INFO0062 - Object-Oriented Programming

Presentation of the project

Jean-François Grailet

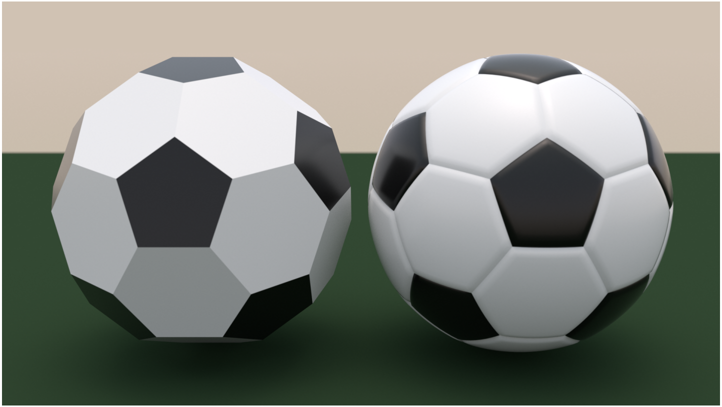
University of Liège

Faculty of Applied Sciences

Academic Year 2018 - 2019

Project

Assembly of a soccer ball



Statement

- You are asked to solve a (3D) puzzle with OOP in Java.
- This puzzle consists in assembling a soccer ball as a *truncated icosahedron*.[1]
    - The puzzle is made of 32 pieces (20 hexagons and 12 pentagons).
    - Each piece has several concave and convex sides.
    - In a valid assembly, every concave side is matched with a convex side.
    - There are 10 types of hexagons and 4 types of pentagons.
- You only have to describe how pieces are assembled in text format.
- To know *how* to do it, let's first see a way to model the problem.

[1] FR: icosaèdre tronqué, cf. `https://en.wikipedia.org/wiki/Truncated_icosahedron`

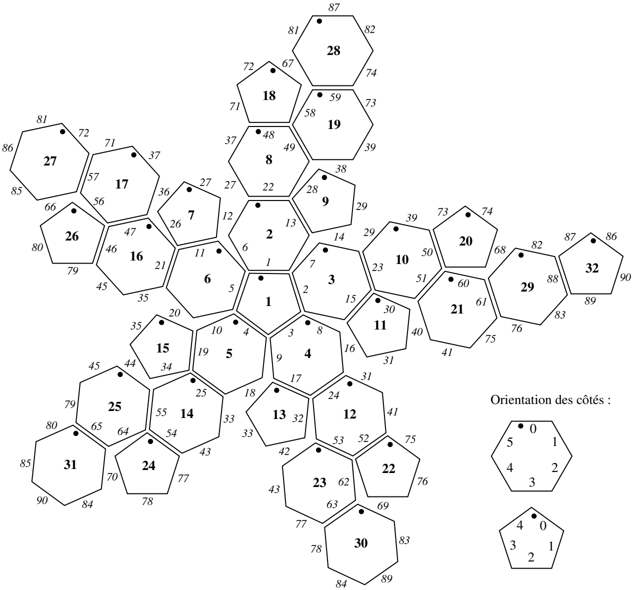
Your tools

- To get started with the project, you can download `project_basis.zip`.[2]
- This archive provides three files.
  - `soccer_ball_net.pdf`: annotated net[3] of the truncated icosahedron
  - `soccer_ball_pieces.pdf`: the (numbered) puzzle pieces
  - `Data.java`: a little class providing constants to handle the net and the pieces

[2] See `http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062.php`

[3] FR: le patron de conception

Your tools (II)



Orientation des côtés :

