Föreläsning 6

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

Refactoring: Döp om identifierare

```
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?

1	2	3	4	5	6	7	8	9	10
7 /	X	7 2	6 /	9 /	X	X	8 /	X	X 8 1
20	39	48	67	87	115	135	155	183	202

Funktionell dekomposition

Funktionell dekomposition innebär att koden bryts ner i ändamåls-enliga metoder.

Fördelar:

- ***underlättar läsbarheten***. Ett beskrivande namn på en metod är lättare att läsa än 10 rader kod.
- ***ökar abstraktionsnivån***. Programmeraren kan betrakta metoder som om de vore inbyggda högnivåkonstruktioner i program-språket, och kan på så vis sättas sig in i mycket större mängder kod.
- ***reducerar duplicering av kod***. Kodavsnitt som upprepas på flera ställen, skall brytas ut till en metod och 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);
   }
   . . .
}
```

Funktionell dekomposition

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

Refactoring: Extrahera metod

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

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);
   }
   . . .
}
```

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

Refactoring: Ersätt felkoder med exceptions

Funktionell dekomposition

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

Refactoring: Extrahera metoder

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

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


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 {
      //thirty lines of cod to do that
   }
}
```

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();
   }
}
```

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

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

Refactoring: Extrahera metoder

The Separation of Concerns Principle

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

Metoden pay löper igenom alla anställda och för varje anställd

1. kontrollerar om utbetalning skall ske
2. beräknar hur stor 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];
}
```

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

Sidoeffekter

En sidoeffekt av ett metodanrop är varje modifikation av data, orsakad av metodens exekvering, som är *observerbar utifrån*.

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 muterande metoder har sidoeffekter, nämligen att förändra tillståndet hos den implicita parametern, d.v.s. det objekt som metoden utförs på (det *anropande objektet*).

Sidoeffekter

En metod kan även förändra andra objekt än den implicita parametern

- explicita parametrar (de objekt som ges i metodens parameterlista)
- tillgängliga klassvariabler

Den som använder en metod förväntar sig oftast 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 troligen att alla element i listan b lagts till i listan a. Om metoden även ändrar innehållet i b, t.ex. tar bort alla element ur listan b, så är det en *sidoeffekt*. Om sådant beteende inte är tydligt dokumenterat och vedertaget får en sådan sidoeffekt betraktas som icke önskvärd.

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 muterande metod.
- en muterande 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 *The Command-Query Separation Principle*.

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

Metoden next() returnerar nästa token. next() är således en accessmetod, men metoden förändrar även tillståndet hos Scanner-objektet. Nästa anrop av next() ger en annan token. Således är metoden också muterande.

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 pop() med detta beteende är en vedertagen praxis (av historiska skäl).

Klassen Stack har dock även accessmetoden peek() som returnerar det översta värdet på stacken – utan att ta bort det.

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

Slutsats: Överallt där det är möjligt skall en metod antingen vara en accessmetod eller en muterande metod.

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() { . . . }
   . . .
}
```

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

Är det inte 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;
   }

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

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

   public int size() {
      return count;
   }

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

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 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 beskriver vilka förpliktelser parterna har.

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

- förvillkor (*preconditons*)
- eftervillkor (*postconditions*)

Kontraktet för en metod utgör metodens *specifikation*.

***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.

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 åtgärder 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--;
}
```

Förvillkor måste kunna 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 version
  @param       förklarning av inparametrar
  @return      förklarning av vad som returneras
  @throws      lista över vilka undantag som kan kastas
  @exception   anger varför 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 aMessage the message to be appended
   @pre size() < elements.length
*/
public void add(Message aMessage) {
      elements[tail] = m;
      tail = (tail + 1) % elements.length;
      count++;
}
```

Kan *inte* kontrolleras av klienten

Här kan förvillkoret size() < elements.length inte kontrolleras av användaren, eftersom elements är en privat detalj i implementation av 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;
}
```

Finns en sådan metod kan vi ange förvillkor enligt:

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

Kan kontrolleras av klienten

Assertions

Om klienten inte uppfyller förvillkoren kan en metod vidta vilka åtgärder den vill, utan hänsyn till konsekvenserna för klienten.

Det enklaste är att inte göra något alls. Men det kan bl.a. resultera i en besvärlig felsökning (för klienten).

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

För att kunna uppmärksamma klienten på att förvillkoret inte är uppfyllt har Java en speciell konstruktion: **assertion**-mekanismen.

Filosofin vi använder här är *fail fast*. När ett fel inträffar rapporteras fel omedelbart och exekveringen avbryts. Detta i syfte att hitta och avlägsna buggar så tidigt som möjligt.

Assertions

Satsen

```
assert condition;
```

kontrollerar om villkoret condition är sant eller inte.

