OGP Assignment 2016-2017: Asteroids (Part II)

This text describes the second part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. Therefore, all grades are scored based on this assignment. The assignment is preferably taken in groups consisting of two students. You may however work out the solution individually. In that case you may not implement some (small) parts of the assignment, as indicated in the following section. In principle, you should compile your solutions for the second and third parts of the assignment with the partner you chose for the first part. You are, however, allowed to start working with a new partner, or to work out the rest of the project on your own. Changes must be reported to ogp-inschrijven@cs.kuleuven.be before the 26th of March 2017. If during the semester conflicts arise within a group, this should be reported to ogp-inschrijven@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

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. In total, 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.

The goal of this assignment is to test your understanding of the concepts introduced in the course. For that reason, we provide a graphical user interface for the game 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 the specified functionality, according to their best judgement. Your solution should be implemented in Java 8, satisfy all functional requirements and follow the rules described in this document. The assignment may not answer all possible questions you may have concerning the system itself (functional requirements) or concerning the way it should be worked out (non-functional requirements). You are free to fill in those details in the way that best suits your project. As an example, if the assignment does neither impose to use nominal programming, total programming, nor defensive programming in

working out some aspect of the game, you are free to choose the paradigm you prefer for that part. The ultimate goal of the project is to convince us that you master all the underlying concepts of object-oriented programming. Specifically, the goal of this exercise is not to hand in the best possible arcade game. Therefore, your grades do not depend on correctly implementing functional requirements only; we will pay attention to documentation, accurate specifications, re-usability and adaptability as well. After handing in your solution to the first part of the assignment, you will receive feedback on your submission. After handing in the third part of this assignment, the entire solution must be defended in front of 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 a 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*.

1 Assignment

Asteroids is an arcade game where the player controls a spacecraft 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 focused on a single class `Ship`. In the second part, we extend `Ship` (changes marked in blue), and we add game `World`s and `Bullet`s. As before, your solution may contain additional helper classes (in particular classes marked *@Value*). If the assignment does not specify how certain aspects of the game are to be worked out, you may select the option you prefer. You may also use inheritance as you see fit.

In the remainder of this section, we describe the classes `Ship`, `Bullet` and `World` in more detail. Unless specified otherwise, all aspects of your implementation of `Ship` shall be documented both formally and informally.

The classes `Bullet` and `World` must be documented in a formal way only.

If the formal documentation of a method is basically the same as its implementation, you may replace the documentation with a simple reference to the implementation. For example:

```
/*
 * Return the Body Mass Index of this person.
 * @see implementation
 */
public getBMI() {
  return this.getWeight()/Math.pow(this.getLength(),2);
}
```

Be careful: If a more declarative specification of a method is possible, we expect that documentation and not a copy of the method's implementation.

1.1 Game World – The Class `World`

A game world is a two-dimensional, rectangular area containing ships and bullets.

Each world has a particular size, described by a width and height expressed in kilometres (km). The size of a world cannot change after construction. Both the width and height must lie in the range 0 to `Double.MAX_VALUE` (both inclusive), for all worlds. In the future, the upper bound on the width and height may decrease. However, all worlds will share the same upper bounds. All methods involving the size of a world must be worked out in a total way. The world defines a two-dimensional coordinate space with $x \in [0, \mathsf{width}]$ and $y \in [0, \mathsf{height}]$.

A world contains ships and bullets. At all times, each ship or bullet is located in at most one world. No world contains the same ship or bullet twice. Ships and bullets are circular entities. If such an entity is located in a world, then the circle must lie fully within the bounds of that world and it shall not overlap with other ships or bullets within that world. Exceptions to this rule involve the interaction of bullets with the ship that has fired these bullets, which are explained in the following sections.

To account for rounding issues with the **double**-representation of real numbers in Java (cf. Sec. 2), your implementation of Asteroids shall employ a notion of *significant overlap* between two objects whenever checks for overlap of two objects are required: Two objects A and B overlap significantly if the distance between the centres of A and B is $\leq 99\%$ of the sum of the objects' radii σ_A and σ_B.

Similarly, your implementation shall employ a notion of *apparently within boundaries* of an object inside a a container whenever checks are needed to verify that the contained object is fully within the boundaries of its containing object. An object A apparently lies within the boundaries of a container C if the distance between each boundary of C and the centre of A is $\geq 99\%$ of the radius of A.

The class `World` shall provide methods for adding and removing ships and bullets. Those methods must be worked out defensively.