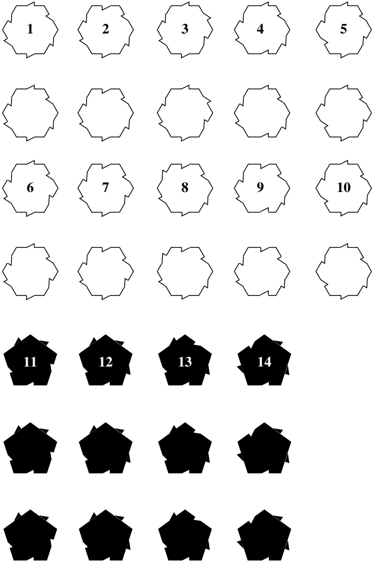
Your tools (III)

- Each facet in the net is annotated as follows.
  - Center numbers is a position, i.e., a unique integer to denote the facet.
  - Sides are annotated with an integer denoting adjacency with other facets.
  - The black circle denotes the initial orientation of the facet.
- This is translated in `Data.java` as follows.
  - `CONNECTIONS` is a 2D array with 32 lines (one per position).
  - Each line references a linear array with 5 or 6 integers.
  - Integers in each line gives the connection of the facet with others.

Your tools (IV)

```
final static int[][] CONNECTIONS = {
{  1,  2,  3,  4,  5 }, // Position 1 (fit for a pentagon)
{ 22, 13,  7,  1,  6, 12 }, // Position 2 (fit for an hexagon)
{ 14, 23, 15,  8,  2,  7 }, // Position 3 (fit for an hexagon)
{  8, 16, 24, 17,  9,  3 }, // Etc.
{  4,  9, 18, 25, 19, 10 },
{  6,  5, 10, 20, 21, 11 },
{ 27, 12, 11, 26, 36 },
/* ... (26 other positions) */ };
```

Your tools (V)



Your tools (VI)

- Each puzzle piece (or element)
  - has a unique number to denote the type,
  - has a unique spread of concave/convex sides,
  - comes in several instances.
- This is translated in `Data.java` as follows.
  - `ELEMENTS_SIDES` is a 2D array with 14 lines (one per piece type).
  - Each line references a linear array with 5 or 6 integers.
  - These integers are 1 (convex side) and -1 (concave side).
  - `NB_ELEMENTS` is a linear array with 14 elements.
  - Each cell gives the amount of instances of a given piece type.

Your tools (VII)

```
final static int[][] ELEMENTS_SIDES = {
{  1, -1,  1,  1,  1, -1 }, // Hexagon (type 1)
{ -1, -1,  1,  1,  1, -1 }, // Hexagon (type 2)
{  1,  1, -1,  1,  1, -1 }, // Hexagon (type 3)
/* ... (7 other hexagons) */
{  1, -1, -1, -1,  1 }, // Pentagon (type 11)
{  1,  1, -1, -1,  1 }, // Pentagon (type 12)
/* ... (2 other pentagons) */ };

final static int[] NB_ELEMENTS =
{ 2, 2, 2, 2, /* ... */ , 3, 3, 3, 3 };
```

Your task

- Using OOP, find how to fit the pieces on the net to get a valid assembly.
- An assembly is valid when each concave side is matched with a convex side.
- After solving the problem, you just have to describe it in text format.
- We suggest you the following display policy: display 32 lines giving
  - a position,
  - a type of piece (that fits the position),
  - the orientation of the piece (i.e., amount of rotations).

Your task (II)

Example of a valid solution

```
Position 1 - Element 11 - Orientation 0
Position 2 - Element 1 - Orientation 2
Position 3 - Element 1 - Orientation 0
Position 4 - Element 2 - Orientation 1
Position 5 - Element 2 - Orientation 2
Position 6 - Element 7 - Orientation 0
Position 7 - Element 11 - Orientation 3
Position 8 - Element 3 - Orientation 1
Position 9 - Element 13 - Orientation 2
Position 10 - Element 3 - Orientation 0
Position 11 - Element 11 - Orientation 3
Position 12 - Element 4 - Orientation 1
// ... (20 other positions)
```

Your task (III)

- Note that you can do the project without our suggestions.
  - You could build and use your own `Data.java`, for instance.
  - Or use your own display policy.
- However, in that case, we ask you to describe your data and/or display policy.
  - See submission guidelines.

