terminerar.
- Att lägga till ett förvillkor gör *specifikationen svagare*, eftersom den blir enklare att uppfylla för servern.
- Att lägga till ett eftervillkor gör *specifikationen starkare*, eftersom den blir svårare att uppfylla för servern.
- Förvillkor och klassinvarianter kontrolleras med assertions.

Självdokumenterande kod

- Sträva alltid efter att skriva självdokumenterande kod, d.vs. kod som är så lätt att läsa och förstå att inga kommentarer behövs:
  - använd beskrivande namn
  - följ *The Separation of Concerns Principle*
  - använd *Design by Contract*.
- Om din kod är så invecklad och krånglig att den erfordrar förklarande kommentarer, så skall koden troligen skrivas om.
- Skriv kommentarerna före eller samtidig med koden, inte efter.

  Build it in, don't add it on.
- Den externa dokumentationen av en metod skall vara tillräckligt specifik för att utesluta implementeringar som inte är godtagbara, men tillräckligt generell för att tillåta alla implementationer som är godtagbara.

Refactoring: Ersätt temporär variabel med metod

```
public double getPrice() {
  int basePrice = quantity * itemPrice;
  double discountFactor;
  if (basePrice > 1000)
     discountFactor = 0.95;
  else
     discountFactor = 0.98;
  return basePrice * discountFactor;
}//getPrice
```

```
public double getPrice() {
   return basePrice() * discountFactor();
}//getPrice

private double basePrice() {
   return quantity * itemPrice;
}//basePrice

private double discountFactor() {
    if  (basePrice() > 1000)
      return 0.95;
   else
      return 0.98;
}//discountFactor
```

Refactoring: Ersätt magiska tal med symboliska konstanter

```
public double potentialEnergy(double mass, double height) {
   return (mass * 9.81 * height);
}//potentialEnergy
```

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
public double potentialEnergy(double mass, double height) {
   return (mass * GRAVITATIONAL_CONSTANT * height);
}//potentialEnergy
```