Länkade listor och automatisk testning

Algoritmer och datastrukturer

Obligatorisk Laboration nr 3

Syfte

Att ge träning i programmering av länkade listor på låg abstraktionsnivå med primitiv pekarmanipulering. Även om programmen här skrivs i Java är principerna tillämpliga på andra språk, t.ex. C och C++. Dessutom ger laborationen tillfälle att tillämpa automatisk testning på den utvecklade koden. Behovet av testning är stort eftersom programkod med pekarhantering ofta innehåller fel.

Mål

Efter övningen skall du

- kunna arbeta med länkade listor utan ”skyddsnät” i form av bekväma standardklasser.
- inse nackdelar och risker med pekarmanipulering.
- kunna genomföra enkel verktygsstödd testning.
- inse fördelar men också begränsningar med testning.

Litteratur

Uppgifterna bygger på exemplen i föreläsningarna om länkade listor, nr 5 och nr 6. Se för övrigt kurs-PM för litteraturhänvisning till kursboken.

Färdig programkod

Given programkod som behövs i uppgifterna finns på kurshemsidan under fliken **Laborationer**.

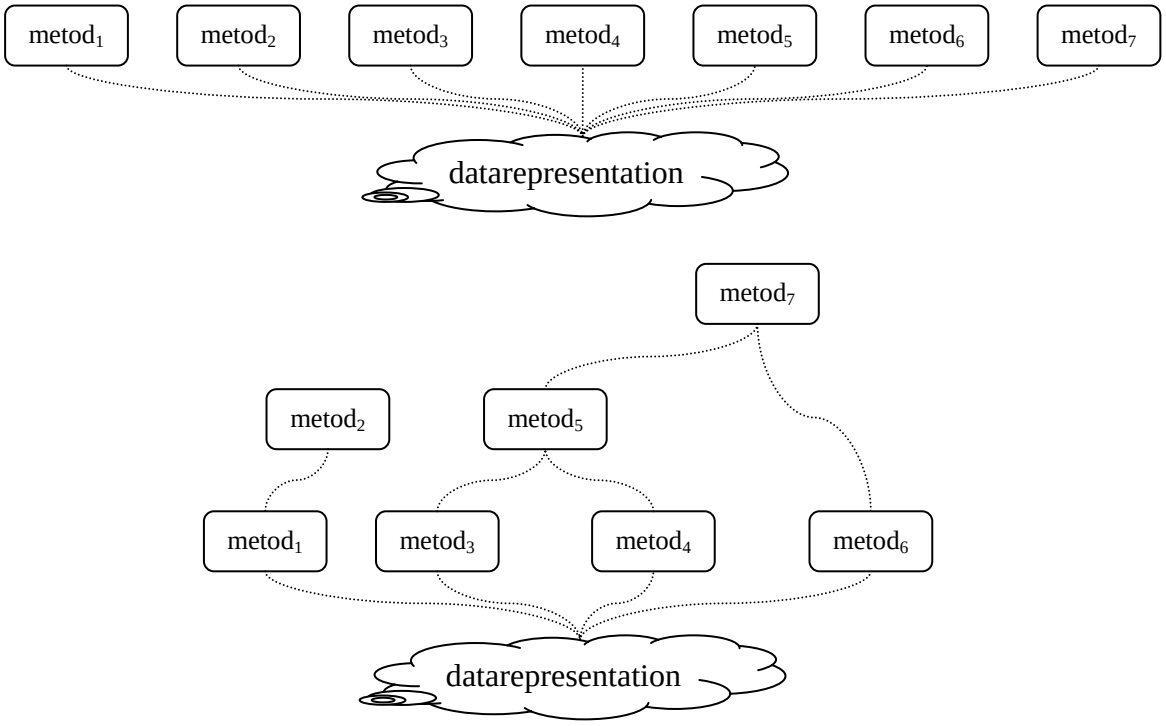
Genomförande och redovisning

Uppgifterna är obligatoriska.