Submission guidelines

- Your project must compile and compute a valid assembly as asked.
  - To ensure it will compile fine, you can use *Network 8* computers. [4]
- You project must solve the puzzle with an object-oriented approach.
- The `main()` method of your project must be located in a `SoccerBall` class.
- You can do the project on your own or with a classmate.
- You are free to add any additional feature if you want to.
  - However, no extra point will be given for something not asked by the statement.

[4] Cf. additional slides on `http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062.php`

Submission guidelines (II)

- You will submit your project as a ZIP (`.zip`) archive.
  - It should contain only `.java` files (no subfolder).
  - Subfolders will be tolerated if and only if you use packages.
- You can add a `README` file (`.txt` or PDF) **if relevant**.
  - E.g. if you use a unique display policy.
  - E.g. if some additional feature requires commentary.
- **You don’t have to submit a report.**

Submission guidelines (III)

- Submit your archive to `oop@montefiore.ulg.ac.be`.
- The deadline is April 22 (included).
- If you do the project by yourself
  - Archive name: `oop_lastname_firstname.zip`
  - E-mail subject: `OOP - 2019 - lastname firstname`
  - Prefer setting only the first letter of your names in uppercase[5].
- If you do the project with a classmate
  - Archive name: `oop_lastname1_lastname2.zip`
  - E-mail subject: `OOP - 2019 - lastname1 lastname2`
  - Prefer providing your last names in alphabetical order.

[5] FR: en majuscule

Tips & tricks

General advice

- Always keep in mind that this is an **object-oriented** programming project.
- You are therefore expected to
  - model entities of the problem as objects (e.g., each facet/piece is an object),
  - give them relevant responsibilities,
  - put to practice OOP concepts (e.g. encapsulation, communication, etc.).
- Functionality of the project only count for a small part of the final grade.

How to solve the puzzle

- This problem is in fact similar to the eight queens puzzle.
  - I.e., you have to assemble pieces such that they are compatible with others.
  - Cf. chapter 4 of the theoretical course.
- Of course, this project is more complex to program because
  - there are 32 pieces in the assembly (versus 8 queens),
  - there are 14 different types of pieces (versus only queens).
- You can still use a very similar methodology to solve the puzzle.
  - Of course, you are free to try other approaches as long as they work.

How to solve the puzzle (II)

- **Finding a solution for a given position**
  - Pick a piece.
  - Rotate it until it fits or until a full rotation is completed.
  - If it fits, it sits.
  - Otherwise, try another type of piece.
- **Finding a solution when no piece can fit a given position**
  - Ask previous position to rotate its piece once.
  - Ask previous position to find a solution (again).
  - Try again to find a solution on the current position.
- It's up to you to find how to handle the basic cases.

Challenges

- Before using this algorithm, you have several problems to solve.
  - How do you model the polyhedron net ?
  - How do you check the compatibility of a piece with the assembly ?
  - How do you ensure your solution never tries the same piece twice ?
  - How are pieces provided at first ?

Challenges (II)

- Here are a few practical questions to guide you.
  - Should you use different classes to model a facet and a piece ?
  - How about facet/piece objects keeping references to their neighbors ?
  - How about a FIFO data structure to store your puzzle pieces ?
  - Or instead, how about sorted lists of puzzle pieces ?
  - Is inheritance relevant in this context ?

Using the Java library

- You can use the Java library to ease your task.
- In particular, classes to handle collections of objects will be very useful.
- For instance, you can have a look at
  - `java.util.ArrayList`
  - `java.util.LinkedList`
  - `java.util.Vector`
  - `java.util.Set`

Using the Java library (II)

- Of course, you can still create your own data structures.
- Note also that if you want to sort collections, you can use
  - `sort()` from `java.util.Collections`,
  - the `Comparable` interface and its methods.

