OGP Assignment 2013-2014: **Worms** (Part I)

This text describes the first part of the assignment for the course *Object-oriented Programming*. There is no exam for this course, so all grades are scored on the assignment. The assignment is preferably made in groups consisting of two students; only in exceptional situations the assignment can be made individually. Each team must send an email containing the names and the course of studies of all team members to ogp-project@cs.kuleuven.be before the 1st of March. If you cooperate, only one member of the team should send an email putting the other member in CC.

If during the semester conflicts arise within a group, this should be reported to ogp-project@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

The assignment consists of three parts. The first part focusses on a single class, the second on associations between classes, and the third on inheritance and generics. After handing in the third part, the entire solution must be defended before Professor Steegmans.

A number of teaching assistants (TAs) will advise the students and answer their questions. More specifically, each team has a number of hours where the members can ask questions to a TA. The TA plays the role of consultant who can be hired for a limited time. In particular, students may ask the TA to clarify the assignment or the course material, and discuss alternative designs and solutions. However, the TA will not work on the assignment itself. Consultations will generally be held in English. Thus, your project documentation, specifications, and identifiers in the source code should be written in English. Teams may arrange consultation sessions by email to ogp-project@cs.kuleuven.be. Please outline your questions and propose a few possible time slots when signing up for a consultation appointment. To keep track of your development process, and mainly for your own convenience, we encourage you to use a source code management and revision control system such as *Subversion* or *Git*.

During the assignment, we will create a simple game that is loosely based on the artillery strategy game *Worms*. Note that several aspects of the assignment will not correspond to the original game. Your solution should be implemented in Java 6 or higher and follow the rules described in this document.

The goal of this assignment is to test your understanding of the concepts introduced in this course. For that reason, we provide a graphical user interface and it is up to the teams to implement the requested functionality. This functionality is described at a high level in this document and the student may design and implement one or more classes that provide this functionality, according to his best judgement. The grades for this assignment do not depend only on functional requirements. We will also pay attention to documentation, accurate specifications, re-usability and adaptability.

1 Assignment

Worms is a turn-based artillery strategy game in which the player controls a team of worms that can move in a two-dimensional landscape. The worms are equipped with tools and weapons that are to be used to achieve the goal of the game: kill the worms of other teams and have the last surviving worms. In this assignment, we will create a game loosely based on the original artillery strategy released in 1995 by Team17 Digital.

In the first part of the assignment, we focus on a single class `Worm`. However, your solution may contain additional helper classes (in particular classes marked *@Value*). In the second and third part, we will add additional classes to our game. In the remainder of this section, we describe the class `Worm` in more detail. All aspects of your implementation must be specified both formally and informally.

1.1 Position, Orientation, Radius, Mass and Action Points

Each worm is located at a certain position (x, y) in a two-dimensional space. Both x and y are expressed in metres (m). All aspects related to the position of a worm shall be worked out defensively.

Each worm faces a certain direction expressed as an angle θ in radians. For example, the angle of a worm facing right is 0, a worm facing up is at angle $\pi/2$, a worm facing left is at angle π and a worm facing down is at angle $3\pi/2$. All aspects related to the direction must be worked out nominally.

The shape of a worm is a circle with finite radius σ (expressed in metres) centred on the worm's position. The radius of a worm must at all times be at least 0.25 m. Yet, the effective radius of a worm may change during the program's execution. In the future, the lower bound on the radius may change and it is possible that different lower bounds will then apply to different worms. Each worm also has a mass m expressed in kilograms (kg). m is derived from σ, assuming that the worm has a spherical body and a homogeneous density p of 1062 kg/m^3: $m = p \cdot (4/3 \cdot \pi\sigma^3)$. All aspects related to a worm's radius and mass must be worked out defensively.

Each worm has a maximum number of action points, and a current number of action points, which shall be represented by integer values. The maximum number of action points of a worm must be equal to the worm's mass m, rounded to the nearest integer. If the mass of a worm changes, the maxima must be adjusted accordingly. As explained in Section 1.2, the current number of action points may change during the program's execution. Yet, the current value of a worm's action points must always be less than or equal to the maximum value, but it must never be less that zero. All aspects related to action points must be worked out in a total manner.