Om villkoret är sant fortsätter exekveringen normalt. 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 the head of the queue.
   @pre size() > 0
*/
public void remove() {
   assert count > 0 : "Violated precondition size() > 0.";
   head = (head + 1) % elements.length;
   count--;
}
```

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

```
java -enableassertions 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 hantera problem är att kasta ett undantag (*exception*).

Kastas undantag är dessa 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 {
   . . .
}
```

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 att specificera ett förvillkor och ett undantag i kontraktet.

Metoden readFile ovan har inget förvillkor. Men här ges ett tydligt besked 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) {
   . . .
}
```

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();
```

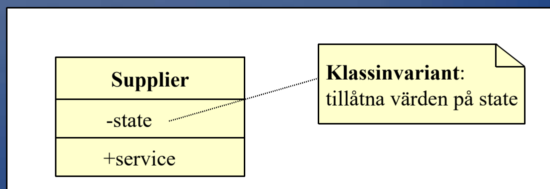
Vilka eftervillkor har metoden remove?

Klassinvarianter

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

En klassinvariant:

- beskriver vilka tillåtna tillstånd objekten kan ha
- måste vara sant före och efter ett metodanrop
- 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 kontrollera att

1. villkoret är sant efter att varje konstruktor har exekverat
2. villkoret bevaras av varje muterande metod

Vi bortser från accessmetoder eftersom dessa inte förändrar objektets tillstånd.

Punkt 1 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 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;
}
```

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;
}
```

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--;
}
```


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 returnerat. Eftersom vi antar att $head_{old}$ uppfyller invarianten när metoden startar vet vi att

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

Från definitionen av operatorn % följer att

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

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

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

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. De är också mycket typiska (så länge instansvariablerna endast kan modifieras via klassens metoder*). Man kan då vanligtvis garantera att vissa värden ligger inom giltiga intervall, eller att vissa referenser aldrig är **null**.

*Vi kommer senare i kursen att tala om hur den interna representationen i en klass/objekt kan exponeras.

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 garantera 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
```


Assertions: Sammanfattning

Assertions är ett verktyg för debugging.

Fail fast - ju tidigare en bugg upptäcks ju lättare kan den avhjälpas.

```
/**
   @pre x >= 0
   @return  square root of x
*/
public static double sqrt(double x) {
    assert x >= 0;
    double root;
    //code to compute root
    assert Math.abs(x - root * root) < 0.0001;
    return root
}
```


Kontraktbaserad design och arv

Då arv används uppkommer ofta situationen att objekt av en viss klass kan ersättas med subklassobjekt. Vi måste diskutera hur kontrakten för subklassens metoder förhåller sig till kontrakten för basklassmetoderna. *Vilka begränsningar gäller för överskuggade metoders för- och eftervilkor?*

Kontraktet för en metod utgör metodens *specifikation*.

- Att stärka förvillkoret gör *specifikationen svagare*.
- Att stärka eftervillkoret gör *specifikationen starkare.*

Kontraktbaserad design och arv

Villkoret A är *starkare* än villkoret B omm A => B (A implicerar B).

Implikation		
A	B	A => B
T	T	T
T	F	F
F	T	T
F	F	T

False är det *starkaste* villkoret eftersom false => A för alla villkor A.
True är det *svagaste* villkoret eftersom A => true för alla villkor A.

Starkaste villkoret		
F	A	F => A
F	T	T
F	F	T

Svagaste villkoret		
A	T	A => T
T	T	T
F	T	T

Kontraktbaserad design och arv

Konjunktion		
A	B	A & B
T	T	T
T	F	F
F	T	F
F	F	F

Disjunktion		
A	B	A | B
T	T	T
T	F	T
F	T	T
F	F	F

Exempel: A & B => A
A & B => B
A => A | B
B => A | B

Kontraktbaserad design och arv

Starkt villkor => *Svagt villkor*
Pre$_1$ => *Pre*$_2$
Post$_1$ <= **Post**$_2$

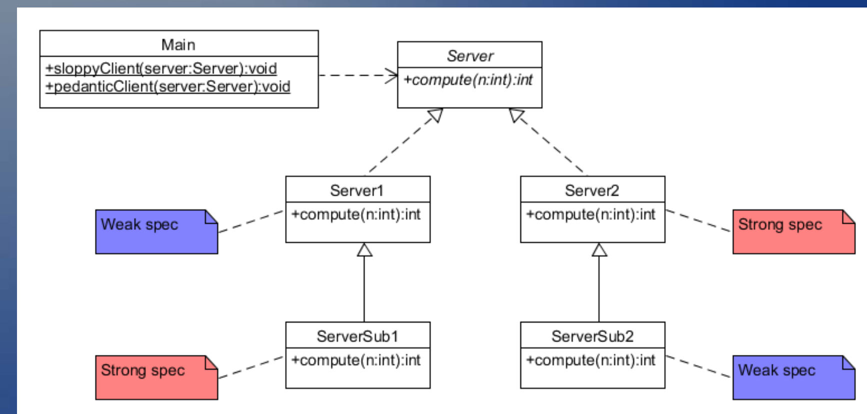
Spec$_1$
Pre$_1$
Post$_1$

Spec$_2$
Pre$_2$
Post$_2$

Kontraktbaserad design och arv

Exempel: Plug and Play.

En klient läser in heltal och anlitar en server för att beräkna n! (n-fakultet)
Vilka krav måste ställas på servern för att den skall vara utbytbar?



Kontraktbaserad design och arv

Två klienter med olika ambitionsnivå

