Objektorienterad modellering och design

Mastermind

Som ett exempel på objektorienterad programutformning skall vi implementera det sällskapsspel som kallas Mastermind. Tillverkarens regler följer.

> MASTERMIND is a game which gives each player a chance to outsmart his opponent. The Codemaker secretly sets up a line of Code Pegs behind his shield and the Codebreaker has up to ten opportunities to try and duplicate the colour and each exact position of the hidden Code Pegs without ever seeing them.
>
> The Codemaker secretly puts 4 Code Pegs in 4 holes behind the shield. Use any combination of the six colours. You may use 2 or more Code Pegs of the same colour if you wish.
>
> The Codebreaker will try to duplicate the exact colours and positions of the code hidden behind the shield. Each time the Codebreaker places a row of Code Pegs (they are then left in position troughout the game), the Codemaker must give him the following information by placing the black and white Key Pegs in the Key Peg holes alongside the Code Pegs placed by the Codebreaker, or leaving holes vacant.
>
> **Black Key Pegs** are placed by the Codemaker in any of the Key Peg holes for every Code Peg placed by the Codebreaker which is in the same colour and in exactly the same position as one of the Code Pegs behind the shield.
>
> **White Key Pegs** are placed by the Codemaker in any one of the Key Peg holes when any hidden Code Peg behind the shield matches the Codebreaker´s Code Pegs in colour only, but not in position.
>
> **Example:** If one red Code Peg is behind the shield and the Codebreaker places 2 red Code Pegs in the wrong position ONE white Key Peg is used.
>
> If the Codebreaker duplicates the hidden code behind the Codemaker's shield, the Codemaker places 4 black Key Pegs and reveals the hidden code. The games is over.
>
> *Invicta Plastic Ltd, Leicester, England.*

Även om problemet är litet och går att lösa med ett par sidor programkod skall vi göra en objektorienterad modell med ett tiotal klasser. Fördelarna med detta blir tydligare om problemet är större, men principerna blir tydligare i det lilla formatet. Ett större problem skulle ta mycket mer plats och tid utan att ge mycket mer.

Objektorientering innebär att vi skall identifiera de objekt som finns i den verklighet vi skall simulera och bland dem välja ut dem som skall beskrivas med klasser i programmet. Presumptiva klasser finner man ofta som substantiv i beskrivningen av verkligheten. I reglerna finner vi följande substantiv: Mastermind, game, player, codemaker, opponent, line, code peg, opportunity, colour, position, hole, shield, combination, codebreaker, time, row, information och key peg.

Några av dessa substantiv är objekt som ej skall modelleras av detta program. Detta gäller den *player* som är *opponent* och *codebreaker*, som skall finnas utanför programmet. Några substantiv representeras lämpligen med språkets primitiva typer. I vårt exempel är *colour* och *position* sådana substantiv. Substantiven *line* och *row*, som i texten används som synonymer och skulle

kunna reprersenteras med vektorer (array). *Hole* svarar mot ett index i en vektor. *Code peg* och *Key peg* svarar mot element i dessa vektorer vars värden är av typen *colour*. Substantivet *shield* har inget inre tillstånd och används i beskrivningen bara för att avskilja kodkonstruerarens pegs från kodknäckarens. Substantiven *combination*, *time* och *information* kräver inte några egen representation.

Programmet är styrt av kommandon. Det går utmärkt att exekvera ett kommando direkt sedan man tolkat det. Vi kommer i stället att använda *Command*–mönstret av pedagogiska skäl; vi vill visa hur det fungerar i ett litet problem även om det design–mässigt är omotiverat.

Vi kommer att implementera kommandon för att starta ett nytt parti, göra en gissning, inspektera alla tidigare gissningar och resultat, att inspektera den hemliga koden och att avsluta hela spelsessionen.

`Mastermind` är ett lämpligt namn på den klass som representerar hela spelet. Spelets tillstånd utöres av den hemliga koden och en lista av gissningar. Den klass `Code` som skall representera koden och en gissning är central och kommer att innehålla metoder för att undersöka om två koder är lika och eller hur olika de är.

Game får vare en klass som modellerar ett parti. Kodkonstrueraren, codemaker, kommer vi att simulera med en slumptalsgenerator, som vi hämtar ur `java.util.Random`.

Användarkommunikationen är primitiv och beskriv i en egen klass som också hinnehåller det minimala huvudprogrammet.

```
 1  public class UI {
 2      private MasterMind masterMind = new MasterMind();
 3      private BufferedReader input = new BufferedReader(new InputStreamReader(
 4              System.in));
 5      private CommandParser parser = new CommandParser();
 6      public void play() throws IOException {
 7          while (true) {
 8              System.out.print("> ");
 9              Command command = parser.build(input.readLine());
10              command.execute(masterMind);
11          }
12      }
13      public static void main(String[] argv) throws IOException {
14          new UI().play();
15      }
16  }
```

Den metod som analyserar ett kommando finns i `CommandParser` och tittar bara på det första tecknet.

```
 1  public class CommandParser {
 2      public Command build(String string) {
 3          switch (string.charAt(0)) {
 4          case 'q':
 5              return new Quit();
 6          case 'h':
 7              return new History();
 8          case 'n':
 9              return new NewGame();
10          case 's':
```

```
11              return new Solution();
12          default:
13              return new Guess(string);
14          }
15      }
16  }
```

