Objektinteraktion

Objektorienterad programmering
Laboration 2

Syfte

Att ge konstruera ett litet objektorienterat program med flera samverkande objekt.

Mål

Efter övningen skall du kunna konstruera ett program med flera klasser och där flera objekt samverkar för att lösa ett problem. Du skall förstå hur man kan koppla ihop objekt genom att konstruktorn i en klass sparar objektreferenser till andra objekt i klassens instansvariabler.

Utvecklingsmiljö

BlueJ på valfri plattform.

Litteratur

Kursboken kap 1-2. *Kolla!*

Färdig programkod

Given programkod finns på kursens hemsida i zip-filen
Laborationer->Programkod för labbarna->sticks.zip.

Labbgrupper

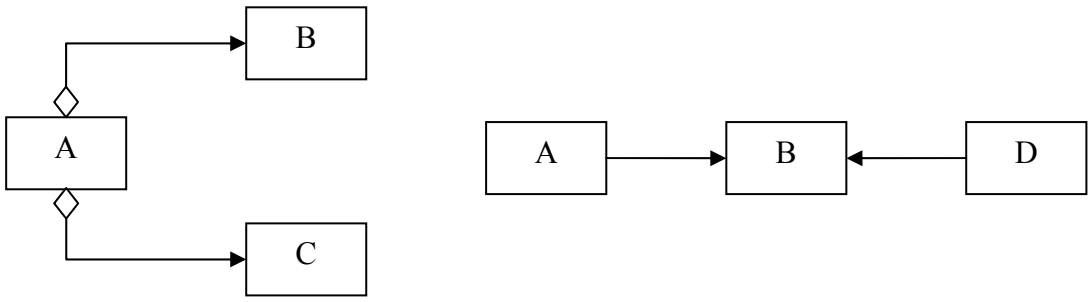
Arbetet genomförs och redovisas i grupper om **två** personer.

Redovisning

Senaste redovisningsdag, se kurs-PM samt Fire. Ladda upp java-filerna med klasserna du utvecklat själv, d.v.s. `Sticks.java`, `ComputerPlayer.java`, `HumanPlayer.java`, samt `GameEngine.java` i Fire. *Glöm inte att trycka på* **SUBMIT***!*

Problembeskrivning

Ett objektorienterat program blir intressant först när flera objekt tillsammans löser ett problem. Man kan då fråga sig hur de olika objekten kan känna till varandras existens. Ett objekt kan i princip använda ett annat objekt på två sätt. Antingen ”har” (äger) objektet det andra objektet för sig själv – och bestämmer helt över dess öde (*aggregationsrelation*), eller så delas det andra objektet mellan flera objekt. I det senare fallet säger man att ett eller flera objekt ”känner till” eller är *associerade* till det gemensamma objektet. Kopplingen mellan två objekt kan vara enkel- eller dubbelriktad. I ett *klassdiagram* i modelleringsspråket UML ritar man en ”har”-relation med en *diamantpil*, och en association med ett streck. En pilspets anger att relationen är enkelriktad. Om pilen är oriktad (ingen pilspets) känner objekten till varandra ömsesidigt. I det vänstra klassdiagrammet nedan kan man utläsa att ett A-objekt har ett B-objekt och ett C-objekt, i det högra delar ett A- och ett D-objekt ett gemensamt B-objekt. A och D kan alltså påverka B oberoende av varandra. B kan vara någon form av gemensam resurs som flera klasser utnyttjar.



Hur implementerar man ovanstående relationstyper i Java. Det finns flera möjligheter, här följer två schematiska exempel. Antag att klasserna A, B, C och D är givna. Vi visar här bara de detaljer som har med relationerna att göra.

Aggregation (”har”)

I klassen A gör vi på följande sätt :

```
class A {
    private B bref;
    private C cref;
...
    public A() {
        bref = new B();
        cref = new C();
    }
...
}
```

Om vi skapar ett A-objekt med **`new`** `A()` så kommer A:s konstruktor att skapa de två B-objekten automatiskt och vi får då en instans av ovanstående klassdiagram.