```
public interface Server {
    int compute(int n);
}
```

Ingen indatakontroll

```
public static void sloppyClient(Server server) {
    Scanner in = new Scanner(System.in);
    while ( in.hasNextInt() ) {
        int n = in.nextInt();
        System.out.println(server.compute(n));
    }
}
```

```
public static void pedanticClient(Server server) {
    Scanner in = new Scanner(System.in);
    while ( in.hasNextInt() ) {
        int n = in.nextInt();
        if ( n >= 0 )
            System.out.println(server.compute(n));
        else
            System.out.println("Negative input value, try again!");
    }
}
```

Kontrollerar indata

Kontraktbaserad design och arv

Server1 och ServerSub1

svag spec

```
public class Server1
implements Server {
  /**
   * @Pre  n >= 0
   * @Post n >= 0 => return n!
   */
  public int compute(int n) {
      int result = 1;
      while ( n != 0 ) {
          result *= n;
          n--;
      }
      return result;
  }
}
```

stark spec

```
public class ServerSub1
extends Server1 {
  /**
   * @Pre  True
   * @Post n >= 0 => return n! &&
   *         n < 0 => return -1
   */
  public int compute(int n) {
      if ( n < 0 )
          return -1;
      else {
          int result = 1;
          while ( n != 0 ) {
              result *= n;
              n--;
          }
          return result;
      }
  }
}
```


Kontraktbaserad design och arv

Server2 och ServerSub2

stark spec

```
public class Server2
implements Server {
  /**
   * @Pre  True
   * @Post n >= 0 => return n! &&
   *         n < 0 => return -1
   */
  public int compute(int n) {
      if ( n < 0 )
          return -1;
      else {
          int result = 1;
          while ( n != 0 ) {
              result *= n;
              n--;
          }
          return result;
      }
  }
}
```

svag spec

```
public class ServerSub2
extends Server2 {
  /**
   * @Pre  n >= 0
   * @Post n >= 0 => return n!
   */
  public int compute(int n) {
      int result = 1;
      while ( n != 0 ) {
          result *= n;
          n--;
      }
      return result;
  }
}
```


Kontraktbaserad design och arv

Vad händer om servern byts ut mot en server där compute har
- starkare specifikation?
- svagare specifikation?

```
pedanticClient(new Server1());    // OK
pedanticClient(new ServerSub1()); // OK
```

ServerSub1.compute har **starkare** specifikation än Server1.compute

```
sloppyClient(new Server2());      // OK
sloppyClient(new ServerSub2());   // Infinite loop!
```

ServerSub2.compute har *svagare* specifikation än Server2.compute

Kontraktbaserad design och arv

Förvillkoret för en metod i en subklass får *inte vara starkare* än förvillkoret hos superklassen. Subklassen får med andra ord inte ställa hårdare krav på en klient än vad superklassen gör.

Eftervillkoret för en metod i en subklass får *inte vara svagare* än eftervillkoret hos superklassen. Subklassen får med andra ord inte göra mindre för en klient än vad superklassen 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 till 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 både instansmetoder (*overriding*) och klassmetoder (*hiding*).

Kontraktbaserad design och LSP

Ovan sagda innebär att *specifikationen* för en överskuggad metod måste vara *minst lika stark* i subklassen som i superklassen:

Metod A är utbytbar mot (kan användas i stället för) metod B, om specifikationen för A är minst lika stark som specifikationen för B.

Det är acceptabelt att göra klassen S till en subklass till klassen T om och endast om det för varje publik metod med identiska signaturer i S och T gäller att S:s metoder har minst lika starka specifikationer som T:s metoder.

Starkare och svagare specifikationer

Vilken metod har starkast specifikation?

```
/**
  @pre val occurs exactly once in a
  @return index i such that a[i] = val
*/
public static int find1(int[] a, int val)
```

```
/**
  @pre val occurs in a
  @return index i such that a[i] = val
*/
public static int find2(int[] a, int val)
```

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 metoder 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 skärpa förvillkoren gör *specifikationen svagare*, eftersom den blir enklare att uppfylla för servern (och svårare att uppfylla för klienten).
- Att skärpa eftervillkoren gör *specifikationen starkare*, eftersom den blir svårare att uppfylla för servern (men enklare att uppfylla för klienten).
- 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.