The class `World` shall also provide a method to return the ship or the bullet, if any, at a given position, i.e. the objects whose centre coincides with the given position. This method must return its result in nearly constant time and must be worked out in a total way.

Finally, it must be possible to query a game world for the set of all ships, the set of all bullets as well as the set of all entities (i.e. all ships and all bullets) in a given world.

1.1.1 Advancing Time

The state of a world evolves as time passes. For example, the position of an entity changes over time if the object is flying through space at a non-zero velocity. Similarly, an entity's velocity can change because of collisions.

The class `World` should provide a method `evolve`, which advances the state of the world a certain number of seconds Δt. You must not work out a specification for this method. Implement this method in a defensive way as follows:

1. Predict the first collision C (i.e. the collision that happens before all other collisions, cf. Sec. 1.4).
2. Suppose C happens in t_C seconds. If t_C is larger than Δt, go to step 5. Otherwise, advance all ships and bullets t_C seconds (to the time right before the first collision).
3. Resolve C (cf. Sec. 1.4.2)
4. Subtract t_C from Δt and go to step 1.
5. Advance all ships and bullets Δt seconds, as explained in the following sections.

Advancing a ship or bullet entails updating its position based on its current velocity. If a ship's thruster is enabled, additionally modify its velocity (*after* updating its position in steps 2 and 5 above) based on its acceleration. Section 1.2 explains how the new velocity of the ship (v'_x, v'_y) is derived.

1.2 The Class `Ship`

1.2.1 Position, Velocity, Orientation and Radius

Each spaceship is located at a certain position (x, y). A ship may be associated with a specific `World` under the constraints specified in Sec. 1.1. If a ship is not associated with a world, it is positioned in an unbounded 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 has an orientation, i.e., it faces a certain direction expressed as an angle in radians. For example, the orientation 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 $3\pi/2$. The orientation of a ship must always be a value in between 0 and 2π. The class `Ship` must provide a method to turn the ship by adding a given angle to the current orientation. All aspects related to the orientation must be worked out nominally.

The shape of each ship is a circle with radius σ (expressed in kilometres) centred on the ship's position. The radius of a ship must be larger than 10 km. It 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 positive. All aspects related to the radius must be worked out defensively.

Each ship has a mass expressed in kilograms (kg). The mass of a ship determines its resistance to acceleration and the effect of collisions with other ships. The mass density ρ of each ship is at least $1.42 \cdot 10^{12} kg/km^3$. Furthermore, the mass m of a ship satisfies:

$$m \geq \frac{4}{3}\pi\sigma^3\rho$$

The mass of a ship may change during its lifetime. You may assume that the mass of a ship will always be much smaller than `Double.MAX_VALUE` in Java. In addition, each ship has a total mass which equals the ship's mass plus the mass of objects carried by that ship. All aspects related to a ship's mass and total mass must be worked out totally.

Conceptually, all of the above characteristics of a ship are real numbers.

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

1.2.2 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 spacecraft based on the current position, velocity and a given time duration Δt. The given duration Δt shall never be less than zero. If the given duration is zero or the ship's velocity is zero, the ship shall keep its current position. 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 orientation. This method must be worked out nominally.

Each ship has a thruster. The thruster is either enabled or disabled. When the thruster is enabled, the space ship accelerates in the direction it is facing. Thus, `Ship` must provide the method `thrustOn` and `thrustOff` to enable and respectively disable the thruster. Enabling thrusters shall result in a change the ship's velocity based on the current velocity v, its orientation θ, and on an acceleration a. For now, each ship is equipped with the same kind of thruster. An active thruster exerts a force $F = 1.1 \cdot 10^{21}$ Newton on its ship. In future iterations, the value of F may not remain the same for each ship. The acceleration generated by the thruster can be derived from Newton's second law of motion ($F = ma$). The new velocity of the ship (v'_x, v'_y) is derived as follows:

$$\begin{aligned} v'_x &= v_x + a \cdot \cos(\theta) \cdot \Delta t \\ v'_y &= v_y + a \cdot \sin(\theta) \cdot \Delta t \end{aligned}$$

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 magnitude 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.

The class `Ship` shall provide methods to inspect the state of the thruster and the ship's acceleration.

1.2.3 Interactions with Bullets

Ships can be hit by bullets, load bullets, and fire bullets (cf. Sec. 1.3 and Sec. 1.4). Similar to ships, `Bullets` are circular entities with a position and a radius.