Association (”känner till”)

I klassen A gör vi på följande sätt :

```
class A {
    private B bref;
...
    public A(B bref) {
        this.bref = bref;
    }
...
}
```

`bref` syftar på *inparametern* bref

`this``.bref` syftar på *instansvariabeln* `bref`

och motsvarande i D

```
class D {
    private B bref;
...
    public D(B bref) {
        this.bref = bref;
    }
...
}
```

Som vi ser ovan kan inga A- eller D-objekt skapas utan att ange referenser till ett B-objekt när objekten skapas.[1] Eftersom A och D beror av B måste vi skapa B-objektet först, och sen A och D i valfri ordning:

```
B b = new B();
A a = new A(b); // koppla ihop a med b
D d = new D(b); // koppla ihop d med b
```

Studera dessa exempel noga och förvissa dig om att du förstått innan du går vidare!

[1] Av klassdiagrammet framgår egentligen inte entydigt om A och D skall dela *samma* B-objekt, eller kunna vara associerade till var sitt B-objekt, och koden ovan tillåter givetvis även att A och D får var sitt B-objekt. Sådan information måste preciseras i en modell med extra kommentarer för att man säkert skall veta hur det är tänkt.

Uppgift

Vi skall göra ett objektorienterat program som spelar tändstickssspelet.

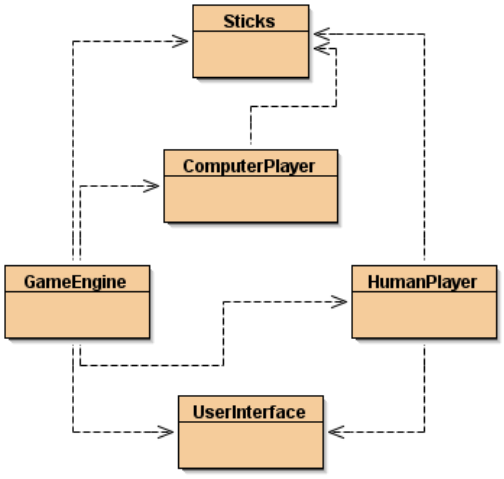
> *Spelet skall spelas av två spelare, datorn och användaren. Man börjar med en hög av 21 stickor. Varje spelare tar i turordning bort en eller två stickor från högen. Den som tvingas ta den sista stickan förlorar. Användaren gör första draget.*

Nedan finns ett exempel som visar hur dialogen med användaren skall se ut.

Följande klasser skall finnas i programmet (de beskrivs mer i detalj senare):

`Sticks`	Håller reda på stickorna.
`ComputerPlayer`	Spelar spelet för datorns räkning.
`HumanPlayer`	Sköter dialogen med användaren. Drar stickor för användarens räkning.
`GameEngine`	Styr spelordningen mellan datorn och användaren.
`UserInterface`	Läser in heltal från tangentbordet. Skriver textmeddelanden på skärmen. *Klassen är färdig.*

I BlueJ kommer projektet att se ut så här




```
Sticks left: 21
How many sticks? 1
Sticks left: 19
How many sticks? 2
Sticks left: 16
How many sticks? 2
Sticks left: 13
How many sticks? 1
Sticks left: 10
How many sticks? 1
Sticks left: 7
How many sticks? 2
Sticks left: 4
How many sticks? 1
Sticks left: 1
How many sticks? 1
Computer wins!
```

BlueJ skiljer ej mellan de två relationstyperna vi pratat om utan ritar allt som associationer. `GameEngine` skall skapa och ha ett objekt av varje av de övriga klasserna. `ComputerPlayer` och `HumanPlayer` skall dessutom associeras till `Sticks`, samt `HumanPlayer` till `UserInterface`. Använd metodiken som beskrivits på föregående sidor.

Allmänna krav på koden

Lösningen skall uppfylla följande krav:

- Inga utskrifter av typ `System.out.println` får göras i de egenutvecklade klasserna. För meddelanden till användaren och inläsning av tal skall `UserInterface` användas.
- Alla instansvariabler skall vara **privata** – *utan undantag*!
- Inga ”slaskvariabler” får deklareras som instansvariabler. De skall vara lokala i resp. metod där de används.