Some final pieces of advice

- Test carefully your solution while programming it.
  - E.g., first test your project for 5 pieces.
  - If you get a problem at this point, then it will be worse for 32 pieces.
- If your solution is too complex, you’re probably doing it wrong.
  - *Complex* here means you have too much code and/or classes.
  - Don’t forget the point of OOP is to make your life simpler for these kinds of problems.

Coding style and documentation

About coding style

- Use meaningful variable, method and class names.

- For instance, compare the readability of the two following methods:

```
public static int a(int b) {
  if (b <= 0)
    return 1;

  return b * a(b - 1);
}
```

```
public static int factorial(int input) {
  if (input <= 0)
    return 1;

  return input * factorial(input - 1);
}
```

About coding style (II)

- Convention for variable/method names is to use lowercase[6] words.
- Starting from the second word, the first letter is uppercase[7].
  - E.g. `priceWithTaxes`.
- Alternatively, you can use lowercase words separated by “_” (underscore).
  - E.g. `price_with_taxes`.
- For constants, the convention is to use uppercase words separated by “_”.
  - E.g. `TVA_IN_BELGIUM`.
- For classes and interfaces, lowercase words that begin with an uppercase letter.
  - E.g. `TaxesCalculator`.

[6] FR: en lettre minuscule

[7] FR: en lettre majuscule

About coding style (III)

- Two conventions for curly braces related to blocks (choose one):

```
while (true) {

}
```

```
while (true)
{

}
```

- Indentation must be coherent and strongly respected:

```
public class MyClass {
      public static void m1() {
      instruction1;
    instruction2;
    }

public static void m2() {
 instruction1;
     instruction2;
}
    }
```

```
public class MyClass {
  public static void m1() {
    instruction1;
    instruction2;
  }

  public static void m2() {
    instruction1;
    instruction2;
  }
}
```

About coding style (IV)

- You can insert spaces or empty lines in your code to improve readability.

```
public class Probability{
  public static double arrange(int n,int k){
    return (double)factorial(n)/factorial(n-k);
  }
  public static int factorial(int input){
    if(input<=0)return 1; return input*factorial(input-1);
  }
}
```

```
public class Probability {
  public static double arrange(int n, int k) {
    return (double) factorial(n) / factorial(n - k);
  }

  public static int factorial(int input) {
    if (input <= 0)
      return 1;
    return input * factorial(input - 1);
  }
}
```

About coding style (V)

- Choose a maximal number of characters per line of code.
- Common convention: 80 columns rule.
- But you can also use 100 columns if you prefer.
- **The most important is to make consistent choices and to respect them.**

Documentation

- You can document your code using comments.
- It is useful to remember what you did, but also to inform other programmers.
- Typically, you should at least describe the role of a class.

```
/*
 * This class offers a set of static methods to perform various
 * calculations relative to the probability theory.
 */

public class Probability {
   ...
}
```

Documentation (II)

- You can describe the purpose of a method by detailing
  - its parameter(s) (if any) and returned value (if any),
  - the instantiation context of its exception(s) (if any).
- You can go as far as using Javadoc (optional).

```
/*
 * This method tests whether the input parameter is odd and
 * returns a boolean to confirm it. In the case where the input
 * parameter is negative, a MyException exception is thrown.
 */

public static boolean isOdd(int input) throws MyException {
  if (input < 0)
    throw new MyException();
  return (input % 2) == 1;
}
```

About language(s)

- You can choose English or French for your documentation.
- Prefer English for the names of variables, methods and classes.
- However, once you chose a language, **stick with it**.

```
/**
 * Cette méthode teste si un entier positif est impair.
 *
 * @param  input       L’entier à tester.
 * @return boolean     Vrai si l’entier est impair, faux sinon.
 * @throws MyException Lancée quand un entier négatif est donné.
 */

public static boolean isOdd(int input) throws MyException {
  if (input < 0)
    throw new MyException();
  return (input % 2) == 1;
}
```