If a bullet is loaded on a ship, its circle must lie fully within the bounds of that ship. The bullet may, however, overlap with other bullets loaded on that same ship. The class `Ship` must provide methods to extend its collection of bullets with a single bullet as well as with an arbitrary series of bullets.[1] Upon creation, each ship shall not have any bullets loaded ~~have 15 bullets loaded, the radius of these bullets shall be greater than 10% or the radius of the ship~~.

Whenever a ship fires a bullet, the bullet is removed from the ship's collection of bullets. All methods related to the collection of bullets and the total number of bullets shall be worked out in a defensive way.

Bullets influence the total mass of a ship. The total mass of a ship is equal to the sum of that ship's mass and the mass of all bullets loaded on it.

At all times, ships that are positioned within a world can fire one of their bullets. The initial speed of such a bullet is $250km/s$ and the direction of its velocity is equal to the direction the ship is facing[2]. The bullet is initially placed next to the firing ship and so that the ship and the bullet do not overlap ~~and the space between the firing ship and the bullet is less than the average of that ship's radius and the bullet's radius~~ $\frac{\sigma_{Ship}+\sigma_{Bullet}}{2}$, at the angle the ship is facing. However, if the bullet's initial position is already (partially) occupied by another entity, then the bullet immediately collides with that entity. If the bullet would be located (partially) outside of the ship's world, it is not added to the ship's world but destroyed immediately. Ships that are not positioned in a world cannot fire bullets. The method to fire a bullet must be worked out totally.

For students working alone: You may replace the collection of bullets by a counter of the total number of bullets available on the ship. If you add a bullet to a ship or remove a bullet from a ship, you must ensure that the bullet fully overlaps with the ship. If so, the bullet counter must be incremented, respectively decremented, and the bullet shall be destroyed. You are not required to work out a method to add an arbitrary series of bullets to a ship. For computing the total mass of a ship's collection of bullets, you can assume that all bullets have a radius of $3km$. Each time a ship fires a bullet, your implementation shall create a new bullet with a radius of $3km$, and position it as described above.

[1] Use a method with variable arguments for this purpose to load multiple bullets.

[2] Note that the velocity of the ship itself has no influence on the initial velocity of its bullets. Even though this is not physically correct, it is fine for our game.

1.3 The Class `Bullet`

Each bullet is located at a certain position (x, y). A bullet may be associated with a specific `Ship`, either being loaded on that ship or being fired by that ship. A bullet may be associated with a specific `World` under the constraints specified in Sec. 1.1. A bullet cannot be held by a world and a ship at the same time, nor can it be loaded on several ships at the same time. If a bullet is not loaded on a ship or associated with a world, it is positioned in an unbounded two-dimensional space. Both x and y are expressed in kilometres (km). All aspects related to the position of a bullet shall be worked out defensively.

Each bullet has velocities v_x and v_y (even if it is not associated with a world) that determine the bullet'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 bullet, 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 bullet, but it will always be less than or equal to c. All aspects related to velocity must be worked out in a total manner.

The shape of each bullet is a circle with finite radius σ (expressed in kilometres) centred on the bullet's position. The radius of a bullet must be larger than 1 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 bullet and its value will be positive. All aspects related to the radius must be worked out defensively.

Each bullet has a mass expressed in kilograms (kg). For simplicity, we assume the mass density ρ of each bullet is the same, namely $7.8 \cdot 10^{12} kg/km^3$. The mass m of an bullet can hence be computed as follows:

$$m = \frac{4}{3}\pi r^3 \rho$$

where r is the radius of the bullet. You may assume that the mass of a bullet will always be much smaller than `Double.MAX_VALUE` in Java. All aspects related to the mass and mass density must be worked out totally.

The class `Bullet` shall provide methods to inspect the position, velocity, radius, density and mass. In addition, the class should provide a method that returns its source, if any, i.e. the ship that fired the bullet.

1.4 Collisions

Ships and bullets that are positioned in the same world may collide with each other and with the boundaries of the world they are in. Ships and bullets that are not positioned in a world, do not collide. Your solution should