- Redovisning sker i grupper om två personer – inte färre, inte fler.
- Varje grupp skall självständigt utarbeta sin egen lösning.
  Samarbete mellan grupperna om övergripande principer är tillåtet, men inte plagiering.
- Redovisa filerna enligt specifikationen på sista sidan, samt `README.txt` med allmänna kommentarer om lösningen.
- Senaste redovisningsdag, se Fire.

Uppgifter

Klassen `Lists` gicks delvis igenom på föreläsningen om länkade listor. Uppgiften är att skriva färdigt klassen genom att implementera resterande metoder. Lösningen skall utföras ”i samma anda” som på föreläsningen, t.ex. skall klassen `ListNode` se ut som nedan utan några konstruktorer. Alla listor skall vara enkellänkade och ha listhuvud. Metoderna måste klara att hantera tomma listor, d.v.s. listor som enbart består av ett listhuvud. Om en metod anropas med **`null`** som argument skall undantaget `ListsException` med texten ”`Lists: null passed to` *metodnamn*” kastas. Det rekommenderas att, om möjligt, implementera metoder med hjälp av andra metoder i klassen. Istället för att låta alla metoder arbeta direkt på ”pekarnivå” som i den övre delen av bilden nedan, kan kanske vissa av dem skrivas med hjälp av andra, eller en kombination av sätten. På så sätt fås en skiktad arkitektur med ett bottenlager med låg abstraktionsnivå, och ovanpå det ett eller flera lager med högre abstraktionsnivå.



Om t.ex. metod$_7$ kan implementeras med anrop av metod$_5$ och metod$_6$ borde den kunna göras enklare än om alla steg utförs i detalj. Det är *inte* tillåtet att ta genvägar via strängoperationer, t.ex. skulle `reverse` kunna implementeras genom att först översätta listan till en sträng, vända strängen baklänges, för att därefter översätta den omvända strängen till en lista. En sådan lösningsmetod vore inte skalbar eftersom den bara fungerar för teckenlistor. Anledningen till att **`char`** valts som elementtyp är enbart för att förenkla exempel och testning.

```
public class ListNode {
     public char element;
     public ListNode next;
}

public class Lists {
    // Givna metoder från föreläsningen
    public static ListNode toList(String chars)
    public static ListNode copy(ListNode l)
    public static ListNode removeAll(ListNode l,char c)

    // Uppgift: implementera och testa dessa
    public static String toString(ListNode l)
    public static boolean contains(ListNode l,char c)
    public static ListNode copyUpperCase(ListNode l)
    public static ListNode addFirst(ListNode l,char c)
    private static ListNode getLastNode(ListNode l)
    public static ListNode addLast(ListNode l,char c)
    public static ListNode concat(ListNode l1,ListNode l2)
    public static ListNode addAll(ListNode l1,ListNode l2)
    public static ListNode reverse(ListNode l)
}
```

De nya metoderna

`public static String toString(ListNode l)`
Returnerar en sträng med tecknen i `l`. Metoden muterar ej `l`.

`public static boolean contains(ListNode l,char c)`
Returnerar `true` om `l` innehåller `c`, `false` annars. Metoden muterar ej `l`.

`public static ListNode copyUpperCase(ListNode l)`
Returnerar en ny lista med alla tecken i `l` som är stora bokstäver (A-Z).
Metoden muterar ej `l`.

`public static ListNode addFirst(ListNode l,char c)`
Adderar `c` först i `l`. Metoden muterar `l` och returnerar en referens till `l`.

`private static ListNode getLastNode(ListNode l)`
Returnerar en referens till den sista noden i `l` (listhuvudet om `l` refererar till en tom lista.)
Metoden muterar ej `l`.

`public static ListNode addLast(ListNode l,char c)`
Adderar `c` sist i `l`. Metoden muterar `l` och returnerar en referens till `l`.