Lämplig arbetsgång

- Börja med att implementera och testa klassen `Sticks`. Skapa ett objekt på ”objektbänken” i BlueJ och kontrollera att objektet fungerar som det skall. Tänk igenom testfallen!
- Fortsätt med `ComputerPlayer` och `HumanPlayer`. Skriv metodstubbar i klasserna, d.v.s. tomma metoder som endast returnerar 0. Detta räcker tills vidare för att kunna utveckla `GameEngine`.
- Implementera därefter `GameEngine`. Deklarera instansvariablerna och skriv konstruktorn. Kompilera och skapa ett objekt. När detta fungerar skriver du `runGame`. Utveckla loopen stegvis. Kompilera och testa ofta!
- När ovanstående kompilerar och exekverar felfritt kan du börja utveckla metoden `move` i `ComputerPlayer` och `HumanPlayer`, alltså en `move`-metod i vardera klassen.
- Om du gjort rätt skall nu spelet fungera felfritt!

Klassbeskrivningar

`public class Sticks`

Håller reda på antalet kvarvarande stickor.

Instansvariabler

`private int maxNoOfSticks;`

Ursprungligt antal stickor.

`private int sticksLeft;`

Kvarvarande antal stickor.

Konstruktorer

`public Sticks(int n)`

Skapar en ”tändstickshög” med `n` stickor genom att initiera instansvariablerna på lämpligt sätt..

`public Sticks()`

Skapar ett tändstickshög med 21 stickor.[2]

Metoder

`public void newGame()`

Initierar/återställer objektet till ursprungligt antal stickor.

`public int sticksLeft()`

Returnerar antalet kvarvarande stickor.

`public int take(int n)`

Om $1 \leq n \leq 2$ minskas antalet stickor med n och det kvarvarande antalet ges som returvärde, om det finns så många, annars returneras -1.

`public class ComputerPlayer`

Spelar spelet för datorns räkning.

Instansvariabler

`private Sticks sticks;`

Referens till objektet med tändstickshögen.

Konstruktorer

`public ComputerPlayer(Sticks sticks)`

Initierar instansvariabeln `sticks` med referensen i inparametern `sticks`.

Metoder

`public int move()`

Genomför datorns drag genom att dra ett lämpligt antal stickor från `sticks`.
För vissa antal av kvarvarande stickor finns en vinnande spelstrategi. Denna skall tillämpas i metoden på ett generellt sätt, lösningen får ej baseras på en uppräkning av specialfall. Metoden skall returnera antalet kvarvarande stickor.

[2] En konstruktor kan anropa en annan med syntaxen **`this`**`(`*parametrar*`);`

`public class HumanPlayer`

Sköter dialogen med användaren samt drar stickor från högen för användarens räkning.

Instansvariabler

`private` `Sticks sticks;`

Referens till objektet med tändstickshögen.

`private` `UserInterface ui;`

Referens till användargränssnittsobjektet.

Konstruktorer

`public` `HumanPlayer(Sticks sticks,UserInterface ui)`

Initierar instansvariablerna med de givna objekten.

Metoder

`public int` `move()`

Frågar användaren efter ett antal stickor att dra. Vid ev. fel skall lämpligt felmeddelande skrivas ut och användaren ges en ny chans. Detta fortgår tills ett korrekt antal angivits. Därefter dras stickorna från `högen`. Metoden skall returnera antalet kvarvarande stickor.

`public class GameEngine`

Styr spelordningen mellan datorn och användaren.

Instansvariabler

```
private Sticks sticks;
private ComputerPlayer computerPlayer;
private HumanPlayer humanPlayer;
private UserInterface ui;
```

Konstruktorer

`public` `GameEngine()`

Skapar objekt av samtliga övriga klasser och sparar referenser till dessa i instansvariablerna. Kopplar ihop `ComputerPlayer`-objektet med `Sticks`-objektet, samt `HumanPlayer`-objektet med `Sticks`-objektet och `UserInterface`-objektet.

Metoder

`public int` `runGame()`

Låter upprepade gånger användaren och datorn göra var sitt drag drag. Användaren skall alltid göra första draget. Om något av dragen resulterar i en tom hög skall vinnaren utropas och spelet avslutas.

`public class UserInterface`

Klassen innehåller metoder för inläsning av heltal från tangentbordet, samt utskrifter av meddelanden i textfönstret. Denna klass är färdig.

Metoder

`public int` `nextInt()`

Läser ett heltal från tangentbordet. Talet returneras. Om felaktiga tecken förekommer i indataraden ignoreras resten av raden och 0 returneras.

forts.

Meddelandemetoder som skriver olika meddelandetexter i textfönstret.

```
public void sticksLeftMsg(int n)
public void howManySticksMsg()
public void illegalMoveMsg()
public void computerWinMsg()
public void playerWinMsg()
```

Lycka Till!