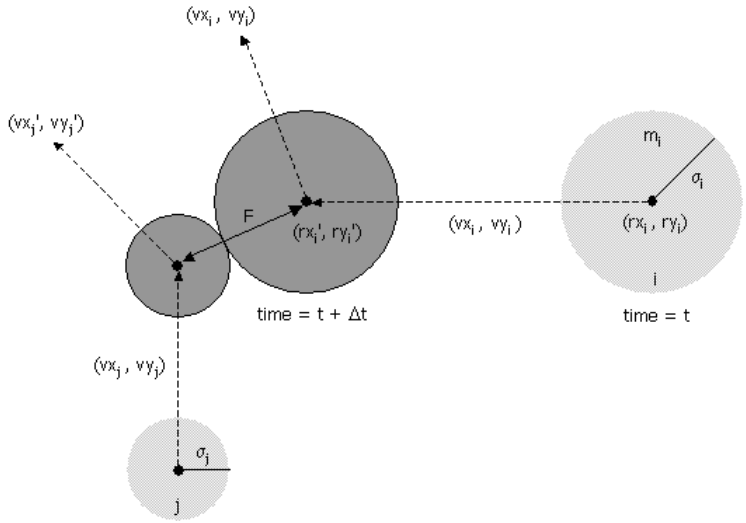
include methods to predict and resolve collisions. To account for rounding issues with the **double**-representation of real numbers in Java (cf. Sec. 2), your implementation of Asteroids shall employ a notion of *apparently collide* whenever checks for collisions of two objects are required: Two objects A and B apparently collide if the distance between the centres of A and B is between 99% and 101% of the sum of the objects' radii σ_A and σ_B.

1.4.1 Collision Prediction

Entities (i.e., spacecrafts and bullets) may collide with each other and the following section specifies the collision behaviour. You must add functionality to detect overlapping and to predict collisions. To do so, `Ship` shall provide the following methods:

- `getDistanceBetween` returns the distance in between two entities. The distance may be negative if both entities overlap. The distance between an entity and itself is zero.
- `overlap` returns `true` if and only if two entities overlap. An entity always overlaps with itself.
- `getTimeToCollision` shall return when (i.e. in how many seconds), if ever, two entities will collide. `getTimeToCollision` shall return `Double.POSITIVE_INFINITY` if the entities never collide. This method does not apply to entities that overlap.
- `getCollisionPosition` shall return where, if ever, two entities will collide. The method shall return `null` if the entities never collide. This method does not apply to entities that overlap.

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



- Given the positions and velocities of two entities 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 spacecrafts i and j at the moment of contact, say $t + \Delta t$. When the spacecrafts 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 spacecrafts 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
  $\Delta r = (\Delta x, \Delta y) = (rx_j - rx_i, ry_j - ry_i)$
  $\Delta v = (\Delta vx, \Delta vy) = (vx_j - vx_i, vy_j - vy_i)$
  $\Delta r \cdot \Delta r = (\Delta x)^2 + (\Delta y)^2$
  $\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$
  $\Delta v \cdot \Delta r = (\Delta vx)(\Delta x) + (\Delta vy)(\Delta y)$.

We expect a declarative specification of the method `getTimeToCollision`. This means that you must specify conditions to be satisfied by the time returned by the method instead of repeating the implementation as part of the specification. In other words, your specification should not be a mere copy of the body of the method. You must only work out a specification for the case in which the method returns a finite value.

Ships and bullets collide not only with each other but also with the boundaries of the space they are in. Your solution should include one or more methods to determine when (i.e. in how many seconds) and where (i.e. which position at the boundaries of the world), a ship or bullet collides with a boundary (if ever), or with some other entity (if ever). All these methods must be worked out in total way, returning infinity (for time and positions), respectively `null` (for entities) if no collision will ever occur.

Predicted collisions between two entities or between an entity and the boundaries of the world are only correct if the movement of these entities is not influenced by other entities nor by collisions they may have with the boundaries of the world they are in. The methods to predict collisions will not take such influences into account. In other words, a predicted collision is only guaranteed to be correct if it is the first collision that will take place in the world, and if no other ships or bullets are added to that world in the meanwhile.

1.4.2 Resolving Collisions

Resolve collisions as follows:

- When two ships collide, they bounce off each other. That is, the velocity of both ships is updated to reflect the collision.
- When a bullet hits its own ship, the bullet is reloaded on the ship, with its position equal to the centre of the ship.
- When a bullet hits another ship or another bullet, both the bullet and the other ship or other bullet die.
- Ships and bullets bounce off boundaries. More specifically, when an entity collides with a horizontal boundary (e.g. the top of the world), negate the y component of its velocity. For example, when a ship with velocity $(5, 8)$ collides with the top boundary, change its velocity to $(5, -8)$. When an entity collides with a vertical boundary, negate the x component of its velocity.
- When a bullet collides with a boundary for the third time, the bullet dies. That is, a bullet bounces off boundaries only twice. In the future,

the maximum number of boundary bounces may vary from bullet to bullet.

