OGP Assignment 2012-2013: Asteroids (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 should send an email containing the names and the course of studies of all team members to ogp-project@cs.kuleuven.be before the 4th 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 (TA) 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/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 shall be written in English. 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 loosely based on the arcade game *Asteroids*. 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. The requested functionality is described at a high level in this document and it is up to the student to design and implement one or more classes that provide this functionality. 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

Asteroids is an arcade game where the player controls a space craft in an asteroid field. The goal of the game is to evade and destroy objects such as enemy vessels and asteroids in a two-dimensional, rectangular space. In this assignment, we will create a game loosely based on the original arcade game released in 1979 by Atari.

In the first part of the assignment, we focus on a single class `Ship`. 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 `Ship` in more detail. All aspects of your implementation shall be specified both formally and informally.

1.1 Position, Velocity, Orientation and Radius

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

Each spaceship has velocities v_x and v_y that determine the vessel's movement per time unit in the x and y direction, respectively. Both v_x and v_y are expressed in kilometres per second (km/s). The speed of a ship, computed as $\sqrt{v_x^2 + v_y^2}$, shall never exceed the speed of light c, $300000 km/s$. In the future, this limit need not remain the same for each ship, but it will always be less than or equal to c. All aspects related to velocity must be worked out in a total manner.

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

The shape of each ship is a circle with finite radius σ (expressed in kilometres) centred on the ship's position. The radius of a ship must be larger than 10 and never changes during the program's execution. In the future, this lower bound may change, however it will always remain the same for each ship and its value will be at least zero. All aspects related to the radius must be worked out defensively.

All characteristics of a ship must 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 ship 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).

The class `Ship` shall provide methods to inspect the position, velocity, direction and radius.

1.2 Moving, Turning and Accelerating

A spaceship can move, turn and accelerate. The class `Ship` shall provide a method `move` to change the position of the space craft based on the current position, velocity and a given time duration Δt. The given duration Δt shall never be less than zero. As this method affects the position of the ship, it must be worked out defensively.

The class `Ship` must provide a method to turn the ship by adding a given angle to the current direction. This method must be worked out nominally.

Finally, `Ship` must provide a method `thrust` to change the ship's velocity based on the current velocity v, its direction θ, and on a given amount a. The new velocity of the ship (v'_x, v'_y) is derived as follows:

$$\begin{aligned} v'_x &= v_x + a.\cos(\theta) \\ v'_y &= v_y + a.\sin(\theta) \end{aligned}$$

The given amount a must never be less than zero. This method must be worked out totally (replacing a by zero, if it is less than zero). If the new velocity's speed would exceed the upper bound c, then reduce v_x and v_y such that the speed becomes c (but do not modify the new direction of the velocity). For simplicity, we assume that turning and accelerating does not take any time.

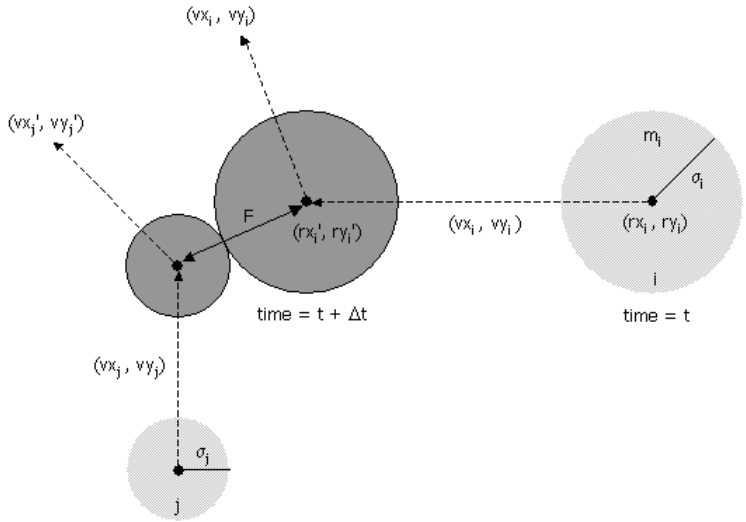
1.3 Collision Prediction

Space crafts may collide with each other and the next phases of the assignment will specify the collision behaviour. For now the vessels may pass each

other and overlap without altering their properties. Yet, you shall add functionality to detect overlapping and to predict collisions. To do so, `Ship` shall also provide the following methods:

- `getDistanceBetween` returns the distance in between two space crafts. The distance may be negative if both ships overlap. The distance between a ship and itself is zero.
- `overlap` returns `true` if and only if two space crafts overlap. A ship always overlaps with itself.
- `getTimeToCollision` shall return when (i.e. in how many seconds), if ever, two space crafts will collide. Return `Double.POSITIVE_INFINITY` if the ships never collide.
- `getCollisionPosition` shall return where, if ever, two space crafts will collide. Return `null` if the ships never collide.

Implement these methods defensively. Below we explain how collision points and collision times can be computed.



- Given the positions and velocities of two space crafts i and j at time t, we wish to determine if and when they will collide with each other.
- Let (rx'_i, ry'_i) and (rx'_j, ry'_j) denote the positions of space crafts i and j at the moment of contact, say $t + \Delta t$. When the space crafts collide, their centres are separated by a distance of $\sigma = \sigma_i + \sigma_j$. In other words: $\sigma^2 = (rx'_i - rx'_j)^2 + (ry'_i - ry'_j)^2$

- During the time prior to the collision, the space crafts move in straight-line trajectories. Thus,
  $rx'_i = rx_i + \Delta t \cdot vx_i$, $ry'_i = ry_i + \Delta t \cdot vy_i$
  $rx'_j = rx_j + \Delta t \cdot vx_j$, $ry'_j = ry_j + \Delta t \cdot vy_j$

- Substituting these four equations into the previous one, solving the resulting quadratic equation for Δt, selecting the physically relevant root, and simplifying, we obtain an expression for Δt in terms of the known positions, velocities, and radii:

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0, \\ \infty & \text{if } d \leq 0, \\ -\frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise,} \end{cases}$$

  where $d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)(\Delta r \cdot \Delta r - \sigma^2)$ and
  $\Delta r = (\Delta x, \Delta y) = (rx_j - rx_i, ry_j - ry_i)$
  $\Delta v = (\Delta vx, \Delta vy) = (vx_j - vx_i, vy_j - vy_i)$
  $\Delta r \cdot \Delta r = (\Delta x)^2 + (\Delta y)^2$
  $\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$
  $\Delta v \cdot \Delta r = (\Delta vx)(\Delta x) + (\Delta vy)(\Delta y)$.

If an intermediate computation in an algorithm implemented in one of the aforementioned methods overflows (i.e. yields $-\infty$ or ∞), then the method's return value does not have to match the answer which is mathematically (i.e. when reasoning without overflow) expected.

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 must 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 `Ship` 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 robots. 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 `Asteroids`. After starting the program, you can press keys to modify the state of the program. The command keys are `Tab` for switching space crafts, `left` and `right` arrow key to turn, `up` to accelerate, `c` to show collision points, and `Esc` to terminate the program. Be aware that the GUI displays only part of the (infinite) space. Your space crafts 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 `Ship`. 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` implementa-

tion 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 17th of March 2013 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 18th and the 29th of March. More information will be provided via Toledo.