If not stated otherwise, all numeric characteristics of a worm shall be treated as double precision floating-point numbers. That is, use Java's primitive type `double` to store the radius, the x-coordinate, etc. The characteristics of a worm must be valid numbers (meaning that `Double.isNaN` returns `false`) at all times. However, we do not explicitly exclude the values `Double.NEGATIVE_INFINITY` and `Double.POSITIVE_INFINITY` (unless specified otherwise).

In addition to the above characteristics, each worm shall have a name. A worm's name may change during the program's execution. Each name is at least two characters long and must start with an uppercase letter. In the current version, names can only use letters (both uppercase and lowercase), quotes (both single and double) and spaces. James o'Hara is an example of a well-formed name. It is possible that other characters may be allowed in later versions of the game. All aspects related to the worm's name must be worked out defensively.

The class `Worm` shall provide methods to inspect name, position, direction, radius, mass, and action points of a worm.

1.2 Turning and Moving

A worm can move and turn. The class `Worm` shall provide a method `move` to change the position of the worm based on the current position, orientation, and a number of steps. The given number of steps shall never be less than zero. As this method affects the position of the worm, it must be worked out defensively.

The class `Worm` must provide a method `turn` to change the orientation of the worm by adding a given angle to the current orientation. This method must be worked out nominally.

Active turning and moving costs action points. Changing the orientation of a worm by 2π shall decrease the current number of action points by 60. Respectively, changing the orientation of a worm by a fraction of $2\pi/f$ must imply a cost of $60/f$ action points. Movement always occurs in steps. The distance covered in one step shall be equal to the radius of the worm. The cost of movement shall be proportional to the horizontal and vertical component of the step such that a horizontal step is at the expense of 1 action point, while a vertical step incurs costs of 4 action points. The total cost of a step in the current direction can be computed as $|cos\,\theta| + |4sin\,\theta|$. Since action points are to be handled as integer values, all expenses of action points shall be rounded up to the next integer.

In the future, worms may also move passively, e.g. fall down a chasm or get blasted away. Passive movement may not incur a decrease of the worm's action points.

1.3 Jumping

Worms may also jump along ballistic trajectories. The class `Worm` shall provide a method `jump` to change the position of the worm as the result of a jump from the current position (x, y) and with respect to the worm's orientation θ and the number of remaining action points APs. As this method affects the position of the worm, it must be worked out defensively.

Given the remaining activity points APs and the mass m of a worm, the worm will jump off by exerting a force of $F = (5 \cdot APs) + (m \cdot g)$ for 0.5 s on its body. Here, g represents the Earth's standard acceleration of 9.80665 m/s^2. We can compute the initial velocity of the worm as $v_0 = (F/m) \cdot 0.5\ s$. With this, the worm will jump a distance $d = ({v_0}^2 \cdot sin(2\theta))/g$ horizontally, within the following $t = d/(v_0 \cdot cos\,\theta)$ seconds.

As illustrated in Fig. 1.3, jumping worms always travel horizontally as if launched from a solid ground line parallel to the x-axis at the worm's y position, and return to that line. However, if the worms orientation is in the range $\pi < \theta < 2\pi$, i.e. the worm is facing downwards, the worm shall not move. Jumping consumes all remaining action points of a worm.

The class `Worm` shall also provide a method `jumpTime` that returns the above t for a potential jump from the current position, and a method `jumpStep` that computes in-flight positions $(x_{\Delta t}, y_{\Delta t})$ of a jumping worm at any Δt sec-