`public static ListNode concat(ListNode l1,ListNode l2)`
Sätter samman `l1` med `l2` så att alla noderna i `l2` kommer efter noderna i `l1`.
Efter operationen refererar `l2` till en tom lista. Metoden muterar både `l1` och `l2`,
och returnerar en referens till `l1`.
Exempel:

före		efter	
`l1 = [a,b,c]`	`l2 = [d,e]`	`l1 = [a,b,c,d,e]`	`l2 = []`

`public static ListNode addAll(ListNode l1,ListNode l2)`

Adderar alla elementen i `l2` till slutet av `l1`. Metoden muterar `l1` men ej `l2`, och returnerar en referens till `l1`.
Exempel:

```
före                                  efter
l1 = [a,b,c]  l2 = [d,e]    l1 = [a,b,c,d,e]  l2 = [d,e]
```

`public static ListNode reverse(ListNode l)`

Returnerar en ny lista med elementen i `l` i omvänd ordning.
Metoden skall ej mutera `l`.

Anm. Förvissa dig om att du förstått skillnaden mellan metoderna `concat` och `addAll`.

Arbetsgång

I den givna koden finns metodstumpar för metoderna ovan, utöver de kompletta som gicks igenom på föreläsningen. Utveckla och testa en metod i taget, lämpligen i ordning uppifrån och ner i uppräkningen ovan. *Fortsätt aldrig med nästa metod innan den föregående passerat alla relevanta test* (se nedan)!

Testning av listklassen

Enkla test

Klassen `SimpleListTest` testar några enkla fall och skriver ut resultaten i textfönstret. Klassen har en `main`-metod som kör testfallen. Avkommentera testfall i koden allteftersom du implementerar respektive metoder i `Lists`. Facit till testen finns på sista sidan.