When two ships collide, they bounce off each other. The new velocity of the ships i and j can be computed as follows:

$$(v'^i_x, v'^i_y) = (v^i_x + J_x/m_i, v^i_y + Jy/m_i)$$

$$(v'^j_x, v'^j_y) = (v^j_x - J_x/m_j, v^j_y - Jy/m_j)$$

where

$$J_x = \frac{J\Delta x}{\sigma}$$

$$J_y = \frac{J\Delta y}{\sigma}$$

$$J = \frac{2m_i m_j(\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

Note that the mass of an object influences collisions resolution.

1.5 Death

Ships and bullets can die when they collide with each other or with the boundaries of their world. When an entity dies, it is removed from its world (if any). It shall further be possible to destroy `Worlds`. When a world is destroyed all its ships and bullets are removed from it. The ships and bullets themselves are not destroyed. However, they no longer belong to a world.

2 Storing and Manipulating Real Numbers as Floating-Point Numbers

In your program, you shall use type **double** as the type for variables that conceptually need to be able to store arbitrary real numbers, and as the return type for methods that conceptually need to be able to return arbitrary real numbers.

Note, however, that variables of type **double** can only store values that are in a particular subset of the real numbers (specifically: the values that can be written as $m \cdot 2^e$ where $m, e \in \mathbb{Z}$ and $|m| < 2^{53}$ and $-1074 \leq e \leq 970$), as well as positive infinity (written as `Double.POSITIVE_INFINITY`) and negative infinity (written as `Double.NEGATIVE_INFINITY`). These variables can additionally store some special values called *Not-a-Number* values, which are used as the result of operations whose value is mathematically undefined

such as 0/0; see method `Double.isNaN`. Therefore, arithmetic operations on expressions of type **double**, whose result type is also **double**, must generally perform *rounding* of their mathematically correct value to obtain a result value of type **double**. For example, the result of the Java expression `1.0/5.0` is not the number 0.2, but the number[3]

0.2000000000000000111022302462515654042363166809082031 25

When performing complex computations in type **double**, rounding errors can accumulate and become arbitrarily large. The art and science of analysing computations in floating-point types (such as **double**) to determine bounds on the resulting error is studied in the scientific field of *numerical analysis*.

However, numerical analysis is outside the scope of this course; therefore, for this assignment we will be targeting not Java but *idealised Java*, a programming language that is entirely identical to Java except that in idealised Java, the values of type **double** are exactly the extended real numbers plus some nonempty set of *Not-a-Number* values:

$$\textbf{double} = \mathbb{R} \cup \{-\infty, +\infty\} \cup \mathit{NaNs}$$

Therefore, in idealised Java, operations in type **double** perform no rounding and have the same meaning as in regular mathematics. Your solution should be correct when interpreting both your code and your formal documentation as statements and expressions of idealised Java.

So, this means that for reasoning about the correctness of your program you can ignore rounding issues. However, when testing your program, of course you cannot ignore these. The presence of rounding means that it is unrealistic to expect that when you call your methods in your test cases, they will produce the exact correct result. Instead of testing for exactly correct results, it makes more sense to test that the results are within an acceptable distance from the correct result. What "acceptable distance" means, depends on the particular case. For example, in many cases, for a nonzero expected value, if the relative error (the value $|r-e|/|e|$ where r and e are the observed and expected results, respectively) is less than 0.01%, then that is an acceptable result. You can use JUnit's `assertEquals(double, double, double)` method to test for an acceptable distance.

3 Testing

Write a JUnit test suite tests all public methods of your implementation. Include this test suite in your submission.

[3]You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`. Rounding also affects literals, e.g., a `new BigDecimal(0.2)` will not equal the literal `0.2`.

4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on spaceships. The user interface is included in the assignment as a JAR file. Contrary to the GUI for part 1, you should import this JAR file as an archive into the root your existing project from part 1. This will only overwrite resources in folder `src-provided`. That folder contains the source code of the user interface and further helper classes. Generally, the files in this folder require no modification from your side. The classes that you develop must be placed in the folders `src` (implementation classes) and `tests` (test classes).

To connect your implementation to the GUI, update your class `Facade` in package `asteroids.facade` to implement the provided interface `IFacade` from package `asteroids.part2.facade`. This interface extends the IFacade interface from part 1 with additional methods (and deprecates some others). `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, run the `main` method of the provided class `Part2` in package `asteroids.part2`. After starting the program, you can press keys to modify the state of the program. The command keys are ← and → to turn, ↑ to enable/disable the thruster, Space to fire, C to show collision points, and Esc to terminate the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment are the classes described in this assignment. 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 17th of April 2017 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!