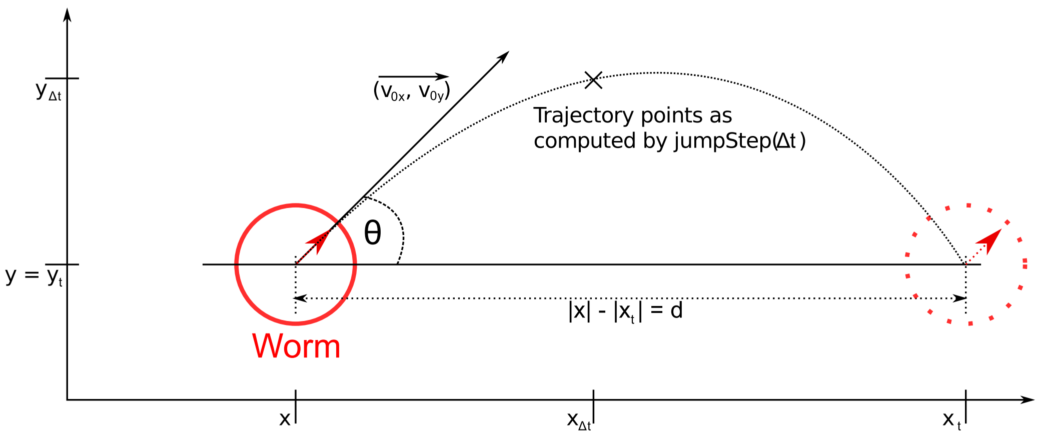

Figure 1: Illustration of a jumping worm's trajectory.

onds after launch. $(x_{\Delta t}, y_{\Delta t})$ may be computed as follows:

$$\begin{aligned} v_{0x} &= v_0 \cdot cos\,\theta \\ v_{0y} &= v_0 \cdot sin\,\theta \\ x_{\Delta t} &= x + (v_{0x}\Delta t) \\ y_{\Delta t} &= y + (v_{0y}\Delta t - \tfrac{1}{2} g \Delta t^2) \end{aligned}$$

The methods `jumpTime` and `jumpStep` must not change any attributes of a worm. The above equations represent a simplified model of terrestrial physics and consider uniform gravity with neither drag nor wind. Future phases of the assignment may involve further trajectory parameters or geographical features of game world.

2 Reasoning about Floating-point Numbers

Floating-point computations are not exact. This means that the result of such a computation can differ from the one you would mathematically expect. For example, consider the following code snippet:

```
double x = 0.1;
double result = x + x + x;
System.out.println(result == 0.3);
```

The last statement outputs `false`, even though $0.1 + 0.1 + 0.1$ is mathematically equal to 0.3. The output is `false` because the variable `result` holds the value 0.30000000000000004.

A Java `double` consists of 64 bits. Clearly, it is impossible to represent all possible real numbers using only a finite amount of memory. For example, $\sqrt{2}$ cannot be represented exactly and Java represents this number by an approximation. Because numbers cannot be represented exactly, floating point algorithms make rounding errors. Because of these rounding errors, the expected outcome of an algorithm can differ from the actual outcome.

For the reasons described above, it is generally bad practice to compare the outcome of a floating-point algorithm with the value that is mathematically expected. Instead, one should test whether the actual outcome differs at most ϵ from the expected outcome, for some small value of ϵ. The class `Util` (included in the assignment) provides methods for comparing doubles up to a fixed ϵ.

The course *Numerieke Wiskunde* discusses the issues regarding floating-point algorithms in more detail. For more information on floating point numbers, we suggest that you follow the tutorial at
http://introcs.cs.princeton.edu/java/91float/.

3 Testing

Write JUnit test suite for the class `Worm` that tests each public method. Include this test suite in your submission.

4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on worms. The user interface is included in the assignment.

To connect your implementation to the GUI, write a class `Facade` that implements `IFacade`. `IFacade.java` contains additional instructions on how to implement the required methods. To start the program, run the `main` method in the class `Worms`. After starting the program, you can press keys to modify the state of the program. The command keys are `Tab` for switching worms, `left` and `right` arrow key (followed by pressing `return`) to turn, `up` to move forward, `+` and `-` to increase and decrease the worm's radius, `n` to change the worm's name, `j` to jump, and `Esc` to terminate the program. Be aware that the GUI displays only part of the (infinite) space. Your space worms may leave and return to the visible area.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Worm`. No additional grades will be awarded for changing the GUI.

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `IFacade`. As described in the documentation of `IFacade`, the methods of your `IFacade` implementation shall only throw `ModelException`. An incomplete test class is included in the assignment to show you what our test cases look like.

5 Submitting

The solution must be submitted via Toledo as a jar file individually by all team members before the 16th of March 2014 at 11:59 PM. You can generate a jar file on the command line or using eclipse (via `export`). Include all source files (including tests) and the generated class files. Include your name, your course of studies and a link to your code repository in the comments of your solution. When submitting via Toledo, make sure to press `OK` until your solution is submitted!

6 Feedback

A TA will give feedback on the first part of your project. These feedback sessions will take place between the 17th and the 31st of March. More information will be provided via Toledo.