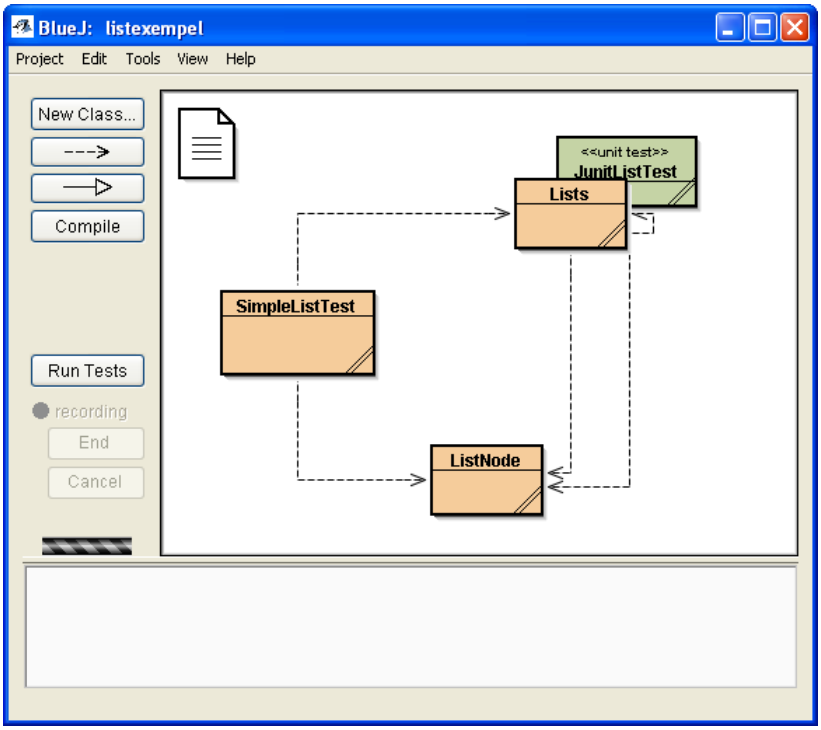
Regressionstestning med JUnit

Vid testning med testramverket JUnit definieras en speciell testklass för varje klass som skall testas. I vårt exempel heter testklassen `JunitListTest`.[1] Testklassen innehåller testmetoder som testar metoderna i `Lists`. Test av en metod kan innebära att även andra metoder i klassen anropas. Man kan exekvera en testmetod åt gången, eller samtliga testmetoder i en följd. Hela testsviten kan då enkelt upprepas automatiskt med en enkel knapptryckning efter varje förändring av koden, vilket är en stor arbetsbesparing jämfört med manuella metoder. Detta kallas *regressionstestning*. Men kom ihåg: *Testning kan påvisa närvaron av fel – men i allmänhet inte bevisa frånvaron av fel*. Testklassen är färdig och de givna testmetoderna får inte ändras, men definiera gärna nya. Nedan beskrivs först kortfattat hur man testar med BlueJ eller eclipse. Om du föredrar eclipse kan du hoppa över avsnittet om BlueJ.

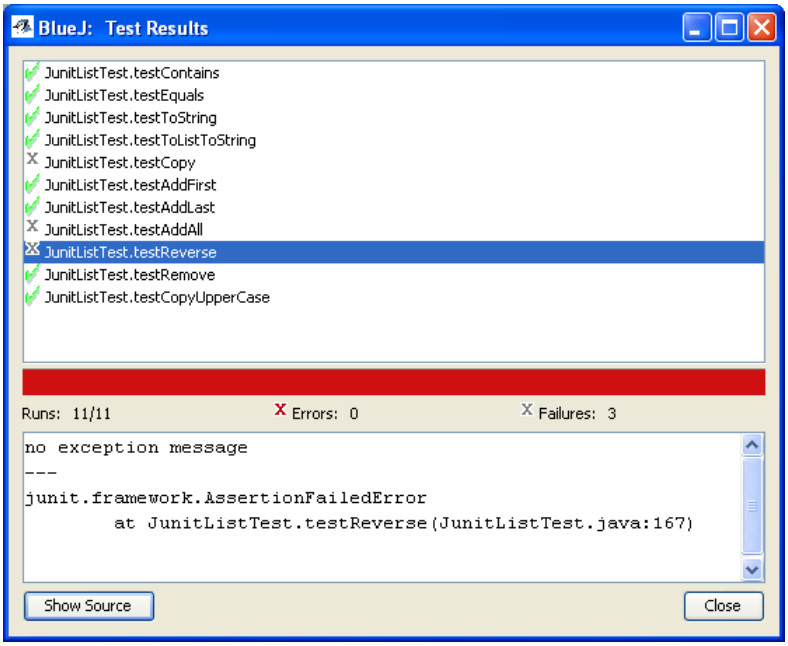
[1] Testen i den här laborationen är skrivna i version 4 av JUnit.

JUnit-testning med BlueJ

Koden som distribueras för laborationen är utvecklad med javaverktyget BlueJ som är mycket användarvänligt. Det finns installerat på skolans datorer och kan enkelt installeras på din egen dator, se länk på kurshemsidan. En av fördelarna med BlueJ är att verktyget stöder regressionstestning med JUnit.
Ett BlueJ-projekt består av en filkatalog med java- och klassfiler, samt några extra verktygsspecifika filer. Det enklaste sättet att starta programmet för att arbeta med ett färdigt projekt är att dubbelklicka på filnamnet `bluej.pgk`. Huvudfönstret i BlueJ visar klasserna i ett förenklat UML-diagram. Källtextfönster för klasserna öppnas genom att dubbelklicka på klassikonerna i diagrammet. Efter förändringar i koden streckmarkeras ikonerna för alla berörda klasser. Omkompilering görs då enklast med ctrl-K. Testverktygen visas genom att välja View->Show Team and Test Controls. Testklassens ikon visas i exemplet nedan snett bakom ikonen för `Lists`.