Metoden bygger kommondoobjekt som implementerar gränssnittet `Command`. Några kommandon är utelämnade.

```
 1  public interface Command {
 2      public void execute(MasterMind masterMind);
 3  }
 4
 5  class Guess implements Command {
 6      private Code guess;
 7      Guess(String s) {
 8          guess = new Code(s);
 9      }
10      public void execute(MasterMind masterMind) {
11          masterMind.add(guess);
12          System.out.println(masterMind.response(guess));
13      }
14  }
15
16  class Solution implements Command {
17      public void execute(MasterMind masterMind) {
18          System.out.println(masterMind.solution());
19      }
20  }
21
22  class History implements Command {
23      public void execute(MasterMind masterMind) {
24          System.out.println(masterMind.guessList());
25      }
26  }
27
28  class NewGame implements Command {
29      public void execute(MasterMind masterMind) {
30          masterMind.reset();
31      }
32  }
33
34  class Quit implements Command {
35      public void execute(MasterMind masterMind) {
36          masterMind.quit();
37      }
38  }
```

Klassen `Code` tillhandahåller två konstruerare, `Code()` som ger en slumpmässigt vald kod och `Code(String s)` som konstruerar koden från en given sträng. Vi väljer att representera koden med en vektor med fyra heltal. Det kan vara lämpligt att låta tale 1 till 6 svara mot de sex färgerna. Implementeringen av de metoder som avlsutas med semikolon är utelämnade.

```
 1  public class Code {
 2      public static final int SIZE = 4;
 3      public static final int COLORS = 6;
 4      private static Random random = new Random();
 5      private int[] code = new int[SIZE];
 6      public Code() {
 7          for (int i = 0; i < SIZE; i++)
 8              code[i] = random.nextInt(COLORS);
 9      }
10      public Code(String s) {
11          for (int i = 0; i < s.length(); i++) {
12              code[i] = s.charAt(i) - '1';
13          }
14      }
15      public String toString() {
16          StringBuffer buffer = new StringBuffer();
17          for (int i = 0; i < code.length; i++) {
18              buffer.append(code[i] + 1);
19          }
20          return buffer.toString();
21      }
22      public int getColor(int index) {
23          return code[index];
24      }
25      public boolean equals(Object object) {
26          Code other = (Code) object;
27          for (int i = 0; i < SIZE; i++)
28              if (code[i] != other.code[i])
29                  return false;
30          return true;
31      }
32      private int matches(Code other) {
33          int m = 0;
34          for (int i = 0; i < SIZE; i++)
35              if (code[i] == other.code[i])
36                  m++;
37          return m;
38      }
39      private int occs(Code other) {
40          int[] tmp = new int[SIZE];
41          for (int i = 0; i < SIZE; i++)
42              tmp[i] = other.code[i];
43          int m = 0;
44          for (int i = 0; i < SIZE; i++)
45              for (int j = 0; j < SIZE; j++)
46                  if (code[i] == tmp[j]) {
47                      m++;
48                      tmp[j] = -1;
49                      break;
50                  }
51          return m;
```

```
52      }
53      public Response response(Code other) {
54          int m = matches(other);
55          int o = occs(other);
56          return new Response(m, o - m);
57      }
58  }
```

Listan av gissningar representeras med klassen `CodeList`.

```
 1  public class CodeList {
 2      private List<Code> list = new ArrayList<Code>();
 3      public String toString(Code solution) {
 4          StringBuffer buffer = new StringBuffer();
 5          for (Iterator<Code> iter = list.iterator(); iter.hasNext();) {
 6              Code guess = iter.next();
 7              buffer.append(guess).append(' ').append(guess.response(solution))
 8                      .append('\n');
 9          }
10          return buffer.toString();
11      }
12      public void add(Code pattern) {
13          list.add(pattern);
14          setChanged();
15          notifyObservers(pattern);
16      }
17      public int size() {
18          return list.size();
19      }
20      public void clear() {
21          list.clear();
22      }
23      public Code get(int index) {
24          return list.get(index);
25      }
26  }
```

När man jämför en gissning med den hemliga koden får man veta hur många vita resp. svarta pegs som skall visas. För att kunna returnera detta talpar behövs en primitiv klass.

```
 1  public class Response {
 2      private int white, black;
 3      public Response(int white, int black) {
 4          this.white = white;
 5          this.black = black;
 6      }
 7      public String toString() {
 8          StringBuffer buffer = new StringBuffer();
 9          for (int i = 0; i < white; i++)
10              buffer.append('+');
11          for (int i = 0; i < black; i++)
12              buffer.append('-');
13          return buffer.toString();
```

```
14      }
15      public int white() {
16          return white;
17      }
18      public int black() {
19          return black;
20      }
21  }
```

Den klass som representerar hela spelet har inte mycket intelligens.

```
 1  public class MasterMind {
 2      private CodeList guessList = new CodeList();
 3      private Code solution = new Code();
 4      public void add(Code guess) {
 5          guessList.add(guess);
 6      }
 7      public int guessCount() {
 8          return guessList.size();
 9      }
10      public String guessList() {
11          return guessList.toString(solution);
12      }
13      public void quit() {
14          System.exit(0);
15      }
16      public void reset() {
17          guessList.clear();
18          solution = new Code();
19      }
20      public Response response(Code guess) {
21          return guess.response(solution);
22      }
23      public Code solution() {
24          return solution;
25      }
26  }
```