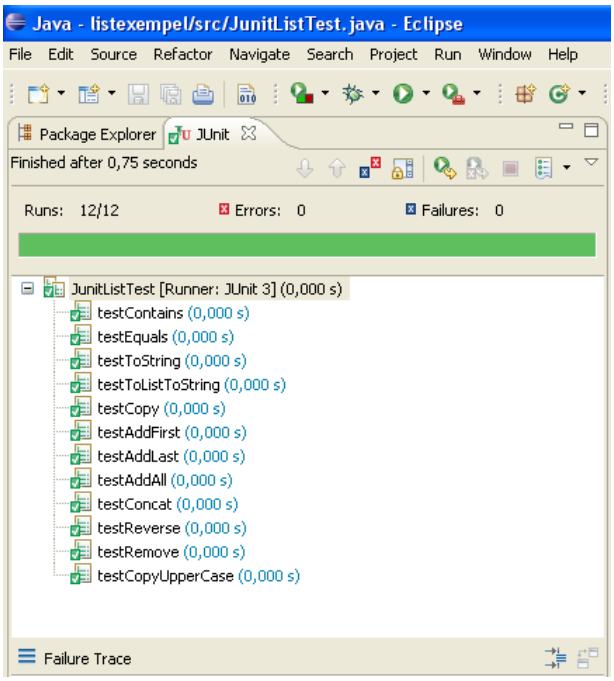
För att köra alla testmetoderna i en automatiserad följd trycker man på knappen Run Tests och får då upp ett fönster som sammanfattar resultaten



Tre typer av testresultat visas: En grön bock anger lyckat test, ett grått kryss (Failures) anger att testet misslyckades därför att något villkor ej uppfylldes, och ett rött kryss (Errors) att exekveringen avbröts, t.ex. p.g.a. ett ofångat undantag, indexfel i fält etc. Vid ett dubbelklick på en rad i listan visas information om vilken kodrad i testmetoden som fallerade. Man kan då starta avlusaren och sätta en brytpunkt på raden i fråga, och därefter exekvera den specifika testmetoden igen (högerklicka på testklassikonen och välj metoden där). När exekveringen stannar vid brytpunkten är det enkelt att exekvera stegvis med avlusaren för att försöka hitta källan till felet. Alltså: *Testning kan påvisa närvaron fel – och avlusning avslöja* orsaken *till felen.*

JUnit-testning med eclipse

Om du föredrar att använda eclipse finns funktionalitet för testning även i detta verktyg.



- Skapa ett nytt projekt och importera klasserna, inklusive `JunitListTest`.
- Högerklicka på projektnoden och välj
  Properties->Java Build Path->Libraries->Add Library->JUnit->Next->JUnit4->Finish
- Visa testpanelen (fliken) för JUnit: Window->Show View->JUnit

För den intresserade: Till Eclipse finns pluginmodulen EclEmma som genomför *kodtäckningsanalys*. Man kan då enkelt se vilka delar av koden som "motionerats" av testsviten, och vilka delar som inte testats alls. Testad kod markeras med grönt, otestad med rött.

Redovisning

Kopiera utskrifterna från de enkla testen med `SimpleListTest` till en fil med namnet `simpletestresults.txt`. Ladda upp `Lists.java` och `simpletestresults.txt` till Fire. Vid rättningen används samma version av testklassen som finns på kursens hemsida. En nödvändig förutsättning för godkänd laboration är att klassen `Lists` klarar samtliga test.

Extra pyssel

Studera gärna testklassen `JunitListTest` lite närmare. Det är en utmaning att konstruera bra mjukvarutest. De givna testfallen har säkert brister och det vore riskabelt att förlita sig alltför mycket på dem. Försök att formulera flera, bättre och mer heltäckande test! Det är bl.a. viktigt att undersöka alla gränsfall noga, t.ex. att en tom lista beter sig korrekt när det första elementet sätts in. *Definiera eventuella egna test i nya testmetoder – men ändra inte de givna!*

Utskriften från SimpleListTest

```
l1: XabIdRXA7pX
l2: XabIdRXA7pX
l2: SabIdRA7pP
l1: XabIdRXA7pX
true
false
13: SIRAP
l4: PARIS
l3: SIRAP-i-PARIS
l4:
l3: SIRAP-i-PARIS-palindrom
l5: -palindrom
```